Lukas Iffländer

Attack-aware Security Function Management

Dissertation, Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik, 2020

Gutachter: Prof. Dr. Samuel Kounev, JMU Würzburg,
Prof. Dr. Nuno Manuel dos Santos Antunes, Universidade de Coimbra

Tag der mündlichen Prüfung: 28.01.2021

Abstract

Over the last decades, cybersecurity has become an increasingly important issue. Between 2019 and 2011 alone, the losses from cyberattacks in the United States grew by 6217%. At the same time, attacks became not only more intensive but also more and more versatile and diverse. Cybersecurity has become everyone's concern. Today, service providers require sophisticated and extensive security infrastructures comprising many security functions dedicated to various cyberattacks. Still, attacks become more violent to a level where infrastructures can no longer keep up. Simply scaling up is no longer sufficient. To address this challenge, in a whitepaper, the Cloud Security Alliance (CSA) proposed multiple work packages for security infrastructure, leveraging the possibilities of Software-defined Networking (SDN) and Network Function Virtualization (NFV).

Security functions require a more sophisticated modeling approach than regular network functions. Notably, the property to drop packets deemed malicious has a significant impact on Security Service Function Chains (SSFCs)—service chains consisting of multiple security functions to protect against multiple attack vectors. Under attack, the order of these chains influences the end-to-end system performance depending on the attack type. Unfortunately, it is hard to predict the attack composition at system design time. Thus, we make a case for dynamic attack-aware SSFC reordering. Also, we tackle the issues of the lack of integration between security functions and the surrounding network infrastructure, the insufficient use of short term CPU frequency boosting, and the lack of Intrusion Detection and Prevention Systems (IDPS) against database ransomware attacks.

Current works focus on characterizing the performance of security functions and their behavior under overload without considering the surrounding infrastructure. Other works aim at replacing security functions using network infrastructure features but do not consider integrating security functions within the network. Further publications deal with using SDN for security or how to deal with new vulnerabilities introduced through SDN. However, they do not take security function performance into account. NFV is a popular field for research dealing with frameworks, benchmarking methods, the combination with SDN, and implementing security functions as Virtualized Network

Functions (VNFs). Research in this area brought forth the concept of Service Function Chains (SFCs) that chain multiple network functions after one another. Nevertheless, they still do not consider the specifics of security functions. The mentioned CSA whitepaper proposes many valuable ideas but leaves their realization open to others.

This thesis presents solutions to increase the performance of single security functions using SDN, performance modeling, a framework for attack-aware SSFC reordering, a solution to make better use of CPU frequency boosting, and an IDPS against database ransomware.

Specifically, the primary contributions of this work are:

- We present approaches to dynamically bypass Intrusion Detection Systems (IDS) in order to increase their performance without reducing the security level. To this end, we develop and implement three SDN-based approaches (two dynamic and one static).

  We evaluate the proposed approaches regarding security and performance and show that they significantly increase the performance compared to an inline IDS without significant security deficits. We show that using software switches can further increase the performance of the dynamic approaches up to a point where they can eliminate any throughput drawbacks when using the IDS.

- We design a DDoS Protection System (DPS) against TCP SYN flood attacks in the form of a VNF that works inside an SDN-enabled network. This solution eliminates known scalability and performance drawbacks of existing solutions for this attack type.

  Then, we evaluate this solution showing that it correctly handles the connection establishment and present solutions for an observed issue. Next, we evaluate the performance showing that our solution increases performance up to three times. Parallelization and parameter tuning yields another 76% performance boost. Based on these findings, we discuss optimal deployment strategies.

- We introduce the idea of attack-aware SSFC reordering and explain its impact in a theoretical scenario. Then, we discuss the required information to perform this process.

  We validate our claim of the importance of the SSFC order by analyzing the behavior of single security functions and SSFCs. Based on the results, we conclude that there is a massive impact on the performance up to three orders of magnitude, and we find contradicting optimal orders for different workloads. Thus, we demonstrate the need for dynamic reordering.

Last, we develop a model for SSFC regarding traffic composition and resource demands. We classify the traffic into multiple classes and model the effect of single security functions on the traffic and their generated resource demands as functions of the incoming network traffic. Based on our model, we propose three approaches to determine optimal orders for reordering.

- We implement a framework for attack-aware SSFC reordering based on this knowledge. The framework places all security functions inside an SDN-enabled network and reorders them using SDN flows.

  Our evaluation shows that the framework can enforce all routes as desired. It correctly adapts to all attacks and returns to the original state after the attacks cease. We find possible security issues at the moment of reordering and present solutions to eliminate them.

- Next, we design and implement an approach to load balance servers while taking into account their ability to go into a state of Central Processing Unit (CPU) frequency boost. To this end, the approach collects temperature information from available hosts and places services on the host that can attain the boosted mode the longest.

  We evaluate this approach and show its effectiveness. For high load scenarios, the approach increases the overall performance and the performance per watt. Even better results show up for low load workloads, where not only all performance metrics improve but also the temperatures and total power consumption decrease.

- Last, we design an IDPS protecting against database ransomware attacks that comprise multiple queries to attain their goal. Our solution models these attacks using a Colored Petri Net (CPN).

  A proof-of-concept implementation shows that our approach is capable of detecting attacks without creating false positives for benign scenarios. Furthermore, our solution creates only a small performance impact.

Our contributions can help to improve the performance of security infrastructures. We see multiple application areas from data center operators over software and hardware developers to security and performance researchers. Most of the above-listed contributions found use in several research publications.

Regarding future work, we see the need to better integrate SDN-enabled security functions and SSFC reordering in data center networks. Future SSFC should discriminate between different traffic types, and security frameworks should support automatically learning models for security functions. We see the need to consider energy efficiency when regarding SSFCs and take CPU boosting technologies into account when designing performance models as well as placement, scaling, and deployment strategies. Last, for a faster adaptation against recent ransomware attacks, we propose machine-assisted learning for database IDPS signatures.

Zusammenfassung

In den letzten Jahrzehnten wurde Cybersicherheit zu einem immer wichtigeren Thema. Allein zwischen 2019 und 2011 stiegen die Verluste durch Cyberattacken in den Vereinigten Staaten um 6217%. Gleichzeitig wurden die Angriffe nicht nur intensiver, sondern auch immer vielseitiger und facettenreicher. Cybersicherheit ist zu einem allgegenwärtigen Thema geworden. Heute benötigen Dienstleistungsanbieter ausgefeilte und umfassende Sicherheitsinfrastrukturen, die viele Sicherheitsfunktionen für verschiedene Cyberattacken umfassen. Dennoch werden die Angriffe immer heftiger, so dass diese Infrastrukturen nicht mehr mithalten können. Ein einfaches Scale-Up ist nicht mehr ausreichend. Um dieser Herausforderung zu begegnen, schlug die Cloud Security Alliance (CSA) in einem Whitepaper mehrere Arbeitspakete für Sicherheitsinfrastrukturen vor, die die Möglichkeiten des Software-definierten Netzwerks (SDN) und der Netzwerkfunktionsvirtualisierung (NFV) nutzen.

Sicherheitsfunktionen erfordern einen anspruchsvolleren Modellierungsansatz als normale Netzwerkfunktionen. Vor allem die Eigenschaft, als bösartig erachtete Pakete fallen zu lassen, hat erhebliche Auswirkungen auf Security Service Function Chains (SSFCs) - Dienstketten, die aus mehreren Sicherheitsfunktionen zum Schutz vor mehreren Angriffsvektoren bestehen. Bei einem Angriff beeinflusst die Reihenfolge dieser Ketten je nach Angriffstyp die Gesamtsystemleistung. Leider ist es schwierig, die Angriffszusammensetzung zur Designzeit vorherzusagen. Daher plädieren wir für eine dynamische, angriffsbewusste Neuordnung der SSFC. Außerdem befassen wir uns mit den Problemen der mangelnden Integration zwischen Sicherheitsfunktionen und der umgebenden Netzwerkinfrastruktur, der unzureichenden Nutzung der kurzfristigen CPU-Frequenzverstärkung und des Mangels an Intrusion Detection and Prevention Systems (IDPS) zur Abwehr von Datenbank-Lösegeldangriffen.

Bisherige Arbeiten konzentrieren sich auf die Charakterisierung der Leistungsfähigkeit von Sicherheitsfunktionen und deren Verhalten bei Überlastung ohne Berücksichtigung der umgebenden Infrastruktur. Andere Arbeiten zielen darauf ab, Sicherheitsfunktionen unter Verwendung von Merkmalen der Netzwerkinfrastruktur zu ersetzen, berücksichtigen aber nicht die Integration von Sicherheitsfunktionen innerhalb des Netzwerks. Weitere Publika-

tionen befassen sich mit der Verwendung von SDN für die Sicherheit oder mit dem Umgang mit neuen, durch SDN eingeführten Schwachstellen. Sie berücksichtigen jedoch nicht die Leistung von Sicherheitsfunktionen. Die NFV-Domäne ist ein beliebtes Forschungsgebiet, das sich mit Frameworks, Benchmarking-Methoden, der Kombination mit SDN und der Implementierung von Sicherheitsfunktionen als Virtualized Network Functions (VNFs) befasst. Die Forschung in diesem Bereich brachte das Konzept der Service-Funktionsketten (SFCs) hervor, die mehrere Netzwerkfunktionen nacheinander verketten. Dennoch berücksichtigen sie noch immer nicht die Besonderheiten von Sicherheitsfunktionen. Zu diesem Zweck schlägt das bereits erwähnte CSA-Whitepaper viele wertvolle Ideen vor, lässt aber deren Realisierung anderen offen.

In dieser Arbeit werden Lösungen zur Steigerung der Leistung einzelner Sicherheitsfunktionen mittels SDN, Performance Engineering, Modellierung und ein Rahmenwerk für die angriffsbewusste SSFC-Neuordnung, eine Lösung zur besseren Nutzung der CPU-Frequenzsteigerung und ein IDPS gegen Datenbank-Lösegeld.

Im Einzelnen sind die sechs Hauptbeiträge dieser Arbeit:

- Wir stellen Ansätze zur dynamischen Umgehung von Intrusion-Detection-Systemen (IDS) vor, um deren Leistung zu erhöhen, ohne das Sicherheitsniveau zu senken. Zu diesem Zweck entwickeln und implementieren wir drei SDN-basierte Ansätze (zwei dynamische und einen statischen).

  Wir evaluieren sie hinsichtlich Sicherheit und Leistung und zeigen, dass alle Ansätze die Leistung im Vergleich zu einem Inline-IDS ohne signifikante Sicherheitsdefizite signifikant steigern. Wir zeigen ferner, dass die Verwendung von Software-Switches die Leistung der dynamischen Ansätze weiter steigern kann, bis zu einem Punkt, an dem sie bei der Verwendung des IDS etwaige Durchsatznachteile beseitigen können.

- Wir entwerfen ein DDoS-Schutzsystem (DPS) gegen TCP-SYN-Flutangriffe in Form eines VNF, das innerhalb eines SDN-fähigen Netzwerks funktioniert. Diese Lösung eliminiert bekannte Skalierbarkeits- und Leistungsnachteile bestehender Lösungen für diesen Angriffstyp.

  Dann bewerten wir diese Lösung und zeigen, dass sie den Verbindungsaufbau korrekt handhabt, und präsentieren Lösungen für ein beobachtetes Problem. Als nächstes evaluieren wir die Leistung und zeigen, dass unsere Lösung die Leistung bis zum Dreifachen erhöht. Durch Parallelisierung und Parameterabstimmung werden weitere 76%

der Leistung erzielt. Auf der Grundlage dieser Ergebnisse diskutieren wir optimale Einsatzstrategien.

- Wir stellen die Idee der angriffsbewussten Neuordnung des SSFC vor und erläutern deren Auswirkungen anhand eines theoretischen Szenarios. Dann erörtern wir die erforderlichen Informationen zur Durchführung dieses Prozesses.

  Wir validieren unsere Behauptung von der Bedeutung der SSFC-Ordnung, indem wir das Verhalten einzelner Sicherheitsfunktionen und SSFCs analysieren. Aus den Ergebnissen schließen wir auf eine massive Auswirkung auf die Leistung bis zu drei Größenordnungen, und wir finden widersprüchliche optimale Aufträge für unterschiedliche Arbeitsbelastungen. Damit beweisen wir die Notwendigkeit einer dynamischen Neuordnung.

  Schließlich entwickeln wir ein Modell für den SSFC hinsichtlich der Verkehrszusammensetzung und des Ressourcenbedarfs. Dazu klassifizieren wir den Datenverkehr in mehrere Klassen und modellieren die Auswirkungen einzelner Sicherheitsfunktionen auf den Datenverkehr und die von ihnen erzeugten Ressourcenanforderungen als Funktionen des eingehenden Netzwerkverkehrs. Auf der Grundlage unseres Modells schlagen wir drei Ansätze zur Berechnung der gewünschten Reihenfolge für die Neuordnung vor.

- Auf der Grundlage dieses Wissens implementieren wir einen Rahmen für die angriffsbewusste SSFC-Neuordnung. Das Rahmenwerk platziert alle Sicherheitsfunktionen innerhalb eines SDN-fähigen Netzwerks und ordnet sie mit Hilfe von SDN-Flüssen neu an.

  Unsere Auswertung zeigt, dass das Rahmenwerk alle Routen wie gewünscht durchsetzen kann. Es passt sich allen Angriffen korrekt an und kehrt nach Beendigung der Angriffe in den ursprünglichen Zustand zurück. Wir finden mögliche Sicherheitsprobleme zum Zeitpunkt der Neuordnung und präsentieren Lösungen zu deren Beseitigung.

- Als Nächstes entwerfen und implementieren wir einen Ansatz zum Lastausgleich von Servern hinsichtlich ihrer Fähigkeit, in einen Zustand der Frequenzerhöhung der Zentraleinheit (CPU) zu gehen. Zu diesem Zweck sammelt der Ansatz Temperaturinformationen von verfügbaren Hosts und platziert den Dienst auf dem Host, der den verstärkten Modus am längsten erreichen kann.

Wir evaluieren diesen Ansatz und zeigen seine Funktionalität auf. Für Hochlastszenarien erhöht der Ansatz die Gesamtleistung und steigert die Leistung pro Watt. Noch bessere Ergebnisse zeigen sich bei Niedriglast-Workloads, wo sich nicht nur alle Leistungsmetriken verbessern, sondern auch die Temperaturen und der Gesamtstromverbrauch sinken.

- Zuletzt entwerfen wir ein IDPS, das vor Lösegeld-Angriffen auf Datenbanken schützt, die mehrere Abfragen umfassen, um ihr Ziel zu erreichen. Unsere Lösung modelliert diese Angriffe mit einem Colored Petri Net (CPN).

  Eine Proof-of-Concept-Implementierung zeigt, dass unser Ansatz in der Lage ist, die beobachteten Angriffe zu erkennen, ohne für gutartige Szenarien falsch positive Ergebnisse zu erzeugen. Darüber hinaus erzeugt unsere Lösung nur eine geringe Auswirkung auf die Leistung.

Unsere Beiträge können dazu beitragen, die Leistungsfähigkeit von Sicherheitsinfrastrukturen zu erhöhen. Wir sehen vielfältige Anwendungsbereiche, von Rechenzentrumsbetreibern über Software- und Hardwareentwickler bis hin zu Sicherheits- und Leistungsforschern. Die meisten der oben aufgeführten Beiträge fanden in mehreren Forschungspublikationen Verwendung.

Was die zukünftige Arbeit betrifft, so sehen wir die Notwendigkeit, bessere SDN-fähige Sicherheitsfunktionen und SSFC-Neuordnung in Rechenzentrumsnetzwerke zu integrieren. Künftige SSFC sollten zwischen verschiedenen Verkehrsarten unterscheiden, und Sicherheitsrahmen sollten automatisch lernende Modelle für Sicherheitsfunktionen unterstützen. Wir sehen den Bedarf, bei der Betrachtung von SSFCs die Energieeffizienz zu berücksichtigen und bei der Entwicklung von Leistungsmodellen sowie Platzierungs-, Skalierungs- und Bereitstellungsstrategien CPU-verstärkende Technologien in Betracht zu ziehen. Schließlich schlagen wir für eine schnellere Anpassung an die jüngsten Lösegeld-Angriffe maschinengestütztes Lernen für Datenbank-IDPS-Signaturen vor.

Acknowledgements

This thesis would not have been possible without the help and support of a significant number of people. I want to thank every one of them.

First, I want to thank my supervisor, Professor Samuel Kounev, without whom I might never have started my Ph.D. project. We share over five years of a very constructive and productive relationship that culminated in this thesis.

Next, I want to express my appreciation for four people at my workplace, who, while not directly involved in my research, made a massive contribution to my academic success. Professor Wolff von Gudenberg brought me to this chair and employed me for five years. Without him, I might not have come in the sight of my advisor. Fritz Kleemann, our administrator, and network manager was always there to discuss tech issues and allowed me and my colleagues to carry out our research unhampered. Susanne Stenglin, our secretary, did her best to keep us out of the jungle of our university's administration. Martina Janousch was not just our cleaning lady but also an honorary psychologist for many of us.

I want to thank my current and former colleagues for many useful inputs into my research, fruitful discussions, and a marvelous time: Nikolas Herbst, Christian Krupitzer, Simon Spinner, Piotr Rygielski, Aleksandar Milenkoski, Jürgen Walter, Jóakim von Kistowski, André Bauer, Simon Eismann, Norbert Schmitt, Johannes Grohmann, Dennis Kaiser, Veronika Lesch, Marwin Züfle, Thomas Prantl, Robert Leppich, Steffan Herrnleben, Lukas Beierlieb, Christoph Sendner, Christoph Hagen, André Greubel and Alexandra Dmitrienko. Many of them became my co-authors.

Regarding co-authors, I also received great support from Klaus-Dieter Lange, and Nishant Rawtani at HPE research, and the same goes for the SPEC RG security research group of which I want to name Aleksandar Milenkoski, Nuno Antunes, and Marco Viera.

Many work-packages of my thesis received support from my graduate workers and research assistants. I was lucky to supervise seven Bachelor's theses by Hayreddin Ciner, Ala Eddine Ben Yahya, Jan-Philipp Heilmann, Lukas Beierlieb, Nicolas Fella, Samuel Metzler, and Ariane Geiger and seven Master's theses by Xiaofen Liu, Christina Hempfling, Jonathan Stoll, Alexander Leonhardt, Michael Jobst, Andreas Knapp, and Lukas Beierlieb. Furthermore, I supervised two additional teachers' theses by Rebecca Otto and Katja Frauendorfer and two practical theses by Christina Hempfling and Lukas Beierlieb. Nicolas Fella and Marc-Philipp Knechtle supported my work as research assistants.

I also want to attribute my professional and scientific results to the people that kept my back in my personal life. My parents, Petra and Bernhard Iffländer, have always supported me unconditionally throughout every stage of my studies. My girlfriend, Anja Schmotz, shared many happy moments of success with me but also helped me through the times when I was not sure if I would succeed in this project.

I'm grateful for the support I received from the Stiftung der Deutschen Wirtschaft. Throughout my scholarship, I met many inspiring people and improved myself. Since the sheer list of people would consist of over forty names, I can only name a few representatives: Mara Höhner and Isabell Osterrieder, the leaders of our group, Isabella Mantel, our representative in the scholarship council, and Professor Dominik Burkard and Christoph Hartmann, our persons of trust.

Last, I want to thank my friends and supporters in my honorary occupation, the German passenger federation PRO BAHN. Special thanks go out to our honorary chairman Karl-Peter Naumann, my current and former federal co-chairs Detlef Neuß, Anja Schmotz, Andreas Schröder, Simon Bredemeier, Jörg Bruchertseifer, Stefan Barkleit, and Marcel Drews. The same goes for my current and former bavarian co-chairs Winfried Karg, Siegfried Weber, Timm Kretschmar, Jörg Lange, Jörg Schäfer, and especially Matthias Wiegner, who left us far too early. Without these astonishing people (and many others that would exceed any reasonable space when listed) it would have been impossible to carry out my positions in parallel to my Ph.D. project.

Publication List

Peer-Reviewed International Journal Articles

Florian Wamser, Thomas Zinner, Lukas Iffländer, and Phuoc Tran-Gia. "Demonstrating the Prospects of Dynamic Application-aware Networking in a Home Environment". In: *ACM SIGCOMM Computer Communication Review* 44.4 (Aug. 2014), pp. 149–150. ISSN: 0146-4833. DOI: `10.1145/2740070.2631450`

Peer-Reviewed International Conference Contributions

Full Research Papers

Thomas Prantl, Peter Ten, Lukas Ifflander, Alexandra Dmitrenko, Samuel Kounev, and Christian Krupitzer. "Evaluating the Performance of a State-of-the-Art Group-oriented Encryption Scheme for Dynamic Groups in an IoT Scenario". In: *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Nov. 2020, pp. 1–8. DOI: `10.1109/mascots50786.2020.9285948`

Norbert Schmitt, Lukas Iffländer, André Bauer, and Samuel Kounev. "Online Power Consumption Estimation for Functions in Cloud Applications". In: *Proceedings of the 16th IEEE International Conference on Autonomic Computing (ICAC)*. Umea, Sweden: IEEE, June 2019. DOI: `10.1109/icac.2019.00018`

Lukas Iffländer, Christopher Metter, Florian Wamser, Phuoc Tran-Gia, and Samuel Kounev. "Performance Assessment of Cloud Migrations from Network and Application Point of View". In: *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Mobile Networks and Management*. Ed. by Jiankun Hu, Ibrahim Khalil, Zahir Tari, and Sheng Wen. Vol. MONAMI 2017. Cham: Springer International Publishing, 2018, pp. 262–276. ISBN: 978-3-319-90775-8. DOI: `10.1007/978-3-319-90775-8_21`

Florian Wamser, Lukas Iffländer, Thomas Zinner, and Phuoc Tran-Gia. "Implementing Application-Aware Resource Allocation on a Home Gateway for the Example of YouTube". In: *Lecture Notes of the Institute for Computer Sciences,* *Social Informatics and Telecommunications Engineering. Mobile Networks and Management*. Ed. by Ramón Agüero, Thomas Zinner, Rossitza Goleva, Andreas Timm-Giel, and Phuoc Tran-Gia. Vol. MONAMI 2014. Cham: Springer International Publishing, 2015, pp. 301–312. ISBN: 978-3-319-16292-8. DOI: `10.1007/978-3-319-16292-8_22`

Vision Papers

Lukas Iffländer, Jürgen Walter, Simon Eismann, and Samuel Kounev. "The Vision of Self-aware Reordering of Security Network Function Chains". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*. ACM Press, 2018. DOI: `10.1145/3185768.3186309`

Short, Tutorial, Poster and Tool Papers

Thomas Prantl, Peter Ten, Lukas Iffländer, Stefan Herrnleben, Alexandra Dmitrenko, Samuel Kounev, and Christian Krupitzer. "Towards a Group Encryption Scheme Benchmark: A View on Centralized Schemes with focus on IoT". in: *2021 ACM/SPEC International Conference on Performance Engineering (ICPE)*. ICPE'21. Apr. 2021

Thomas Prantl, Lukas Iffländer, Stefan Herrnleben, Simon Engel, Samuel Kounev, and Christian Krupitzer. "Performance Impact Analysis of Securing MQTT Using TLS". in: *2021 ACM/SPEC International Conference on Performance Engineering (ICPE)*. ICPE'21. Apr. 2021

Lukas Iffländer, Norbert Schmitt, Andreas Knapp, and Samuel Kounev. "Heat-aware Load Balancing-Is it a Thing?" In: *Proceedings of the 11th Symposium on Software Performance 2020 (SSP'20)*. Nov. 2020

Lukas Iffländer, Nishant Rawtani, Hayreddin Ciner, Lukas Beierlieb, Klaus-Dieter Lange, and Samuel Kounev. "Architecture for a Dynamic Security Service Function Chain Reordering Framework". In: *1st IEEE International Conference on Autonomic Computing and Self-Organizing Systems - ACSOS 2020*. Aug. 2020

Lukas Beierlieb, Lukas Iffländer, Samuel Kounev, and Aleksandar Milenkoski. "Towards Testing the Performance Influence of Hypervisor Hypercall Interface Behavior". In: *Proceedings of the 10th Symposium on Software Performance 2019 (SSP'19)*. Nov. 2019

Lukas Iffländer, Jonathan Stoll, Nishant Rawtani, Veronika Lesch, Klaus-Dieter Lange, and Samuel Kounev. "Performance Oriented Dynamic Bypassing for Intrusion Detection Systems". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. Mumbai, India:

ACM, 2019, pp. 159–166. ISBN: 978-1-4503-6239-9. DOI: `10.1145/3297663.3310313`

Lukas Iffländer and Nicolas Fella. "Performance Influence of Security Function Chain Ordering". In: *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. Mumbai, India: ACM, 2019, pp. 45–46. ISBN: 978-1-4503-6286-3. DOI: `10.1145/3302541.3311965`

Christoph Hagen, Alexandra Dmitrienko, Lukas Iffländer, Michael Jobst, and Samuel Kounev. "Efficient and Effective Ransomware Detection in Databases". In: *34th Annual Computer Security Applications Conference (ACSAC)*. ACM. Dec. 2018. URL: `https://se2.informatik.uni-wuerzburg.de/publications/download/paper/1797.pdf`

Lukas Iffländer, Stefan Geißler, Jürgen Walter, Lukas Beierlieb, and Samuel Kounev. "Addressing Shortcomings of Existing DDoS Protection Software Using Software-Defined Networking". In: *Proceedings of the 9th Symposium on Software Performance 2018 (SSP'18)*. Hildesheim, Germany, Nov. 2018

Florian Wamser, Thomas Zinner, Lukas Iffländer, and Phuoc Tran-Gia. "Demonstrating the Prospects of Dynamic Application-aware Networking in a Home Environment". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM Press, 2014, pp. 149–150. ISBN: 978-1-4503-2836-4. DOI: `10.1145/2619239.2631450`

Peer-Reviewed International Workshop Contributions

Lukas Iffländer, Nishant Rawtani, Lukas Beierlieb, Nicolas Fella, Klaus-Dieter Lange, and Samuel Kounev. "Implementing Attack-aware Security Service Function Chain Reordering". In: *2020 Workshop on Self-Aware Computing - SEAC 2020*. May 2020

Lukas Beierlieb, Lukas Iffländer, Aleksandar Milenkoski, Charles F. Gonçalves, Nuno Antunes, and Samuel Kounev. "Towards Testing the Software Aging Behavior of Hypervisor Hypercall Interfaces". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Nov. 2019. URL: `https://se2.informatik.uni-wuerzburg.de/publications/download/paper/2013.pdf`

Lukas Iffländer, Alexander Dallmann, Philip-Daniel Beck, and Marianus Ifland. "PABS-a Programming Assignment Feedback System". In: *Proceedings of the secod workshop for automated grading of programming exercises (ABP)*. 2015. URL: `http://ceur-ws.org/Vol-1496/paper5.pdf`

Peer-Reviewed Book Chapters

Lukas Iffländer and Alexander Dallmann. “Der Grader PABS”. in: *Automatische Bewertung in der Programmierausbildung*. Ed. by Oliver J. Bott, Peter Fricke, Uta Priss, and Michael Striewe. Vol. 6. Digitale Medien in der Hochschullehre. ELAN e.V. and Waxmann Verlag, 2017. Chap. 15, pp. 241–254

Technical Reports

Lukas Iffländer, Alexandra Dmitrienko, Christoph Hagen, Michael Jobst, and Samuel Kounev. *Hands Off my Database: Ransomware Detection in Databases through Dynamic Analysis of Query Sequences*. Tech. rep. Universität Würzburg, July 2019. eprint: `1907.06775`. URL: `https://arxiv.org/abs/1907.06775`

Theses

Lukas Iffländer. “Performance Assessment of Service Migration Strategies”. Master Thesis. Am Hubland, Informatikgebäude, 97074 Würzburg, Germany: University of Würzburg, Jan. 2016

Lukas Iffländer and Nils Gageik. “Entwicklung und Evaluierung eines Systems zur Bestimmung der Orientierung und Position eines Objektes durch inertiale und magnetische Sensoren”. Bachelor Thesis. University of Würzburg, Dec. 2012

Contents

Chapter 1

Introduction

Introducing this doctoral thesis, we first motivate the objectives of the research and set the general context. Then, before introducing the current state-of-the-art, we proceed to present a detailed statement of problems addressed in this work and show that previous works do not sufficiently address the identified problems. Building on this, we identify specific goals and research questions. Next, we summarize the results of this research, which explores these goals. We also describe how we evaluate these contributions. In the end, we provide an outline of the thesis.

1.1 Motivation

In recent years, cybersecurity has come to the attention of the general public. The FBI's cybercrime report [Gor20] shows a steadily growing number of reports and reported losses. For the United States of America alone, the losses grew by 6217% from \$485 000 000 in 2011 to \$3 500 000 000 in 2019. Similar reports from Germany's Federal Office for Information Security [Sch19] and the British Department for Digital, Culture, Media, and Sport [Dow20] confirm this to be a global issue. This growth in the damage that cyberattacks cause correlates with their increase in strength and versatility. Therefore, cybersecurity has become a dominant issue for businesses as well as personal life.

Today, most IT services run inside cloud environments. Complex security architectures protect these environments from attacks. The inspection of the incoming traffic is very compute intensive, and the employed computing resources are not available for the regular cloud operation. In the past, security systems kept up with their adversaries through the advances in microtechnology. However, with the decline of Moore's Law [TW17]—the number of transistors doubling every two years—and especially the expectation that the performance would follow the same law, this approach is coming to an end.

On the other hand, the number of Internet of Things (IoT) devices grows at an exponential rate [Sta20]. Many of these devices lack when it comes to security features, and sophisticated update policies are rare. These deficiencies make IoT devices an easy target for cybercriminals who aim to take over the control of such devices and use them as parts of their botnets to attack online services [Ant+17]

The increasing importance of cybersecurity in combination with the loss of the ability to counter these threats by replacing the used hardware every few years creates a motivation to find ways to use the available resources more efficiently. A major inspiration to this thesis and the accompanying publications was a position paper by CSA [Mil+16] envisioning a security framework that optimizes the use of security functions by converting them into VNFs and combining them with SDN. In this thesis, we develop several of the functionalities envisioned in this whitepaper.

1.2 Problem Statement

Security functions that protect against attacks add additional complexity for network architects. Unlike "normal" network functions, security network functions do not just process packets and forward them; they add the complexity of dropping packets considered as malicious. Furthermore, putting security functions at a very high load level often creates the undesired behavior of increasing the number of false positives.

Security network functions often defend only against one or few selected attacks. No single function can defend against all possible attacks. Thus, especially for so-called SSFCs—infrastructures chaining multiple network security functions behind each other—the dropping property creates a new level of complexity. While changing the order of non-security-related network functions does not impact the performance of the functions at the back of the chain, for security functions, every packet filtered out at the beginning reduces the load on the following security functions. As described in the previously mentioned reports, recent cyberattacks regularly change their composition and the used attack vectors. Therefore, it is next to impossible for network designers to predict at design time what the attack composition will look like throughout the life time of a security architecture.

In general, we see a lack of mutual adaptation between single security functions, SSFCs, and the interconnecting network infrastructure. Single security functions often lack features like scalability and statelessness or have to process packets that—e.g., at a certain point of a connection—are irrelevant but still create resource demands. SSFCs—like non-security-related function chains—use a fixed order created at design time. Depending on the occurring attack or attack mix, malicious packets would proceed through many security functions that have to process them but do not defend against their attack type. This detour creates additional unnecessary resource demands. Network infrastructures that adapt to the occurring attacks and are aware of the behavior of the deployed security functions could alleviate this issue.

In summary, this thesis addresses two main problems: (i) Many security functions do not interact well with the surrounding infrastructure. The lack of integration leads to shortcomings in terms of performance and flexibility. (ii) SSFCs design is mainly static and SSFCs do not adapt to the surrounding circumstances and the incoming traffic mix. Besides these two main problems, we also tackle two additional issues: (iii) Cloud architectures rarely use the performance potential from short-term CPU frequency boosting. (iv) Ransomware attacks—attacks that erase or encrypt data and blackmail the user into paying for the restoration—against databases become more complex and contain a series of queries. So far, no matching signature-based security solutions exist.

1.3 State of the Art

Many works focus on characterizing the performance of security functions like IDS [Sen06; Sch+03]. Also, they deal with the mentioned behavior under overload conditions [DB11]. However, these works rarely describe approaches to alleviate the found shortcomings and focus solely on the security functions themselves without considering the surrounding network.

For Distributed Denial-of-Service (DDoS) protection, more works consider the surrounding infrastructure. For example, they use stateful SDN to monitor the traffic using simple state machines on the switches to detect DDoS attacks. Other works aim at protecting against saturation of the connection between the data plane and the control plane [Shi+13a]. Two approaches work on the protection against Transmission Control Protocol (TCP) floods. However, they either inherit the shortcoming of existing solutions [Jak+16] or provide insufficient evaluation and no code to reproduce their results [Zhe+18]. In general, these works aim at replacing existing security functions by SDN and not at augmenting them using the SDN.

Two categories of works tackle the topic of SDN and security. On the one hand, the first category deals with security for SDN. Therefore, they analyze SDN's security challenges in general [SNS16; YY15] and for specific network operating systems [YL16]. The second category, on the other hand, uses SDN to enhance security. Therefore, they propose multiple solutions to integrate IDS in SDN-enabled networks [Chi+14], create blocking and diversion rules based on alerts from security functions [Xin+14] or use SDN to replace dedicated security functions entirely [Yoo+15]. However, these works do not focus on augmenting the security functions performancewise.

Much research targets the NFV domain. Works deal with: (i) NFV frameworks [Gal+15], (ii) methods to benchmark NFV frameworks and VNFs [Chi+12], (iii) possibilities that NFV offers in combination with SDN [Lor+17], or (iv) implementations of network functions as VNFs [Bre+14]. Further works target the general concept of SFCs and dynamic service chaining [Fay+15; Ble+14]. Still, they do not consider the special properties of security functions. Works that analyze challenges explicitly state the intelligent positioning of (security) functions inside a function chain as an open problem [Lui+15].

A major inspiration for our work is the CSA's "Security Position Paper: Network Function Virtualization" [Mil+16]. The authors define multiple work packages and challenges for NFV-enabled security architectures. This work addresses multiple of their mentioned goals.

To summarize, existing works leave the intelligent placement of security functions inside SFCs and the performance-oriented cooperation between security functions and SDN open. Consequently, we require approaches and solutions to these problems.

1.4 Research Questions

Based on the previously stated problems and their unsatisfying resolution in the current state-of-the-art, we formulate multiple research questions. We define meta-research questions (MRQs) that generalize a problem and specify them in further detail with multiple concrete research questions (RQs). The numbering of the meta-research questions corresponds to the respective chapters, and the numbering of the actual research questions corresponds to the respective sections. If two or more research questions correspond to the same section, we append ascending Latin letters.

MRQ 4: How and to what extent can SDN help improve the performance of (security) network functions?

- **RQ4.1a** How can SDN-based approaches improve IDS?
- **RQ4.1b** What effects do bypassing approaches have on the performance and security of IDS?
- **RQ4.1c** How do adaptive approaches, which perform reconfigurations at runtime, compare to static approaches?

RQ4.1d How do different workload levels impact the performance and security of the SDN-based approaches?

RQ4.1e Do the SDN-based approaches change their behavior when using hardware or software switches?

RQ4.2a How can SDN-based approaches improve DPSs against TCP SYN (Synchronization) packet (SYN) flood attacks?

RQ4.2b What is necessary to make such a solution stateless and independently deployable?

RQ4.2c How does such a solution perform compared to existing solutions?

RQ4.2d To what extent can parallelization improve the performance of such a solution, and how vital is parameter-tuning?

RQ4.2e Which deployment and scaling strategies suit the solution?

MRQ 5: To what extent can we improve security systems by introducing dynamic function chain reordering?.

RQ 5.1 What components and capabilities define a Security Service Function Chaining (SSFCing) framework?

RQ 5.2a How do single security functions perform under attack load?

RQ 5.2b What is the impact of the ordering when combining different security service functions?

RQ 5.3a How to model single security functions for the reordering decision?

RQ 5.3b How to model security function chains for the reordering decision?

RQ 5.3c What strategies are suitable for determining a better order?

MRQ 6: How to design a framework for dynamic SSFC reordering?

RQ 6.1 How to structure a framework for dynamic function chain reordering?

RQ 6.3a What results does a prototype implementation provide?

RQ 6.3b Do new attack vectors, and other issues arise from dynamic function chain reordering – and if yes, how can these issues be addressed?

MRQ 7: Can a CPU boost-oriented heat-aware server load rotation improve server performance?

- **RQ7.1a** Can SDN leverage potential in short-term CPU frequency boost technologies to increase the computing performance of a system?
- **RQ7.1b** How to design an SDN-based load-balancing system with such capabilities?
- **RQ7.2** What existing solutions are suitable to implement this approach?
- **RQ7.3a** To what extent do different workloads impact the approach (e.g., low load and high load)?
- **RQ7.3b** Is it possible to extend this effect for more prolonged periods?
- **RQ7.3c** How is the effect of this solution on the power consumption?

MRQ 8: How to apply signature-based intrusion detection to multi-query database ransomware attacks?

- **RQ 8.1a** How to model multi-query database ransomware attacks?
- **RQ 8.1b** Which components does a multi-query database IDPS require and how do they interact?
- **RQ 8.2** How to integrate a prototype multi-query database IDPS into a MySQL server?
- **RQ 8.3** How does the multi-query database IDPS perform in terms of security and performance?

1.5 Contributions and Evaluation Summary

This thesis contains six primary contributions. These contributions address the previously stated research questions.

Contribution 1: Dynamic Network Intrusion Detection System Bypassing
This contribution addresses *MRQ4*'s *RQ4.1a* to *RQ4.1e*. It contains an approach to dynamically bypass IDS in the phase of a connection when the configured attacks are unlikely to occur to reduce their resource demands. To this end, we propose three approaches using SDN. Two of these approaches are dynamic, and one approach follows static SDN flows. As part of the contribution, we created a demo implementation.

We evaluate the approach regarding security and performance metrics for different deployment scenarios (using a virtual and a physical switch) as well as different workloads. We show that our approaches can increase the performance to a level that matches the performance without an IDS while not creating new security issues. At overload, the performance slightly drops but remains high. The dynamic approaches profit significantly from using a software switch instead of a hardware switch, while the static approach remains at similar levels for both configurations. We published some of these results in [Iff+19b].

Contribution 2: TCP Handshake Remote Establishment and Dynamic Rerouting using Software-defined Networking (THREADS)

In the second contribution, we address *MRQ4*'s *RQ4.2a* to *RQ4.2e*. We develop THREADS, a novel approach to defend services against TCP SYN floods. This approach uses SDN to create stateless DPS VNFs that take over the TCP handshake. After a successful handshake, the SDN-enabled network sends traffic directly to the service. This contribution eliminates the issues previous solutions have: (i) statefulness, (ii) the need for all traffic to run via the security function, or (iii) limited scaling independent of the service itself.

We validate our approach by testing a Proof-of-Concept (PoC) implementation in an evaluation environment and compare it to the widely used solutions SYNPROXY and SYN cookies. THREADS performs up to three times faster than existing solutions without parameter tuning. However, it introduces a significant delay for connections for the first data packet. We present multiple solutions to this issue. Next, we develop two parallelization strategies and evaluate them while performing parameter tuning showing another increase of additional 76%. Last, we analyze an optimal deployment and scaling strategy for THREADS concluding that it performs best under horizontal scaling with small to medium instances. We presented results for an early stage of the PoC implementation in [Iff+18a].

Contribution 3: Performance Modeling for Security Service Function Chain Orders

This contribution concerns *MRQ5*. At first, we introduce the general idea behind attack-aware security function chaining. This idea constitutes placing security functions that defend against an occurring attack early in the SSFC to have them drop malicious packets and stop them from creating resource demands at security functions later in the chain. We conclude that for this approach to work, we need a central instance, the so-called Function Chaining Controller (FCC), that takes care of the security function's ordering. This controller requires knowledge about the traffic passing through the security functions and the number detected of attacks. To this end, security function wrappers co-located with the security functions report the number of detected attacks at the specific function to the FCC. We introduced the vision of this approach in [Iff+18c] and [Iff+20a].

We conduct a performance analysis to analyze the effect of the different SSFC orders. First, we measure the performance of single security functions and show that they have different behaviors and performance characteristics under benign and malicious workloads. When combining the security functions in SSFCs of size two, we note that the performance is best when putting the function that defends against the current attack first. The performance difference between the different orders is up to two orders of magnitude. For different attacks, we show orders that contradict each other. Thus, no always optimal static order exists. We published selected results in [IF19].

Based on these experiences, we create models for single security functions and security function chains. We model the traffic categorizing it into traffic classes and map the traffic to constant resource demands per traffic unit (frame, packet, or segment) as well as dynamic demands depending on the size of the traffic unit. Every security function affects the traffic as a function depending on the traffic composition. The model for an SSFC consists of multiple security function models. Traffic that exits one function continues to the next. Thereby, it is possible to compute the total resource demand. Last, we discuss three approaches with different levels of accuracy and varying compute complexity for the decision making process based on this model. We presented the first iteration of the idea for this model in [Iff+18c].

Contribution 4: A Framework for Attack-aware Security Service Function Chain Reordering

In this contribution, we address *MRQ6*. We introduce a framework for attack-aware dynamic SSFC reordering that builds on the knowledge from the previous sections. All security functions of an SSFC reside inside an SDN-enabled network. A *security function wrapper* co-located with every security function reports attacks to the FCC. Then, the FCC computes the desired order for the security functions and executes it via the SDN controller. We presented the communication design of the framework in [Iff+20b].

We then developed a PoC implementation using a simplified decision-making algorithm (putting the function with the most registered attacks first) and a minimal SDN controller tailored to our infrastructure. We put this framework through simulated attack patterns showing that it correctly adapts to all attacks and restores the initial state afterward. This result demonstrates the desired functionality. However, we see rare issues, where a single packet can jump over a security function during reordering and propose four possible solutions to this problem. We published our experiences in [Iff+20a].

Contribution 5: Heat-aware and CPU Boost-oriented Server Load Rotation
This contribution addresses *MRQ7*. We present a solution to heat-aware load balancing, allowing us to maximize the time active CPUs spend in the state of a short-term frequency boost. Therefore, the load balancer must detect or predict the moment when a server is too hot to stay boosted and migrate the running service to another server.

We present a solution consisting of two components: (i) a monitoring component watching the states of all workers, and (ii) an SDN controller that creates flows based on those observations. Next, we develop a PoC implementation and evaluate it. We show the general functionality of the approach and analyze the impact of our solution, showing that it performs best when the whole cluster is at lower load levels. For lower load levels, it is always possible to keep the active server boosted without significant temperature increases. Performancewise we show that in high-load scenarios, we can increase performance per Watt while reducing the average temperature of the cluster. With a lower load, all performance metrics improve, the temperature levels drop, and the total power consumption decreases.

Contribution 6: Signature-based Database Ransomware Detection
This last contribution deals with *MRQ8*. We propose a new multi-component signature-based Intrusion Detection and Prevention System (IDPS) named Dynamic Identification of Malicious Query Sequences (DIMAQS) for relational databases capable of protecting against newer attacks that no longer consist of a single query but instead of query sequences. Therefore, we use a signature comprised of a Colored Petri Net to model the attack behavior.

We create a DIMAQS prototype for MySQL registering as an auditing plugin with the MySQL server enhanced by multiple triggers for the protected tables. We evaluate the results for benign and malicious query sets without any misclassifications. Next, we analyze DIMAQS's performance showing that for a benign data set and synthetic benchmarks, it has a performance impact of below 5%. We published our approach and results in [Iff+19a] and [Hag+18].

Our contributions can assist security infrastructure architects in increasing their systems' performance without adding additional computing resources. Notably, the optimizations for single security functions stand for themselves, allowing for simple implementation. Security function developers can test their functions in our evaluation environment for SSFCs and, thereby, evaluate how they interact with other security functions and if they pose a possible bottleneck. CPU designers can learn lessons from using our framework for heat-aware load balancing to make better use of short-term CPU frequency boosting, and database operators can use DIMAQS to secure their systems. Last, researchers can use our results as a reference to refine modeling and decision making strategies and expand on our SSFC reordering framework.

1.6 Thesis Outline

This thesis comprises multiple chapters. Every chapter—except the first three—addresses the research questions that match the chapter number.

After this introductory chapter (Chapter 1), we introduce relevant foundations of our research in Chapter 2. We discuss the current state-of-the-art in Chapter 3.

Next, we present approaches to increase single security function performance by using SDN in Chapter 4. This includes dynamic IDPS bypassing in Section 4.1 and THREADS in Section 4.2.

Then, we discuss the general idea of dynamic SSFC reordering, the performance impact of the order inside SSFCs, and how to model SSFCs in Chapter 5. Based on these findings, we present our SSFC reordering framework in Chapter 6.

Afterward, we discuss the possibilities of heat-aware load balancing, and present and evaluate a corresponding framework in Chapter 7. In Chapter 8, we propose, implement and evaluate DIMAQS to protect against multi-sequence database ransomware.

Last, we conclude this thesis and provide an outlook on future work in Chapter 9.

Chapter 2

Foundations of Network Security and Modern Networking

In this chapter, we introduce the foundation needed for this work. Section 2.1 introduces cybersecurity with an introduction to information security and to benchmarking metrics for security systems. In Section 2.2, we present typical attack types. Section 2.3 depicts some examples of common state-of-the-art security appliances. Next, we introduce the technologies of Software-defined Networking (SDN) (Section 2.4) and Network Function Virtualization (NFV) (Section 2.5). Section 2.6 presents the architecture of current security systems and what impact the previously introduced technologies already have. Then Section 2.7 discusses power-saving and boosting technologies, and Section 2.8 introduces two relevant modeling formalisms.

2.1 Cybersecurity

Cybersecurity (also computer security, IT security) deals with protecting computer systems or networks from external threats such as damage or theft of hardware, software, or data, and the misdirection or interruption of provided services.

2.1.1 Information Security

A part of cybersecurity is information security. Information security deals with securing data from manipulation and unauthorized access. The CIA-triad [Per15] proposes three essential concepts for information security (i) confidentiality, (ii) integrity, and (iii) availability. Furthermore, many sources add (iv) non-repudiation.

Confidentiality In information security, confidentiality "is the property, that information is not made available or disclosed to unauthorized individuals, entities, or processes. [Bec15]". Confidentiality is not just a synonym for privacy but is more one of its components. It is the implementation to ensure that data is not accessible to outsiders.

Integrity Data integrity deals with assuring and maintaining the completeness and accuracy of data not only at a certain point of time but over the whole lifetime of data [Bor05]. Thus data is unmodifiable by unauthorized or undetected means. Two approaches to integrity exist. Either a system guarantees integrity by storing data in a manner where it can not be manipulated or introduces mechanisms to detect tampering.

Availability The purpose of information systems is to provide information. Thus, the system must be able to provide the required data when it is needed. Therefore, external threats must not impair the availability of a service. "High Availability Systems" is a collective term for permanently available systems [LO09].

Non-repudiation Non-repudiation defines the obligation of a party in a transaction to fulfill its commitments. In information security, non-repudiation also states that neither party can deny having received or having sent information. [MB06]

2.1.2 Benchmarking Security Systems

Benchmarking of security systems usually focuses on their quality (hence the ability to defend a service) and their performance (hence the maximum processed quantity).

2.1.2.1 Security Metrics

When benchmarking quality, security appliances must classify packets as malicious or benign. The confusion matrix, as shown in Figure 2.1, divides the classified packets into four categories [Faw06]:

True Positive (TP) The security appliance classifies a malicious packet as malicious. This classification is the intended behavior.

False Negative (FN) The security appliance classifies a malicious packet as benign. This behavior is suboptimal and unintended. Malicious packets that were incorrectly classified can enter the system and cause damage.

False Positive (FP) The security appliance classifies a benign packet as malicious. This behavior is suboptimal and unintended. Packets that should

		Classification by Security Appliance		
		malicious	**benign**	**total**
Packet is	**malicious**	True positive **TP**	False negative **FN**	P
	benign	False positive **FP**	True negative **TN**	N
	total	P′	N′	

Figure 2.1: Confusion matrix for security appliances.

reach a service can not reach the service. Depending on whether the misclassification is permanent or not, this can lead to a permanent disruption of the service.

True Negative (TN) The security appliance classifies a benign packet as benign. This classification is the intended behavior.

Multiple metrics come in to play:

- The *false-positive rate*, as shown in Equation (2.1), is the ratio of false positives (incorrectly labeled benign packets) to the total number of benign packets. It indicates the percentage of mislabeled benign packets. The goal is to minimize this metric. The reversed metric, the true-negative rate, is rarely used.
$$FP_{rate} = \frac{FP}{N} \tag{2.1}$$
- The *true-positive rate*, as shown in Equation (2.2), gives the ratio of true positives (correctly labeled malicious packets) to the total number of malicious packets. Thus, it indicates what percentage of malicious packets are correctly labeled. The goal is to maximize this metric. The reversed metric, the false-negative rate, is rarely used.
$$TP_{rate} = \frac{TP}{P} \tag{2.2}$$

- The *precision*, as specified in Equation (2.3), gives the ratio between the number of true positives and the number of all packets classified as malicious (false positives and true positives). Since $P = TP + FP$ precision is just another description of the true-positive rate. Thus, it gives the percentage of how many classifications are correctly classified. The goal is to maximize this metric.

$$precision = \frac{TP}{TP + FP} \tag{2.3}$$

- The *recall*, as specified in Equation (2.4), gives the ratio between the number of true positives an the number of malicious packets in total (true positives and false negatives). Hence, it gives the percentage of malicious packets that are correctly classified. The goal again is to maximize the metric.

$$recall = \frac{TP}{TP + FN} \tag{2.4}$$

- The *accuracy*, as specified in Equation (2.5), gives the overall share of correctly classified packets. Thus it is specified as the ratio between the sum of true positives and true negatives (hence, correctly classified files) and the sum of all malicious and benign packets. Also, security systems should aim to maximize this metric.

$$accuracy = \frac{TP + TN}{P + N} \tag{2.5}$$

- The last standard metric is the so-called *F-measure*. Equation (2.6) shows an inverse correlation between the F-measure and both the inverse precision as well as the inverse recall. It reaches its maximum (and optimal value) of 1.00 when both recall and precision reach their maximum. The F-measure is commonly used to balance between recall and precision. In most cases, it is challenging to optimize both components. A more rigorous system with a higher recall (thus, a better detection rate of malicious attacks) often tends to misclassify benign packets. This tendency, in turn, leads to an increase in false positives and thereby a decrease of the precision metric.

$$F = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} \tag{2.6}$$

- Last, *duplicate classifications* are specific to our environment of network security. In network environments, a network component may retransmit a packet that it already sent before, e.g., due to congestion. A security appliance then encounters the packet twice. This behavior increases the number of alerts as well as the load on the security appliance but has no impact on the accuracy components.

2.1.2.2 Performance Metrics

Security appliances share their key performance metrics with most other software systems [KLK20].

Throughput defines the amount processable by the system in a time frame. Standard units are requests per second, MBit per second, packets per second.

Latency the latency gives the amount of time for a machine to wait for a response to a request. Latency can be defined for simple round trips (e.g., ping) but also for more complex computation (e.g., a request from a webserver). Latency is a common description of this metric for the network application in this work. However, in performance engineering, this is also often called the response time.

2.2 Types of Attacks

There is a seemingly infinite number of different attacks. In the following, we will describe three common types of attacks that we will also later use when evaluating our various approaches towards more performant security. First, we introduce Denial-of-Service (DoS) and Distributed Denial-of-Service (DDoS) attacks in general in Section 2.2.1 and explain the SYN Flood and Hypertext Transfer Protocol (HTTP) Flood Attacks in more detail. Afterward, in Section 2.2.2, we focus on intrusion attacks and explain them using examples.

2.2.1 Denial-of-Service Attacks and Distributed Denial-of-Service Attacks

Denial-of-Service (DoS) attacks focus on disrupting the availability of a selected service. One way is to saturate the service with so many requests that it can no longer service further benign requests. A particular form of this attack is finding requests that create an abnormally high load on the service.

In many scenarios, a single client can not saturate a server. Modern cloud infrastructure offers load balancers in combination with horizontal and vertical scaling algorithms. These features allow service providers to scale the service and handle high loads easily. Therefore, regular DoS attacks evolved to Distributed Denial-of-Service (DDoS) attacks. DDoS attacks still aim at saturating a service but no longer rely on using a single client to achieve this goal. Instead, DDoS attacks employ a multitude of hosts.

While in the beginning, this meant booking numerous servers or otherwise acquiring command over them, the spread of broadband networks and the Internet of Things (IoT) has offered new possibilities. The number of IoT devices rose to 8.3 billion in 2019, and studies expect a further rise up to 21.5 billion in 2025 [Lue20]. Many IoT devices are not part of infrastructures that feature automated patching, and therefore, the devices have open vulnerabilities. The number of known open zero-day vulnerabilities in IoT devices is continuously growing [PHS16]. These vulnerabilities make it easy for attackers to take over unpatched devices. With the spread of broadband technologies, these devices are often connected with fast uplinks and can use this bandwidth for malicious requests. Recently, botnets like the Mirai botnet comprise up to 600,000 devices [Kol+17a]. With the continuing growth regarding device numbers, future botnets might likely be even more massive.

In the following, we will describe two different attacks. The SYN flood attack targets Transmission Control Protocol (TCP)'s connection establishment process while the HTTP flood aims at flooding a webserver.

2.2.1.1 SYN Flood

The SYN flood attack is currently the most common DDoS attack [KBG19]. It attacks TCP's connection establishment process. To understand this attack, we first give a course overview of TCP and the three-way handshake used for connection establishment. Then we describe the SYN flood by specifying its attacker and threat model.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP), as specified in [Pos83a], is a transport layer protocol (Layer 4 in the Open Systems Interconnection (OSI) model) and represents one of the main protocols in the Internet protocol stack. At first, TCP was part of the monolithic Transmission Control Program that later split into TCP and the Internet Protocol (IP). Hence, many sources refer to TCP as TCP/IP.

One of the main characteristics of TCP relevant for this work is its connection establishment mechanism in the form of a three-way handshake. Furthermore,

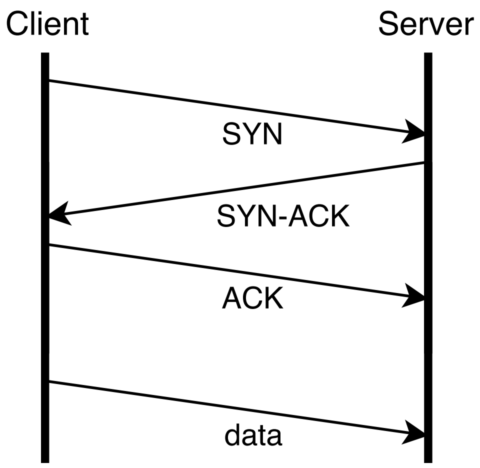


Figure 2.2: TCP three-way handshake process.

TCP ensures reliability, the in-order transmission of information, and error checking.

The TCP Three-Way Handshake:

TCP is connection-oriented. Thus, it requires establishing a connection before transmitting any payload data. Therefore, TCP performs a three-way handshake, as depicted in Figure 2.2. To request a new connection, the client sends a SYN (for synchronize) packet to the server. In this packet, the client encodes a randomly generated sequence number A. TCP uses these sequence numbers to ensure the correct order of data. Next, the server answers with a TCP SYN+ACK (Synchronization and Acknowledgment) Packet (SYN+ACK) (for synchronize & acknowledged) packet. The server encodes inside the acknowledgment number (i) the next number the server is expecting to receive of $A+1$, and (ii) its own randomly generated sequence number B. Finally, the client replies with an TCP ACK (Acknowledgment) packet (ACK) (acknowledged) packet with a sequence number of $A+1$ and an acknowledgment number of $B+1$. The completion of these steps concludes the establishment of a new connection. Further mechanisms provided by TCP, like flow control and congestion control, are omitted here due to not being relevant for the mechanism proposed in this work.

Attacker Model

First, the attacker model describes the circumstances under which it is possible to launch a SYN flooding DDoS attack. In general, an attacker operates using remote connections and does hence not require physical access to the targeted machine. The attacker's goal in this scenario is to disrupt the service and render it unavailable to legitimate requests, resulting in a denial of service. A secondary goal is mitigating the detection of a second attack, called the smokescreen. However, we will omit further details regarding this technique in this work due to its being out of scope. In general, an attacker requires little knowledge about the attacked service. The data required to launch such an attack encompasses only the port the service is listening on, and the public IP address of the attacked service. The IP address is easily obtainable through legitimate ways, such as domain resolution. The port is either publicly known or discovered through port scanning. An attacker has usually obtained control over several machines, e.g., through known vulnerabilities, and can use multiple endpoints to amplify his attack.

Threat Model

Next, we cover the threat model of a typical SYN flood attack. This type of attack exploits the connection based behavior of the TCP protocol by creating a multitude of semi-established connections, rendering the target host unable to accept any new, benign connections. Figure 2.3 illustrates this behavior of an attacker that sends SYN packets, which the server answers with SYN+ACK responses. The final part of the Three-Way-Handshake, the ACK, is omitted by the attacker. This omission leaves the connection in a valid but unfinished state.

This attack exploits the fact that the TCP protocol stores currently opened connections in so-called Transmission Control Blocks (TCBs). The TCB backlog has a limited capacity and stores not only fully established connections, but all connections initiated by at least a SYN packet. This property allows an attacker to send SYN packets with arbitrary source addresses, thus hiding his identity and avoiding connection limits. Eventually, an attacker can fill the TCB backlog with malicious, half-established connections, making the server drop new SYN requests instead of creating new connections. Thus, the server is unavailable for new, benign connections, and a denial of service occurs.

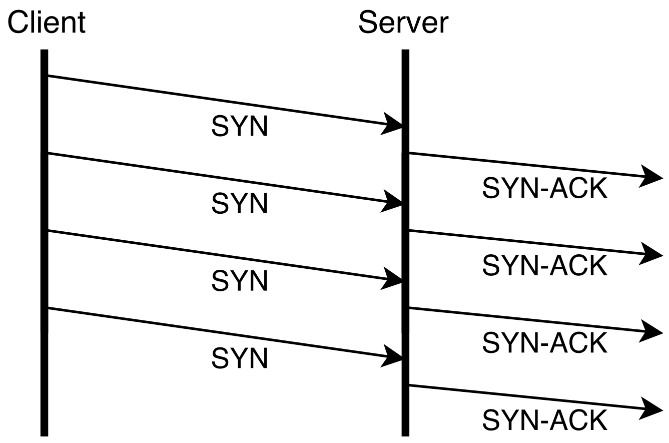


Figure 2.3: SYN flood attack pattern.

Mitigation

Another entity has to take over the connection establishment process to mitigate SYN flood attacks. Section 2.3.2 describes two conventional approaches: SYN cookies, and SYNPROXY.

2.2.1.2 HTTP Floods

The HTTP flood attack works on the application layer - Layer 7 in the OSI model. The attack aims at overwhelming a targeted service with HTTP requests. It is relatively complex to defend against application-layer attacks. The difficulty arises from the challenge of distinguishing benign requests from malicious ones.

Attacker Model

The attacker model shares similarities with the SYN flood attack. Again, the attacker operates remotely and does not require physical access to the target machine. The attackers share the goal of achieving a denial of service or mitigating the detection of smokescreen attacks. Different from before, the attacker requires more knowledge about the attacked service. He needs to know how to structure HTTP requests to the service. Thus, the attack requires knowledge of the arguments for website access or parameters of Representational State Transfer (REST) interfaces. The other data required to launch such an attack is relatively simple to obtain. As described for the SYN flood attack, the attacker can quickly obtain the IP address and port of the service. The attacker is, again, usually in control of a botnet or a similar infrastructure.

Threat Model

The attacker sends as many HTTP requests as possible to the attacked service. For this purpose, the attacker employs his botnet. There are two types of HTTP floods:

HTTP GET attack: For this attack, the botnet sends multiple requests to redeem images, files, or other data provided by the targeted server. Once saturating the target's maximum throughput, the attacker thereby forces the target to drop additional requests from legitimate traffic sources. Thereby, denial-of-service occurs.

HTTP POST attack: HTTP POST requests allow the client to submit data, e.g., for forms on websites. In general, servers have to perform more complex operations when handling POST requests. The server must handle the request and then usually issues an operation to make the data persistent. To this end, the server stores the data to a database or some sort of file storage. These operations are relatively intensive. Thus the attacker can generate a high amount of load with relatively little bandwidth. Thus, the denial-of-service can occur with a smaller amount of employed attacking resources. For this attack type, the attacker requires even more knowledge about the attacked services.

Mitigation

Mitigating application layer attacks is rather complicated. It is not simple to decide which requests are benign and which ones are malicious. One method is to implement a challenge to the requesting machine to test for bots, e.g., a captcha if automated connections are entirely undesirable. For automated connections, challenges like JavaScript computational tasks for the attacker slow down the attack frequency.

For HTTP floods, the use of a Web Application Firewall (WAF) is another alternative. By combining firewalls as described in Section 2.3.3 with a managed IP address reputation database, security systems can selectively block malicious traffic. Existing approaches to said reputation databases are only semi-automated and still require constant analysis by engineers. Large providers can offer WAF services due to their experience and data gathered from hosting thousands and millions of sites.

2.2.2 Intrusion

While the previous attacks rely on having control over a botnet or a similar infrastructure, intrusions do not require this ability. However, the attacker could benefit from the ability to launch a parallel DDoS attack as a smokescreen. Instead, single packets or requests trigger a vulnerability. This approach is similar to DoS attacks that trigger high loads with single or few malicious requests. Nevertheless, the goal of a DoS attack is to put the service out of operation while intrusion attacks usually have different goals, e.g., triggering a remote code execution.

Attacker Model

Again, intrusions like the described DDoS attacks are remote attacks. Thus, the attacker does not need physical access. The attacker must be able to ensure that his malicious packets reach the attacked service. Furthermore, the attacker must know what service he is attacking. The HTTP flood only needs to know that a web server is running at a particular IP address and listens on a specified port. In comparison, for an intrusion attack, it is also necessary to gather information about the webserver (e.g., Nginx or Apache) and its version. When all this information is available, the attacker also requires knowledge about an existing vulnerability for the webserver in its specific version.

There are multiple ways to obtain such vulnerabilities. For older versions, Common Vulnerabilities and Exposuress (CVEs) provide vulnerabilities published by the software vendors. While CVEs aim at motivating people into updating by showing which vulnerabilities no longer occur in newer versions, they also allow attackers to obtain attack vectors when someone decides not to upgrade. The attacker could also look for vulnerabilities themselves, e.g., by fuzzing the source code of the webserver. Last, a black market exists on the Internet, providing vulnerabilities in exchange for payment.

Threat Model

With his knowledge, the attacker sends a request or a series of requests to the service. These packets then trigger an effect that is not desired by the service provider.

An example is an attack against the Apache web server in combination with the Oracle WebLogic Apache Connector. An HTTP POST request with the content "xxxxxxxxxxxxxxxxxxxx" could trigger a buffer overflow that can lead to either a denial-of-service or a privilege escalation allowing to access features that should be protected.

Another example - again against the Apache web server - is a cross-site scripting attack. By calling the JavaScript code `alert(document.cookie)`, the attacker tries to execute scripts above his access rights.

In general, most attacks on web servers focus on cross-site scripting, denial-of-service, and privilege escalations. For other services like, e.g., mail servers or file servers, other vulnerabilities exist, but intrusions against them follow a similar pattern.

Mitigation

The obvious way to mitigate intrusion attacks is to patch the software, so the vulnerability is eliminated. However, in some scenarios, patching, unfortunately, is not an option. Either a patch might not yet exist, or the application has to undergo critical homologation processes after every patching.

When patching the service is not feasible, it is necessary to prevent harmful packages from reaching the service. Intrusion Detection Systems (IDS) can perform this task. We will introduce IDS in Section 2.3.1 and also present signatures to defend against the two attacks mentioned before.

2.3 State-of-the-art Security Appliances

A variety of security appliances protect against the attacks described in Section 2.2 and many others. State-of-the-art security systems comprise a great selection of these appliances in various combinations. The following pages introduce Intrusion Detection Systems (IDS) (Section 2.3.1), DDoS protection systems against the SYN flood attack (Section 2.3.2) and firewalls (Section 2.3.3).

2.3.1 Intrusion Detection and Prevention System

Intrusion Detection and Prevention Systems (*IDPS*) combine IDS and Intrusion Prevention Systems (IPS). IDS can detect attacks [SM07], and many IDS provide additional defense mechanisms. IPS are capable of actively defending against incoming attacks, and many IPS are deployable in a detection-only mode.

2.3.1.1 Types and Categories

IDS can detect attacks originating from outside the protected system as well as local attacks. Therefore, the different monitoring platforms allow classifying IDS into *network-based*, **host-based**, and **hybrid** IDS.

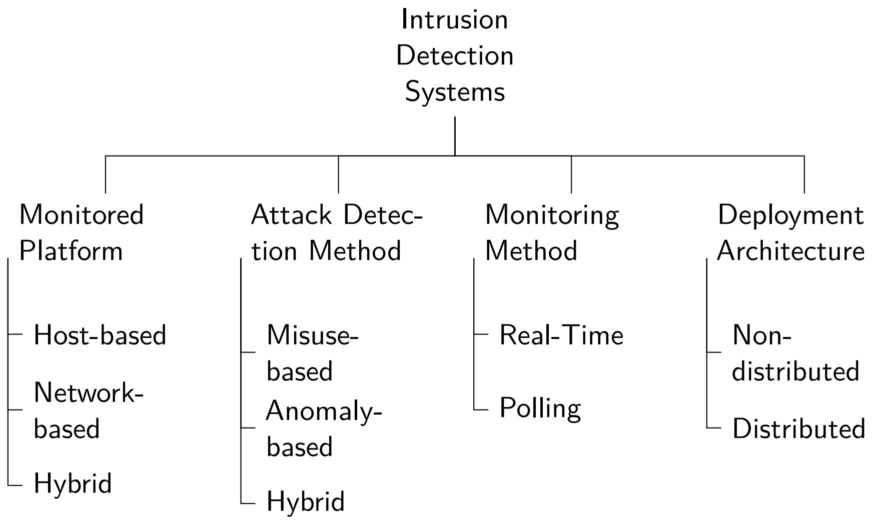


Figure 2.4: Categorization of intrusion detection systems.

The next distinction relies on the used attack detection methods. *Misuse-based* approaches primarily target singular attacks that are usually carried out in a single step [VRB04], exploiting a selected vulnerability. Here, an IDS uses signatures containing features of an attack for its detection. *Anomaly-based* solutions use training techniques to detect when a system diverges from its normal behavior, which allows detecting multi-phase attacks. These are attacks where the type and order of the executed actions lead to the exposure of the system. The single stages can be seemingly friendly activities of which combination leads to a security violation (e.g., zero-day exploits).

Real-time or event-based IDS intercept an activity before it reaches the target system inspecting it synchronously to the traffic flow. *Polling* IDS do not interrupt the traffic flow but analyze the target activity periodically and are asynchronous. A primary application for polling IDS is host-based intrusion detection [GR03]. True to their asynchronous nature, polling IDS are not suitable for intrusion prevention.

Non-distributed IDS are deployable at a singular (central) position inside the system. An alternative is to use a *distributed* IDS that is spread all over the system allowing detection of interconnected attacks on multiple non-collocated targets. Figure 2.4 presents an overview of the different types of IDS.

In this work, we focus on network-based, misused-based, non-distributed, and real-time IDS.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET $HTTP_PORTS
↪  (msg:"SERVER-APACHE Oracle WebLogic Apache Connector buffer
↪  overflow attempt"; flow:to_server,established;
↪  content:"POST"; http_method; content:"xxxxxxxxxxxxxxxxxxxx";
↪  depth:100; metadata:policy max-detect-ips drop, service
↪  http; reference:bugtraq,30273; reference:cve,2008-3257;
↪  reference:url,www.oracle.com/technology/deploy/security/⌋
↪  alerts/alert_cve2008-3257.html; classtype:attempted-admin;
↪  sid:18283; rev:6;)
```

Listing 2.1: Signature for the Buffer Overflow Intrusion Attack on Oracle WebLogic Apache Connector from Section 2.2.2

2.3.1.2 Signatures

In the context of IDS, a signature contains the characteristics of attacks against the system. To enable its detection, an IDS needs to conduct an in-depth analysis of the observed behavior via Deep Package Inspection (DPI). For real-time IDS, a high-performing DPI engine is necessary to handle the unpacking of network traffic down to lower protocol levels. Like anti-virus research, many security experts and enterprises perform ongoing worldwide analysis of attacks and attack methods. This constant analysis leads to the regular publication of new signatures for various operating systems, protocols (e.g., TCP, HTTP, and File Transfer Protocol (FTP)) and applications (e.g., Xen, and Apache webserver). Aside from public signature databases, there are closed signature databases for a selected community as well as commercial and exclusive signatures collections from security researchers and corporations.

For the example attacks in Section 2.2.2, signatures exist. Listing 2.1 shows a signature to defend against the buffer overflow intrusion attack. The rule includes that the request must be a POST request and that the content must contain "`xxxxxxxxxxxxxxxxxxxx`." The signature in Listing 2.2 matches when "`alert(document.cookie)`" is included. Both signatures trigger an IPS to drop the packet. Additionally, the signatures contain information like the CVE number, a classification, and a website with further information[1].

[1]Unfortunately with the frequent redesigns of websites, these often lead to 404 pages.

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
↪  (msg:"SERVER-APACHE Apache SSI error page cross-site
↪  scripting attempt"; flow:to_server,established;
↪  content:"alert(document.cookie)"; fast_pattern:only;
↪  http_header; metadata:policy max-detect-ips drop, service
↪  http; reference:bugtraq,32476; reference:bugtraq,5847;
↪  reference:cve,2002-0840; reference:cve,2008-5278;
↪  reference:url,packetstormsecurity.com/files/cve/⌋
↪  CVE-2002-0840; classtype:web-application-attack; sid:11687;
↪  rev:21;)
```

Listing 2.2: Signature for the Cross-site Scripting attack on the Apache Web Server from Section 2.2.2

2.3.2 SYN Flood Protection

In the following, we describe two of the most popular mechanisms to mitigate DDoS attacks, SYN Cookies as well as SYNPROXY. Being readily available for services running on top of the mainline Linux Kernel makes both of these solutions widely used.

2.3.2.1 SYN Cookies

As mentioned during the description of the threat model, a SYN flooding attack exploits the size limitation of the TCP backlog as a critical resource for establishing new connections. SYN cookies [Edd07] are a fully TCP standard-compliant way of eliminating the need for backlog entries related to half-open connections. In a usual scenario, the backlog stores source and destination addresses as well as ports, the client's Initial Sequence Number (ISN), the server's ISN, and the requested TCP options for half-open connections. Storing these values of half-open connections is necessary to check if a received ACK packet belongs to previous SYN and SYN+ACK packets and whether the client correctly received the server's ISN. The idea of SYN cookies is to store this information not locally, but encode it into the sent SYN+ACK packet and retrieve the information from the ACK response.

Figure 2.5 shows the structure of the TCP header with the source and destination port values determined by the connection parameters. The acknowledgment number must be the client's sequence number submitted with the SYN

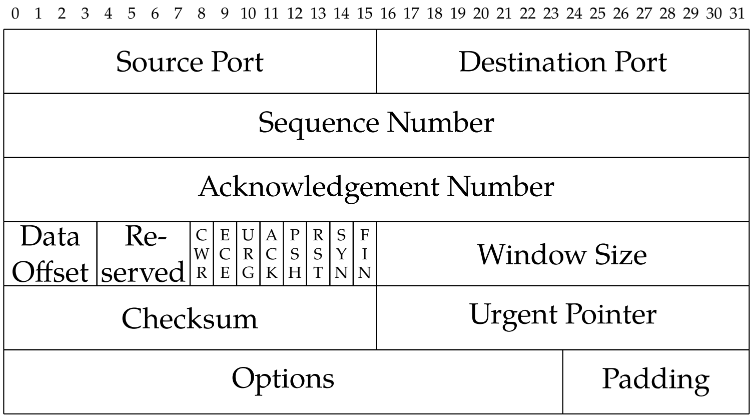


Figure 2.5: TCP header.

Bits	Content
31 to 27	Time counter
26 to 24	Client MSS size
23 to 0	Hash value of connection properties

Table 2.1: Sequence number composition using SYN cookies.

packet incremented by one. The data offset has to describe the size of the header, and SYN and ACK flags must be enabled. While the window size can contain arbitrary values, it has a significant impact on the connection throughput. Thus, it should assume a typical value. The remaining header fields contain checksum calculations. The urgent pointer field can contain arbitrary values, and its interpretation only occurs for segments with an enabled URG flag.

On the one hand, this allows encoding 16 bit of information into it. On the other hand, this is of no use because the client ignores the data and does not transmit it back in his ACK packet. The same is true for the padding at the end of the header. Additionally, it is supposed to consist of zeros. These limitations leave the sequence number and possibly the TCP options as potential information storage.

In practice, SYN cookies use the 32-bit sequence number to store the required data. There are no regulations for the choice of the ISN except that it should increase over time. For security concerns, it also should not be predictable. Table 2.1 shows the sequence number's composition when using SYN cookies.

The time counter is used to fulfill the increasing requirement and is calculated by $unix\_time \gg 6 \pmod{32}$, resulting in a 5-bit number increasing every 64 seconds. MSS stands for Maximum Segment Size [Pos83b] and amounts to 536 bytes by default. The Maximum Segment Size (MSS) option allows the choice of differing values. The client uses the options in the SYN packet to tell the server the maximum size of TCP segments it wishes to receive. The server replies with his MSS choice in the SYN+ACK response and usually saves the client's value in the newly created TCB. Because SYN cookies aim at avoiding local memory consumption, this solution stores the MSS efficiently in the ISN by choosing eight MSS values together with a 3-bit encoding beforehand. The chosen MSS is the largest of these values that is still smaller than the client's choice.

The previously discussed 8 bits are predictable, so the leftover 24 bits have to ensure that the sequence number is not inferable by a third party. At the same time, the server must be able to verify whether the packet is a valid response to a SYN+ACK packet or if it is spoofed using the acknowledge number of an ACK packet. Therefore, the server hashes the client's and server's IP addresses and ports, as well as the time counter and a secret number to avoid predictability. When an ACK packet is received, the server calculates the hashes for the last few values of the time counter. If one of them matches, the server creates a TCB extracting all the necessary values (addresses, ports, client sequence number, client MSS) from the ACK packet. This final step concludes the connection establishment.

For this mechanism to work, the SYN cookies functionality must run on the same machine as the protected service. Due to this limitation, SYN cookies is not scalable independently from the service itself. In cases in which the service does not require additional resources, but the SYN cookies functionality does, this behavior adds additional overhead to the system. For some services not designed for scaling, this would generate significant overhead regarding deployment management.

2.3.2.2 SYNPROXY

The second solution to mitigate TCP SYN floods presented is SYNPROXY, a Netfilter module [Bro19]. Netfilter is "a set of hooks inside the Linux kernel that allows kernel modules to register callback functions with the network stack" [WA19]. When a packet passes a hook, it triggers the execution of all registered modules. SYNPROXY utilizes this functionality to prevent SYN packets from directly reaching the networking stack, where they would trigger TCB allocation and a SYN+ACK response. Instead, the module drops the SYN packet. It sends a manually crafted SYN+ACK packet as if the SYN was legitimately handled by the network stack, thereby masking the connection establishment from the kernel.

Similarly to the previously presented SYN Cookies, SYNPROXY stores the connection state in the packet itself instead of local memory. On arrival of an ACK packet, SYNPROXY checks if the packet belongs to a valid handshake. If this is the case, from the client perspective, the connection establishment seems to have successfully concluded. The server, on the other hand, does not yet know that a connection was requested. Therefore, the SYNPROXY module has to impersonate the client and execute a handshake with the server. Thereby, it opens the connection in the client's stead.

While both ends now know about the connection, they cannot directly communicate with each other, because the ISN chosen by SYNPROXY when impersonating the server is only in 1 out of 2^{32} cases equal to the one the server chooses in the second handshake. To fix this, SYNPROXY has to modify the server's sequence and the client's acknowledge numbers in every subsequent packet.

The synsanity [git18] module reduces the overhead of intercepting and modifying every packet by matching the ISN choices of the module and the network stack. Since it cannot influence the ISN the kernel will use, the only way is to predict which number the network stack would use. Synsanity exactly copies the kernel function and uses the same secret used for calculating SYN Cookie sequence numbers to reach this goal. However, synsanity has become deprecated in the meanwhile.

SYNPROXY allows for either co-deployment with the protected application or deployment on a separate machine. When opting for co-deployment, SYNPROXY suffers from the same shortcomings as SYN cookies, namely the inability to scale the mechanism independently of the service. Additionally, unlike SYN cookies, the application is stateful. This property complicates many processes like, e.g., service migration. When deploying SYNPROXY on a separate machine, all traffic between the client and the server must pass through the proxy. This detour creates an additional single point of failure and additional traffic inside the network that could result in new network bottlenecks.

2.3.3 Firewalls

Oppliger [Opp97] defines firewalls as "intermediate system[s] [...] plugged between the network and the Internet to establish a controlled link, and to erect an outer security wall or perimeter. The aim of this perimeter is to protect the network from network-based threats and attacks, and to provide a single choke point where security and audit can be imposed."

The use of network firewalls lies in the idea of moving security away from a single host and the running applications to dedicated and more manageable entities. Such network firewalls can work on multiple protocol layers.

Most common firewalls work on the third layer of the OSI-Stack, also known as the network layer. Since the main action on this layer is routing, an older name for Layer 3 firewalls is screening routers. These routers filter the incoming packets by a set of rules. Whether a packet matches against these rules or not defines whether it is forwarded. The rules rely on information available in the packet headers. Thus, they include protocol numbers, source and destination IP addresses, connection flags, and other IP options. These features are open to extensions to similar parameters on the fourth layer, the protocol layer. These extensions would then include, e.g., port numbers, flags, or TCP option parameters.

Another type of firewall comprises the so-called proxy servers. These servers require authentication before the individual services can be accessed. If the authentication is successful, the proxy forwards packets between the server and the client. The SYNPROXY solution shown in Section 2.3.2.2 also matches these criteria (hence, the name).

2.4 Software-Defined Networking

Software-defined Networking (SDN) takes on the challenges posed by the increasing number of participants in networks and the associated exponential increase in cost due to the directly correlated growth in resource demands. The objective during development was to achieve greater scalability, flexibility, automation, and independence from hardware manufacturers to reduce capital as well as operational expenditure.

2.4.1 General Approach

Five principles are fundamental to SDN:

The *separation of control and data planes* divides the switching process into the control plane, using routing algorithms to decide on packet forwarding and the data plane technically handling the packet. In conventional switches, this task is performed by an embedded, performance-optimized chip, which generally cannot be directly influenced and therefore leads to the exchange of the entire device for the implementation of new algorithms. SDN allows influencing the forwarding process from the outside via a software interface to communicate with the switch changing its behavior at runtime without having to replace the

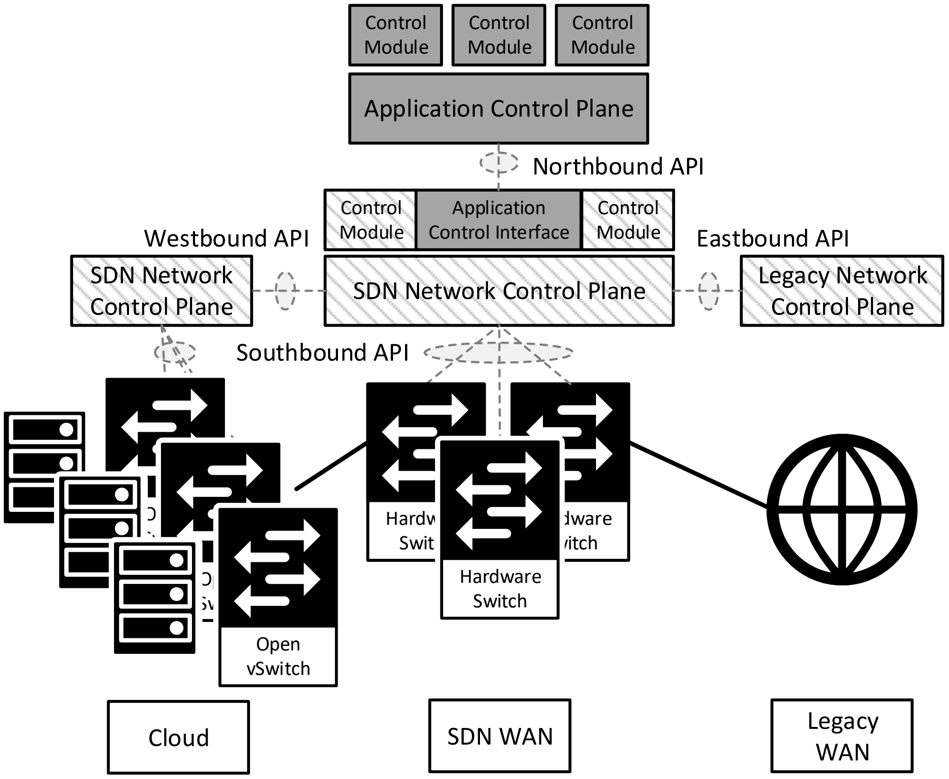


Figure 2.6: Overview of SDN structure and the corresponding Application Programming Interfaces (APIs) [Jar+14].

hardware components. Today, the OpenFlow (OF) protocol often serves as an interface [McK+08], as described below.

The *central control instance*, also called controller, enables the configuration and administration of the network. For reasons of availability or load distribution, it allows for deployment as a physical or virtual replica in the network.

Programmability allows changing the behavior of a switch using software, enabling the installation from algorithms or other applications from different manufacturers - independent of the hardware producer. This feature also allows the applications to operate above the network layer, at the application level, regardless of the switch model or operating system.

Additionally, *protocol independence* allows running different network protocols such as the Ethernet protocol or the TCP/IP stack. In general, it is possible to use any standardized network protocol.

Open interfaces are a prerequisite for vendor independence. This necessity is especially relevant since the communication between the control and data plane requires open protocols to connect different switch models with manufacturer-independent controllers.

SDN brings together many areas that are handled separately in traditional networks via various APIs. There are four essential APIs featuring many ways of implementation as depicted in Figure 2.6 [Jar+14; Ope16a]:

- The *Southbound API* connects the control and data plane.
- The *Westbound API* allows communication between different control instances of different domains.
- The *Northbound API* is responsible for exchanging information between the applications and the control plane.
- The *Eastbound API* provides a contact surface for non-SDN components, such as the Network Management System (NMS) or other mechanisms used in legacy systems.

However, the concept of softwarized networks is not limited to OF and the APIs, as mentioned above. Recently, solutions like P4 [Bos+14] or Domino extend the concept of SDN into the control plane by allowing to define the feature set provided by hardware components programmatically.

2.4.2 OpenFlow

The most prominent protocol used in this context is OpenFlow (OF) [Ope16a]. The usage of a standardized interface between the control and data plane alleviates vendor lock-in. It allows the integration of heterogeneous physical hardware into networks built using the SDN paradigm. OF routes and switches network packets following so-called flows. Flow tables on the controller and the network devices determine the routing of said flows.

In version 1.3, released in 2012, the following aspects among others are specified [Ope12]:

- *Feature Request:* A network device provides information about its capabilities (e.g., the number and properties of ports or the queue behavior).
- *Flow Tables Configuration:* The editing of the flow tables allows for the deletion and creation of new flows and their characteristics (e.g., time limits).

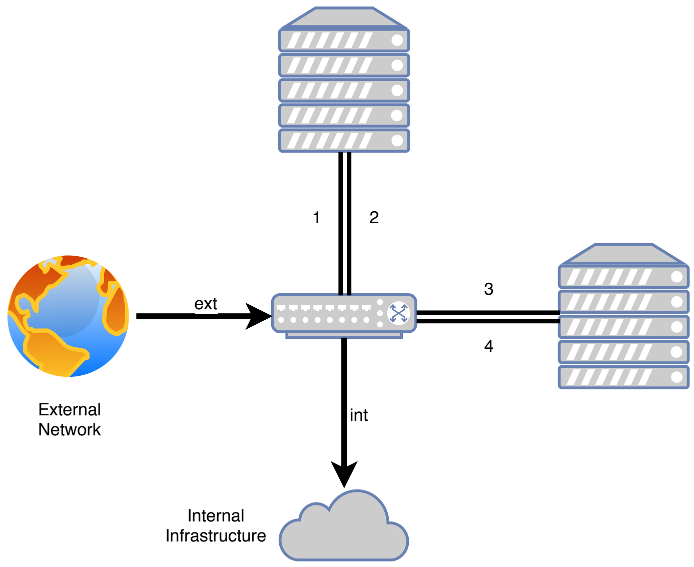


Figure 2.7: A small network with two servers and a switch.

Name	Match	Priority	Hard Timeout	Actions
StdFlow1	Port-In: ext.	100	-	Port-Out: 1
StdFlow2	Port-In: 2	100	-	Port-Out: 3
StdFlow3	Port-In: 4	100	-	Port-Out: int.

Table 2.2: Flow table for network depicted in Figure 2.7

- *Port Status:* OF can permanently change the behavior of specific ports (physical ports on the switch).
- *Packet In:* If no specific behavior for a situation is defined, a new packet reaching a network device in the control layer triggers this event.
- *Packet Out*: These messages allow the control layer to send a response with the determined instruction for the network device.

The protocol, now available in version 1.5.1 [Ope15], has been extended to include the possibilities of using (and coordinating) several controllers, and the capability of configuring the use of different prioritized flow tables and queues in the switch [Ope16b].

Flow tables contain *flows* that apply actions to packets that match specific rules. Actions can be the forwarding to a port of the switch, but also modifying OSI

Layer 2 and Layer 3 properties of packets. In Figure 2.7, two servers connect to an OpenFlow switch. That switch then itself also establishes the connection between the internal infrastructure and the external network. Table 2.2 shows the rules on the switch. Packets coming from the external port match with *StdFlow1* and directly continue to port 1 of the switch. Packets coming from port 2 match rule *StdFlow2* proceed to port 3 and *StdFlow3* forwards packets coming from port 4 to the internal network. This use case is just a simple example of how OF can manipulate traffic routing. Matching rules can be more sophisticated, as it is possible to match with IPv4 / IPv6 addresses and MAC addresses, source and destination addresses for each, and the EtherType of the packets. Table 2.3 illustrates matching fields and combinations from [Flo20]. More match fields are available at [Ext20], respectively.

Ingress Port	Src. MAC	Dst. MAC	Ether type	Src.IP/ Snd.IP	Dst.IP/ Tgt.IP
ANY	ANY	ANY	NO	NO	NO
ANY	ANY	ANY	8100	NO	NO
ANY	NO	NO	806	Any Snd.IP	Any Tgt.IP
ANY	ANY	ANY	800	Any Src.IP	Any Dst.IP
ANY	NO	NO	800	Any Src.IP	Any Dst.IP
ANY	ANY	ANY	800	Any Src.IP	Any Dst.IP
ANY	ANY	ANY	800	Any Src.IP	Any Dst.IP

Table 2.3: Match combinations in OF 1.3 [Ext20]

2.5 Network-Function Virtualization

2.5.1 General Approach

Network Function Virtualization (NFV) is a new paradigm for networks. Typically deployed on proprietary specialized hardware in the past, these functions are replaceable by software solutions running on commodity hardware [Chi+12]. Typical examples of such functions are switching, routing, load balancing, and firewalls Network Address Translation (NAT) [Tay14].

The implementation of a function is usually referenced as Virtualized Network Function (VNF), as it is commonly deployed inside a Virtual Machine (VM) to allow for higher flexibility and scalability.

Not every function is suitable for conversion into a VNF. The use of optimized processors or Field-Programmable Gate Arrays (FPGAs) can still be advantageous due to real-time requirements or the need for many resources at high network speeds [Han+15].

VNFs depend on performance in several ways. First, the network adapters limit the number and speed of available ports. This limitation complicates replacing larger switches with VNFs. Second, the I/O subsystem between the network card and the application can affect the performance. Therefore, many NFV solutions focus on this issue. Third, the resources provided for the application (e.g., main memory, CPU cache) can become a bottleneck. NFV shares this challenge with common compute applications [NSV16].

Many NFV solutions are usually implemented in conjunction with specialized operating systems or drivers to minimize bottlenecks. Mapping the network functions in software separates the data and control layers. This separation is one of the central goals of SDN. Although both NFV and SDN share this separation, both paradigms still can be distinguished and viewed independently of each other. In principle, NFV is also achievable without separation into data and control layers. However, in combination, SDN can help to simplify the use of NFV by improving the availability, integration, and overall performance of a system.

2.5.2 Challenges

While NFV offers new flexibility, it also poses several challenges. [Haw+14] names the seven areas of (i) security, (ii) computing performance, (iii) VNF interconnection, (iv) portability, (v) operation and management, (vi) co-existence with legacy networks, and (vii) Carrier-Grad Service Assurance.

2.5.2.1 Security

NFV environments require *security* at a level close to existing propriety network functions. Security is divided into four functional domains to achieve this goal:

(i) Virtualization environment domain (e.g., hypervisors)

(ii) Computing domain

(iii) Infrastructure domain (e.g., networking)

(iv) Application domain

The virtualization environment itself can offer several vulnerabilities. For example, the hypercall handlers show several attack vectors [Mil+14]. Many works [GAV18; Mil+15] use the hypercall interfaces themselves to inject attacks or perform robustness testing campaigns. Furthermore, Hypervisors are long-running software. Therefore, they are prone to software aging and related bugs [BS14; Mac+12; Mat+12; PR18]. These bugs can lead to security issues. [Haw+14] proposes isolating the served virtual-machine space by adding authentication controls to secure access.

The computing domain concerns itself with the actual execution of the VNF's code on the physical hardware. Usually, a VNF is not running exclusively on a machine or Central Processing Unit (CPU) but instead shares its resources (e.g., CPU, memory, storage) with other virtualized appliances. While [Haw+14] proposes secured threads, memory erasure before reallocation, and encrypted data storage, this only ensures a limited level of security. Attacks like Spectre [Koc+19], Meltdown [Lip+18], Foreshadow [Bul+18; Bul+19], and Foreshadow-NG [Wei+19] showed that widely accepted presumptions on hardware security require reevaluation. Existing mitigation approaches are known to result in severe performance losses [Pro+18].

In the infrastructure domain, a similar issue arises. Multiple VNFs share the same logical-networking layer as well as the same Network Interface Card (NIC). However, this domain is less challenging than the two previous ones. Using secure protocols (e.g., Transport Layer Security (TLS), Secure Shell (SSH)) eliminates these concerns. When used, they encrypt all data on the NIC and the lower layers. Those protocols are extensively tested, available on most platforms, and current hardware often has dedicated acceleration components.

The application domain security is no different for VNFs than for classical devices. The application code implementation must be secure. However, additional security features of modern operating systems can even provide an advantage for NFV applications.

2.5.2.2 Computing Performance

When replacing specialized hardware with VNFs running on commodity hardware, this can impact the computing performance and the resulting network metrics (e.g., the packet throughput at a switch). Since increased flexibility alone has limited usefulness, an NFV solution must be able to match the performance of the replaced system, at least. [Haw+14] proposes massive multi-threading for the deployed applications as well as horizontal scaling over multiple hosts.

Further, the authors advocate independent memory structures to avoid Operating System (OS) deadlocks and the implementation of processor affinity techniques (e.g., Streaming Single Instruction Multiple Input (SIMD) Extensions (SSE) and Advanced Vector Extensions (AVX)). Additionally, they suggest for VNFs to implement their network stack themselves and to have direct access to input/output interfaces. Various solutions emerged to engage these issues. Section 2.5.3 introduces two wide-spread solutions, the unikernel OS ClickOS and the Data Plane Development Kit (DPDK).

2.5.2.3 VNF Interconnection

The classical approach is limited to directly connecting two network functions or going through Layer 2 (L2) switches. In an NFV environment, there are different ways to connect the VNFs:

(i) Both VNFs run on the same host and are on the same Local Area Network (LAN). Thereby, the machines connect to the same virtualized switch.

(ii) The VNFs run on the same server but are on different LANs. Packets pass through the first virtualized switch to the host's NIC, then to an external router or switch and back through the host's NIC. They finally reach their destination via the virtualized switch of the second LAN.

(iii) Two different servers host the VNFs. Packets from one machine take a path via a virtualized switch on the first host to its NIC and then to an external router or switch. From there, they pass to the second host, enter it via its NIC, and reach the second VNF via another virtualized switch.

These different modes, on the one hand, allow for additional flexibility and even host part of an NFV infrastructure in the cloud and part on-premise. On the other hand, this complicates network management and adds additional waypoints for the packets. Therefore, [Haw+14] suggests evaluating the applicability of Single-Root I/O Virtualization (SR-IOV). Some NICs provide this feature, allowing connecting VMs to the NIC directly, skipping the virtualized switch. These interfaces then have higher performance then interfaces provided by the hypervisor. On the other hand, the hypervisor interfaces allow for more straightforward configuration and VM migration. The NFV workload and architecture define which approach is best.

2.5.2.4 Portability

Multiple approaches exist to deploy VNFs [Haw+14]. Each way features advantages and drawbacks.

Deploying directly on bare-metal hardware ensures a predictable mapping of software instances to hardware and, thereby, predictable performance. The drawback is sacrificing resource isolation and, thereby, security. Also, the used software would depend on a particular OS or hardware.

Deploying the VNFs through a virtualized environment can improve their portability by ensuring that the VNF has a uniform view on the hardware resources. This deployment allows for each VNF to run in its preferred operating system. The VNFs are unaware of the underlying operating system and hardware. Furthermore, this approach ensures the resource isolation required for some of the previous criteria.

The selection of the deployment approach depends on the application scenario and architecture. The more the scenario requires the flexibility added by NFV, the more deployment in a virtualized environment becomes preferable.

2.5.2.5 Operation and Management

Flexible usage of NFV's capabilities requires a powerful orchestration management system. Similar to SDN, this system interacts via northbound and southbound interfaces [Haw+14]. Northbound interactions allow managing and accessing the VNFs. Additionally, VNFs could use them to query for information or ask for additional computing resources. Southbound interactions provide communication with the underlying infrastructure.

For powerful yet straightforward management, the system should allow interacting with VNFs in the form of simple drag-and-drop operations. The systems require the description of both the VNFs as well as the underlying infrastructure using standard templates to facilitate this management.

2.5.2.6 Co-Existence with Legacy Networks

Networks rarely transition from full-legacy to full-NFV architectures in a single step. Therefore, NFV solutions must be able to co-exist with legacy components. Thus, they should be able to interact with the legacy management systems with minimal unintended impact on the existing infrastructure. The network forwarding graph should stay unaffected by the existence of the VNFs. Lastly, the transition between legacy network functions and VNFs must ensure safeguards against service interruptions and performance impacts [TK13].

2.5.2.7 Carrier-Grade Service Assurance

Carrier-grade services give certain guarantees for their hardware, software, and other components to ensure reliability and availability. NFV services must provide the same guarantees to replace legacy systems. Therefore, they must ensure resistance to failure, service continuity, and service assurance. Mechanisms that automatically reconstitute VNFs after failure events provide resilience to failure. Redundantly deploying VNFs ensures service continuity. Finally, the management system must monitor the network function performance and scale up by deploying additional VNFs. This step provides service assurance [Tal+15].

2.5.3 Abstraction Layers

As mentioned before, performance is a significant challenge for NFV solutions. This statement is especially true for some compute heavy-applications like DPI. Here a specific challenge arises. When processing packets, this can either be done in the kernel space or the userspace. Using the kernel space results on the one hand in a higher performance but, on the other hand, limits the capabilities to the features provided by kernel-level modules like IPTables and NfTables [WA19]. Running the application in user space adds additional overhead. When packets come in via the NIC, the data is not directly available to userspace applications. Thus, the system must copy the data to memory accessible from userspace. Then the userspace application can work on the data. If the application modifies the data, this triggers another copying process back into kernel space before the NIC can transmit the modified data.

Since both approaches have their drawbacks, this triggered the development of abstraction layers to allow for fast interaction between kernel space and userspace. This section introduces two approaches to this issue; the ClickOS Operating System and the Data Plane Development Kit (DPDK).

2.5.3.1 ClickOS

ClickOS [Mar+14] is a Xen-based unikernel platform optimized for middlebox processing. Therefore, ClickOS overhauls Xen's Input/Output (I/O) subsystem. These subsystems include the back-end switch, virtual net devices, back-, and front-end drivers. ClickOS is capable of booting a new VM in a few milliseconds and only adds minimal overhead delay to network transmissions.

ClickOS aims at parallelly running a multitude of VNFs on a single host and still achieve an impressive throughput. Figure 2.8 shows one hundred virtualized switches running on a single machine. Still, they are capable of saturating a 10 GBit/s port. For all packet size except for the smallest, two 20 GBit/s ports can still be satisfied.

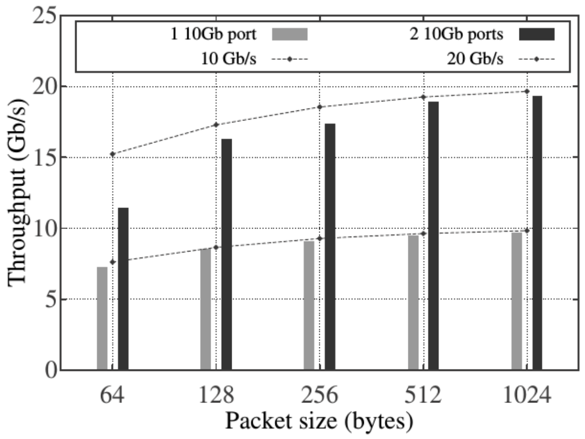


Figure 2.8: ClickOS throughput running 100 instances on one core [Mar+14].

Increasing the number of parallel instances has little effect on performance. Figure 2.9 shows the effect of scaling from 20 instances on one host to 100 instances with little impact. This scalability allows implementing complex function chains.

2.5.3.2 Data Plane Development Kit (DPDK)

The Data Plane Development Kit (DPDK) (formerly also known as Intel DPDK) is a collection of libraries to accelerate packet processing workloads running on a wide variety of CPU architectures [DPD20].

The original ETSI NFV white paper [Chi+12] declared DPDK to be a key enabler technology for the new paradigm. DPDK offers libraries for queue management, network packet buffer management, and memory management. These allow fast communication between the applications and the underlying hardware.

Experimentation results confirm DPDK's performance claim. [Kou+15] shows that DPDK can outperform the classical LibPCAC by a factor of up to ten. On a system with an Intel Xeon E5-2560 v3, LibPCAP shows saturation behavior at around 1 GBit/s when performing DPI while DPDK can saturate a 10 GBit/s. This behavior is reproducible when adding SR-IOV, which only degrades DPDK's performance by 19%.

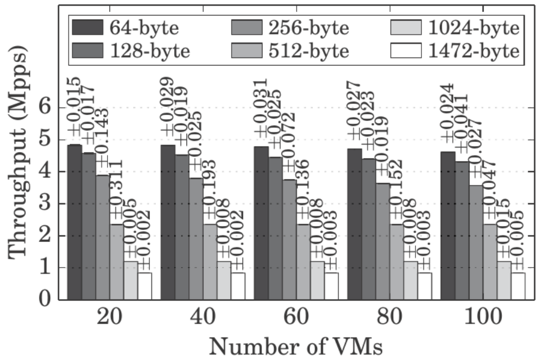


Figure 2.9: ClickOS throughput with varying number of instances [Mar+14].

2.6 Architecture of Current Security Systems

Security architectures evolved in multiple steps. Nevertheless, it still is relatively static.

At first, services directly connected to the Internet without upstream security appliances. Service designers did not assume a malicious attempt since, at first, the Internet provided only connections between research facilities.

In the next evolution with the Internet going public, security appliances appeared between the external network and the services. These appliances were (and in many cases still are) mainly dedicated devices with specialized hardware. Thus, they were usually installed in racks and then interconnected with the network. Employing multiple security appliances was realized by connecting these devices in a fixed order. Network architects derived this order by either following white papers from security vendors like Cisco or HPE or by performing an educated guess. Only in rare cases, approaches to performance modeling came into play.

The next step was to integrate multiple services into the security infrastructure. Figure 2.10 shows an infrastructure with multiple services. The firewall symbol represents security appliances where different colors determine different appliance types. Some security appliances are usable by multiple services (e.g., the first appliances in the depicted chain). The decision of how many services a single appliance can handle depends on the performance properties of the appliance. When designing an architecture, architects derive these values from datasheets. When extending existing architectures, the number of services per appliance often grows in a trial-and-error pattern until the security appliance becomes a bottleneck. In this case, it is possible to augment

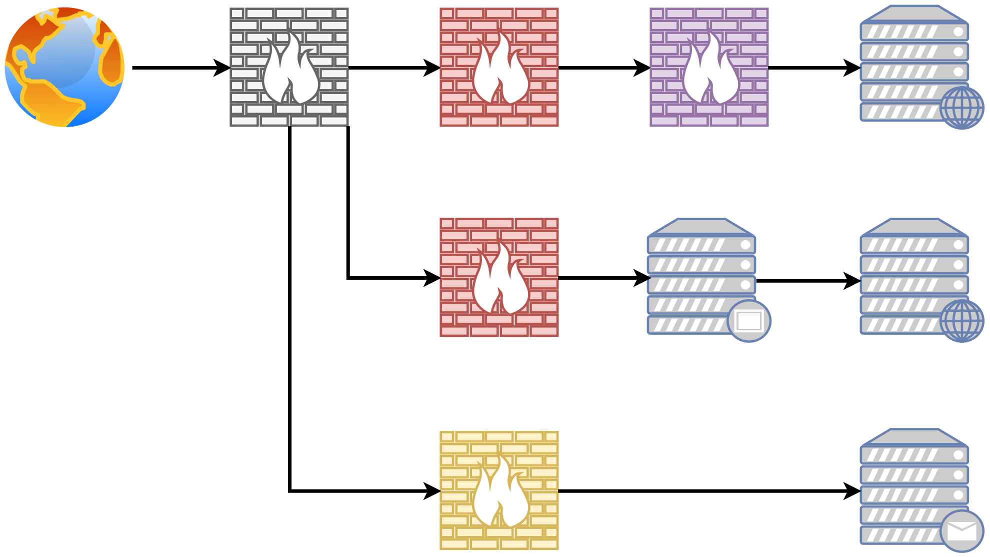


Figure 2.10: Sample classical security infrastructure.

The firewall symbols represent security appliances. Different colors indicate different appliances.

the throughput of the exhausted appliance type by adding further instances (still in many cases by adding a second physical device in a rack). Figure 2.10 shows this approach with the second appliance for the top and middle service (both web servers). In many cases, services can not share all elements of the security infrastructure (as depicted for the third security appliance for the top web server, the proxy server for the central web server, and the second security appliance for the mail server).

We call these chains of security appliances (or security functions) Security Service Function Chains (SSFCs) [Cho+16]. An Service Function Chain (SFC) defines the number and order of security appliances in the chain.

The previously described new paradigms also made their impact on security systems. With the introduction of SDN, security appliances no longer must have a direct interconnection. Instead, SDN-enabled networks can reroute traffic to any security appliance instance, including remote instances in other compute-centers or even in the cloud following the idea of Security as a Service (SECaaS).

Combined with the concurrent introduction of NFV, security appliances no longer need to be specialized hardware components, but instead, generic computing hardware can provide these capabilities. With the virtualization of resources, this allows quickly scaling security architectures to meet rising resource demands by adding additional instances when needed. Since most attack patterns are unpredictable, reactive autoscalers allow adapting when attacks occur.

Still, the architectures do not yet adapt the order of the used security appliances. Only in some cases, DDoS Protection System (DPS) enablement depends on whether a service exceeds a specified load level. These on-demand security appliances are placed in the same position in the SFC every time.

2.7 Power-saving and Boosting Technologies

Nowadays, a multitude of hardware manufacturers exists. Chips are not only used in desktop computers or laptops but also smartphones and tablets. Particularly for mobile devices, lower power consumption is quite essential, but users also will not accept a reduction in performance. Therefore, some manufacturers came up with various technologies for increasing the clock speed while keeping a reasonable energy consumption. An example of mobile phones ist is *Huawei*, with its *"Graphics Processing Unit (GPU) Turbo"*-technology [Hua19], for example. They promise up to 60% more performance and 35% less power consumption. However, in this work, we will focus on technologies for desktop and server CPUs.

In this section, we first introduce the technologies from the two leading x86 manufacturers: Intel's Turbo Boost and AMD's Turbo Core and Precision Boost technologies. Then, we discuss thermal, and power management approaches since they correlate with the limits of the aforementioned boosting technologies. We use the presented technologies and knowledge in Chapter 7 for an approach to increase performance and reduce the power consumption of servers.

2.7.1 Intel Turbo Boost

The first version of *Intel® Turbo Boost Technology* was introduced in 2008 and followed by version 2.0 in 2011. Compared to classical overclocking, the CPU is still working in the specifications of its *Thermal Design Power (TDP)* — the amount of heat a chip is specified to admit and the cooling system should dissipate — but can work for a short period with a higher clock speed than usual. The technology is independent of the OS and enabled by default. It can only be disabled in the BIOS and only CPU-based, not core-based [Int18]. Between version 1.0 and version 2.0 of the boost technology, the principle of operation did not change a lot, but it was improved. Figure 2.11 shows how the boost works in general.

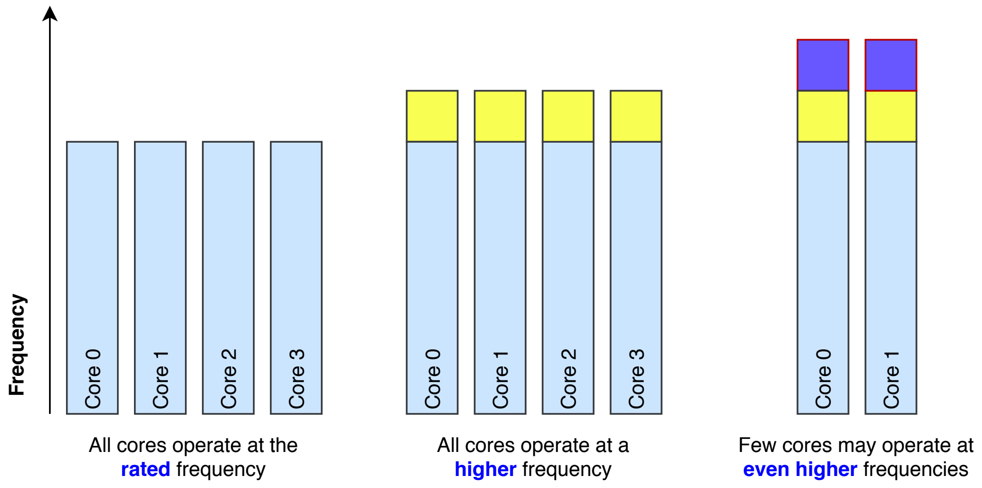

Frequency
Core 0
Core 1
Core 2
Core 3
All cores operate at the **rated** frequency
Core 0
Core 1
Core 2
Core 3
All cores operate at a **higher** frequency
Core 0
Core 1
Few cores may operate at **even higher** frequencies

Figure 2.11: Visualization of turbo boost [Cas09]

If the OS requests extra performance, the CPU activates the boost by itself and delivers higher frequencies on its cores. Since the headroom is dynamic, the maximum frequency depends on the number of boosted cores. The lower the number of boosted cores is, the higher the clock speed of the remaining cores can be. For a specific workload, the heat and energy consumption limit the time for a CPU in boost-state [Cas09]. If an OS now requests more calculation power, the CPU can increase its clock speed by 133.33 MHz steps, depending on the number of active cores. For example, if only one core needs boost, there can be two steps at the same time. There are four different so-called *C-States* to see whether a core is active or not: *C0*, *C1*, *C3*, and *C6*. *C0* and *C1* describe active cores while *C3* and *C6* count as inactive for the *Turbo Boost Technology*. The mentioned *C-State* definition counts for *Nehalem*-based CPUs only [Int08].

2.7.2 AMD Turbo Core and Precision Boost

Similar to Intel´s Turbo Boost technology, *Advanced Micro Devices* (*AMD*) also supports automated overclocking of its CPUs. This technology is called *Turbo Core*. The name outlines the main difference between Intel's and AMD's technologies. While Intel Turbo boost is capable of boosting all cores at the same time, Turbo Core can only boost some cores [SR18]. While Turbo Boost increases the clock rate inside a range of up to five steps of 133.33 MHz [Ern19], Turbo Core has only one fixed step of 400 MHz or 500 MHz, depending on the CPU model [Wal19]. In contrast to Turbo Boost, where some of the cores are "electrically off" [Ern19], Turbo Core just puts the non-boosted cores into a deep sleep state. For example, the AMD Phenom II X′ can boost the 400 MHz only if three of the six cores are in a deep sleep state.

The Turbo Core technology has its advantages but rarely becomes active in real-world environments. Often, just fewer cores are running. AMD soon developed a new and better boost technology as a part of their new Zen product line to alleviate these shortcomings. They call this new technology *Precision Boost*. Version 1 of Precision Boost still retained the limitation, that only a limited number of cores could be in a boosted state. However, Precision Boost improved in the way that each core can boost independently in varying degrees. Table 2.4 shows an example AMD provides for their AMD Ryzen™ 5 1600 processors [Hal19].

State	Clock Rate
Base Clock	3.2 GHz
1–2 boosted cores	3.6 GHz
3–6 boosted cores	3,4 GHz

Table 2.4: Boost states for AMD Ryzen™ 5 1600 processors [Hal19].

Precision Boost's second version considerably improves upon this individual boosting. One can imagine a triangle with the corners labeled as "max clock," "heat," and "power" to understand this process. Now let there be a smaller triangle within that does not intersect any of the corners. With Precision Boost version 2, the CPU can now boost and increase the inner triangle until it intersects with one of the corners of the outer one [AMD18]. Figure 2.12 shows a simplified representation of *Precision Boost* version 1 and version 2.

2.7.3 Thermal and Power Management

Thermal and power management of CPUs and GPUs are becoming increasingly relevant. The benefits of effective power management range from extending battery life, for example, in portable devices or to lower energy consumption in computing centers. As modern processors integrate and include more and more different functional blocks, like CPUs and GPU, the budgeting of heat and power is becoming increasingly crucial. As today's processors combine and include more and more different functional blocks like CPU and GPU, it is getting more and more important to budget heat and power. Traditional

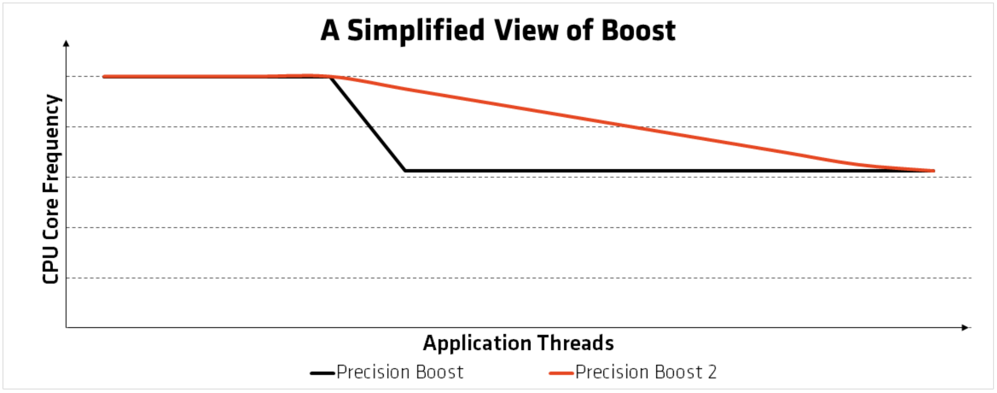


Figure 2.12: Simplified visualization of precision boost [AMD18].

power and heat sink mechanisms involved shutting off or downclocking of these functional blocks when they were not required since heat is a by-product of using electricity. However, this saved heat and power are available to other blocks. Even so, it is very challenging to evaluate the exact use of power, as different processes involve a varying amount of power and heat production. For example, an I/O operation requires less energy than a floating-point operation due to the required transistor logic [Adv14]. The term Thermal Design Power (TDP) describes the maximum heat and power with which a chip can deal. As described above, it is possible to save power by down-clocking or deep-sleeping unused cores. When considering the resulting heat, there are different strategies to deal with it once it appears. One option is to consume less energy and therefore produce less heat, but that puts impracticable limits on the processing power. Still, many useful techniques can deal with the heat. Two types of thermal management exist "active" and "passive" [FZ14].

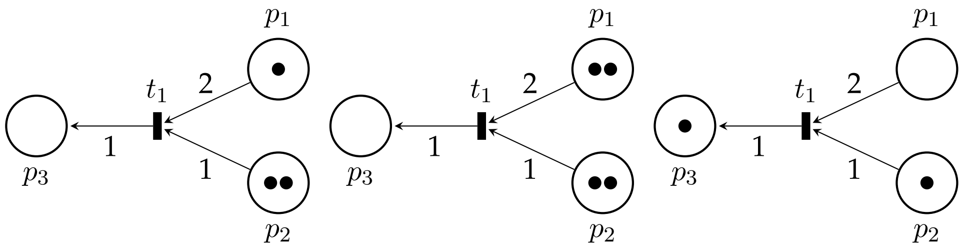
Passive strategies include increasing the conductivity of the surrounding environment. This approach happens to be rather simple and operates at a lower power cost without any difficulties [FZ14]. For example, mobile phones, tablets, or thin laptops use passive cooling systems. So-called heat pipes are a typical example. A heat pipe is a heat transfer device that combines the principles of both thermal conductivity and phase transition to transfer heat between two solid interfaces effectively.

Active techniques are more expensive and mostly more extensive. Nevertheless, they provide better thermal conduction [FZ14]. Typical examples are the fans in desktop computers or servers. Often, heat-management solutions both approaches. For example, many laptops use heat pipes to transport heat from various components to a cooling block that that an active fan cools itself.

2.8 Modeling Formalisms

Modeling behavior signatures requires modeling formalisms. We use Petri Nets (PNs) and Colored Petri Nets (CPNs) to model attack signatures in Chapter 8.

2.8.1 Petri Nets



(**a**) Transition disabled (**b**) Transition enabled (**c**) Transition fired

Figure 2.13: Demonstration of Petri net execution using a simple example.

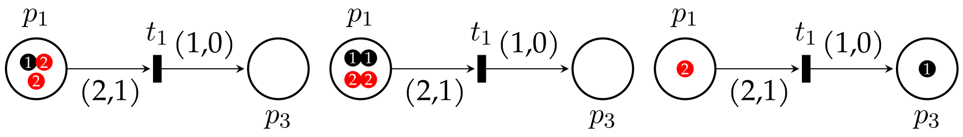
Petri Nets (PNs) are a commonly used mathematical modeling language for the description of distributed systems [Pet81] named after their inventor Carl Adam Petri. They are a class of discrete event dynamic systems offering an exact mathematical definition of their execution semantics and an extensive mathematical theory above other modeling standards like UML activity diagrams. A PN is a directed bipartite graph, in which nodes represent *places* and *transitions*, while edges, called *arcs*, connect either a place to a transition or a transition to a place, but never connect two places or two transitions directly. Transitions are events in the system, and places are conditions that need to be satisfied for the transition to fire. Hence, PNs are also often called place/transition nets.

Places may contain a discrete number of marks called *tokens*. Transitions *fire* if they are *enabled*, which is achievable by placing enough input tokens on the input places — i.e., places directly connected to the transition. The value of the arc defines the number of tokens required per place. Once a transition fires, it consumes the required number of input tokens from the input places. The transition results in creating the specified number of output tokens on the places with arcs from the transition to them (output places).

Figure 2.13 shows a simple example of a PN. Circles represent places, bars are transitions, arrows are arcs, and dots are tokens. The depicted PN consists of three places, one transition, and three arcs. Enabling the transition requires three tokens: Two tokens at place p_1 and one token at place p_2. In Figure 2.13a only one token is available at p_1. Regardless of the total count being three tokens, with only one token on p_1 the transition is not yet enabled. Adding another token to p_1 in Figure 2.13b satisfies the requirement and thus enables the transition. When the transition fires, two tokens are subtracted from the token set at p_1 as well as one token from p_2. At the same time, the transition adds one token to p_3. Figure 2.13c shows the state after the transition firing.

PNs are a powerful tool for modeling [Che+05] and allow for extensions to suit various tasks like queuing PN for performance modeling. In this work, we use Colored Petri Nets, an extension to ordinary PN.

2.8.2 Colored Petri Nets



(**a**) Transition disabled (**b**) Transition enabled (**c**) Transition fired

Figure 2.14: Colored Petri Net example. In comparison to the regular Petri net depicted in Figure 2.13, the number of required *places* is reduced from two to one without reducing functionality.

Colored Petri Nets (CPNs) enable support for tokens of different types, also known as *token colors*. Places can now contain tokens of multiple colors. Arcs can define any combination of the colors for the number of input and output tokens. This addition allows for making Petri nets more compact.

Figure 2.14 illustrates the reduction in representation complexity by presenting a CPN derived from the previous example. The places p_1 and p_2 depicted in Figure 2.13 are now merged into a single place denoted as p_1, while tokens are now assigned different colors: Tokens formerly placed in p_1 are now black (1) and those placed in p_2 are red (2). The transition now requires two black and one red token instead of requiring two tokens from p_1 and one from p_2. The overall Figure 2.14 depicts the same process as before. In Figure 2.14a, one black token is missing for the transition to be enabled. In Figure 2.14b, this token is added, thus enabling the transition. Finally, in Figure 2.14c the transition has fired, subtracting two black and one red token from p_1 and adding a black token to p_3.

Chapter 3
Related Work

Related work comprises multiple categories. First, we introduce works regarding IDS performance, and DDoS protection systems sharing commonalities with our approach. Afterward, we analyze works on SDN security regarding security for SDN as well as SDN applications for security systems. Then, we present papers dealing with various NFV topics as well as Security Function Chaining (SFCing). Last, we take a look at an article on NFV security that has inspired large parts of this work.

3.1 Intrusion Detection System Performance

Related work in the area of IDS performance mostly deals with either the measurement of IDS performance or the inference of essential factors on the performance and optimized signatures.

Sen, in her report "Performance Characterization and Improvement of Snort as an IDS" [Sen06], discusses the performance characteristics of Snort as an IDS and proposes ways to improve its performance by introducing new data structures. After a description of Snort's software architecture, she performed a series of experiments to investigate the effect of the changes on Snort's performance. Like the first experiment, the dependence of packet sizes on bandwidth was studied by passing packets of different sizes (46, 64, 152, 1000, and 1452 bytes) through Snort one at a time. The results of this experiment showed that the bandwidth of Snort increased as packet size increased, thereby indicating a link between the two. However, it showed that although the packet size has a dependence on the efficiency of Snort, the number of network packets had a more significant influence on the throughput. A second experiment examined the impact of the number of signatures on the throughput. It revealed that as the number of rules supported increased, the throughput decreased. Finally, the performance of various pattern-matching algorithms, usable for signatures in Snort, was analyzed. The correlation between the number of packets and packet size on the bandwidth of Snort was a critical aspect that can be taken from this paper to evaluate a system for Snort's performance.

Schaelicke et al. also investigated the performance characteristics of Network Intrusion Detection Systems (NIDS) in their work in [Sch+03] As with [Sen06] they generated packets of varying sizes (64, 512, 1000, and 1452 bytes) and tested them against a varying number of rules, performing all experiments on six different hardware configurations. In their experiments, they also differentiated between the header and body signatures, which examined only the protocol header or the user data, respectively. The tests showed that with larger packets, more signature rules could be present before observing a significant packet loss by Snort.

Moreover, from their experiments, it was seen that microprocessor performance was not the only criterion for Snort's performance. Also, the header signatures exhibited a high processing load, limiting the packets per second. On the other hand, payload rules scaled with an increase in packet size.

Meng et al., are working in their report "EFM: Enhancing the performance of signature-based network intrusion detection systems using enhanced filter mechanism" [MLK14] towards improving NIDS by developing an Enhanced Filter Mechanism mitigating issues such as network packet overload, expensive signature matching, and high false alarms experienced in large-scale networks. The proposed solution consists of three major components, namely (i) a context-aware blacklist-based packet filter, (ii) an exclusive signature matching element, and (iii) a KNN-based false alarm filter. The evaluation of the solution used three experiments: (i) using a light-weight DARPA dataset, (ii) using a real data set collected from a Honeypot, and (iii) in a working network environment. This paper's "Context-Aware Blacklisting" that worked on blacklisting IP addresses with the help of a look-up table, and the "Exclusive signature matching component" that quickly identified a mismatch increased the signature matching process, thereby increasing NIDS performance. The final results looked promising.

Other publications examined NIDS for their behavior under overload conditions. Alhomouda et al. [Alh+11] compared the Suricata and the Snort NIDS. This paper focused on determining the packet loss at different network speeds for the two NIDS. Various trials evaluated the performance of TCP and User Datagram Protocol (UDP) applications, with the rate of packet loss being between 0% and 60%. The testbed consisted of three guest operating systems, namely Linux 2.6, virtual Linux, and FreeBSD, to compare different operating system conditions. The different chosen operating systems showed different characteristics for Suricata and Snort concerning the packet loss behaviors. Therefore, the authors concluded that the choice of NIDS and the Operating System used should be dependent on the type of traffic and the targeted network speed required by the network.

Day and Burns [DB11], in their work, also examined Suricata and Snort in an overloaded condition. In contrast to the publication by Alhomouda et al., this paper analyzed accuracy, dropped packet rate, system utilization, and offline speed. This publication identified the metrics that were responsible for the characterization of the performance under different situations and which system resources played a role in influencing these characteristics. These metrics included packets per second, bytes per second, protocol mix, the number of unique hosts, the number of new connections per second, the number of concurrent connections per second, and the alarms to be triggered per second. To classify these metrics for their performance effects, selected host resources needed monitoring, namely CPU utilization, memory utilization, network interface throughput, storage, and page file metrics for both Snort and Suricata. Additionally, for each NIDS, the number of packets dropped, false negatives, true negatives, true positives, and the total amount of alarms were also relevant. Both NIDS clearly showed lower accuracy with higher load.

The work of Tjhai et al. confirmed the issue of the high false-negative rate of Snort [Tjh+08]. The authors were motivated to investigate the false positive detections more precisely. Under the division into "true alarm rate" and "false alarm rate," they analyzed different frequently used signature sets to their false positive rate. The paper concluded that the high false-positive rate was one of the future challenges for the development of NIDS. One of the metrics that illustrated this work accurately was the true and false positive rate.

The work of Chahal and Nagpal in [CN16] continued with the problem of false-positive subordinate rule sets and developed a concept of the generalization of signatures. Similar signatures were combined to reduce the number of signatures to, in turn, reduce false alerts. Less and more primitive signatures decreased the false positives slightly, but the effect was not sufficient to significantly reduce the error rate.

While all of these works dealt with either improving or evaluating the performance of IDS, none of the works applied SDN to reduce the load, increase the throughput, and reduce the false-positive rate of existing SDN solutions as we do in Section 4.1.

3.2 Distributed Denial-of-Service Attack Protection

Many solutions against DDoS attacks exist. In the following, we present a selection of solutions relying on NFV or SDN.

Boite et al. present their solution StateSec in [Boi+17]. Statesec is a stateful monitoring solution for DPS in SDN-enabled networks. Their primary motivation is that the conventional approach of polling statistics from SDN switches is resource-expensive. Thus, they move responsibilities from the SDN controller to the switches and use entropy to detect DDoS attacks and portscans. For their approach, they use stateful SDN as described in [Bia+14]. This approach allows using a simple finite state machine on the switches for a distributed detection of DDoS attacks, Internet Control Message Protocol (ICMP) floods, and portscans. This finite state machine performs traffic monitoring and pushes the gathered information to the controller. The anomaly detection is performed partly on the switches and the controller. Last, the controller decides whether to take mitigation actions. The authors evaluate their approach against OF and sFlow in a simulated company-wide network. They show that they can detect attacks all attacks for regular attacks and for harder to detect slow DDoS attacks, 80% of all attacks, and 85% of all attackers.

Efforts such as FRESCO [Shi+13a] and AvantGuard [Shi+13b] aim at providing frameworks for the rapid design of security mechanisms. AVANT-GUARD targets the challenge of attacks able to saturate the connection between data and control plane. The paper proposes a modified data plane, such that it proxies TCP handshakes and only informs the control plane of finished handshakes. The implementation runs inside the OF switches. Unfortunately, the authors do not give much detail on that information, so that we can only assume that the required capabilities are limiting the use of most switch models or might even force the use of software switches. As mentioned in Section 4.2.3 for the TCP SYN (Synchronization) packet (SYN) flood, the paper presents an evaluation at 1000 pps that hardly allows comparison with our approach.

VFence [Jak+16] is a scalable agent-based approach to detecting and mitigating DDoS attacks in softwarized networks using TCP connection establishment spoofing, similarly to our approach. However, VFence, similar to SYNPROXY, has two limitations: (i) the traffic always has to traverse the VNFs, and (ii) the VNFs are stateful.

Finally, [Zhe+18] provides a mechanism that uses SDN switches to collect aggregated information and report to a centralized controller. Due to its centralized view of the network, the controller can then detect and react to occurring DDoS attacks. This approach, on the one hand, puts additional load on the controller, and, on the other hand, the controller can not detect certain types of

DDoS attacks due to only receiving aggregated data provided by the switches. The authors state that they are successfully able to detect SYN flood attacks and throttle their flows. Unfortunately, they omitted detailed results due to page limitations.

3.3 Software-defined Networking and Security

SDN creates what Qiao Yan and F. Richard Yu [YY15] refer to as “a very fascinating dilemma.” On the one hand, it offers a wide range of benefits for security implementation and management. On the other hand, it suffers from multiple vulnerabilities that make it a target for attacks. Therefore, related work in SDN security comprises two central topics. First, analyzing the security impact of SDN concepts and technologies and second, implementing (existing) security solutions inside SDN infrastructures.

3.3.1 Security for Software-defined Networking

SDN introduces new security challenges. We first present general concerns and then focus on DoS attacks on SDN.

3.3.1.1 General Concerns

SDN inherits several security challenges from traditional networks. Moreover, it brings with it new issues as it implements new sets of components, interfaces, and techniques. The research demonstrated that attack could target each layer and interface of the SDN stack. Different attacks can affect not only the functioning of the targeted component but also the availability and confidentiality of the whole network.

Scott-Hayward et al. [SNS16] presented a detailed analysis of the security challenges of SDN. They categorized them by type and concerning the SDN layer/interface affected by each issue/attack. Table 3.1 lists the seven main categories of issues and provides several specific examples of the way how those issues can turn into vulnerabilities. In [YL16], more specific experimentation examines two of the most prevalent network operating systems, namely OpenDaylight (ODL) [Med+14] and Open Network Operating System (ONOS) [Ber+14]. The authors explored the attack surface of SDN by actually attacking each layer of the stack. Figure 3.1 maps used attack vectors to the SDN architecture to highlight the component and interface affected by each attack.

Security Issue	SDN Layer Affected or Targeted				
	App Layer	App-Ctl Interface	Ctl Layer	Ctl-Data Interface	Data Layer
UnauthorizedAccess, e.g.					
Unauthorized Controller Access/Controller Hijacking			✓	✓	✓
Unauthorized/Unauthenticated Application, e.g.	✓	✓	✓		
DataLeakage, e.g.					
Flow Rule Discovery (Side Channel Attack on Input Buffer)					✓
Credential Management (Keys, Certificates)					✓
Forwarding Policy Discovery (Processing Timing Analysis)			✓	✓	✓
DataModification, e.g.					
Flow Rule Modification to Modify Packets			✓	✓	✓
Malicious/CompromisedApplications, e.g.					
Fraudulent Rule Insertion	✓	✓		✓	
DenialofService, e.g.					
Controller-Switch Communication Flood			✓	✓	✓
Switch Flow Table Flooding					✓
ConfigurationIssues, e.g.					
Lack of Authentication Technique Adoption	✓	✓	✓	✓	✓
Policy Enforcement	✓	✓	✓		
Lack of Secure Provisioning	✓	✓	✓	✓	✓
SystemLevelSDNSecurity, e.g.					
Lack of Visibility of Network State			✓	✓	✓

Table 3.1: Categorization of the security issues associated with SDN frameworks by layer/interface affected [SNS16].

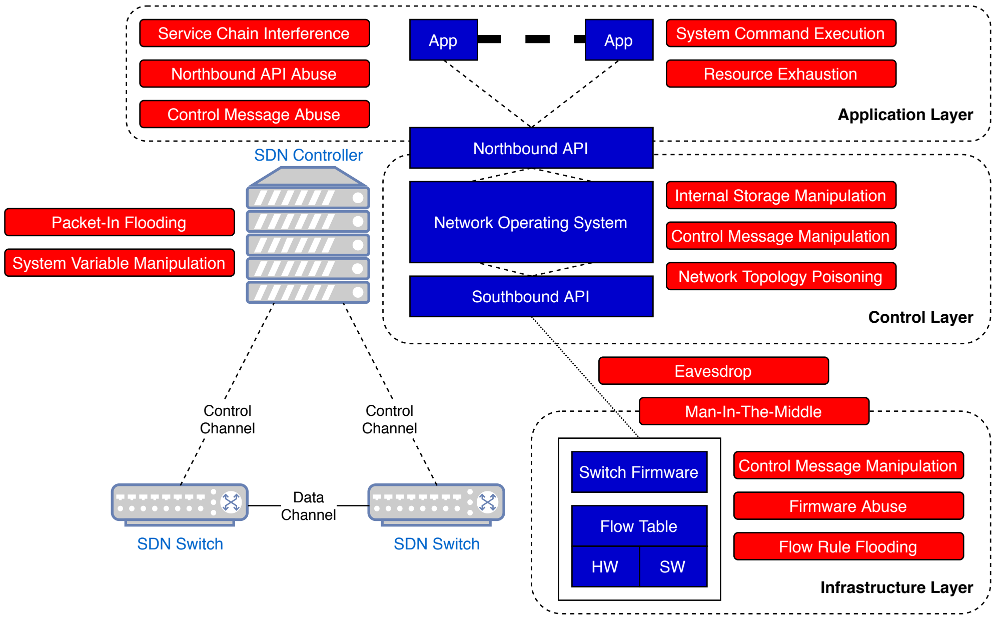


Figure 3.1: Overview of misuse and attack vectors against SDN [YL16].

3.3.1.2 Denial of Service in Software-defined Networking

As shown in Table 3.1, DoS is one of the leading security issues in SDN. In addition to traditional DoS attacks, the intelligence centralization and vertical split into three main functional layers (infrastructure, control, and application) expand the attack surface and inspire attackers to develop new techniques. Those attacks are classifiable into three categories depending on their target in the SDN architecture.

Application Layer Denial-of-Service Attacks

Two methods can disrupt regular traffic to and from the application layer. First, directly targeting an application may consume the resources allocated to it, and results, therefore, in denial of service. Furthermore, it can affect other applications since the isolation of applications and resources in SDN remains an issue [YY15]. Second, the whole layer can be isolated from the control layer by targeting the Northbound API.

Control Layer Denial-of-Service Attacks

A denial of service in the Control layer can arise by targeting any of its components: (i) the controller, (ii) the Northbound API, (iii) the Southbound API, (iv) the Westbound API, or (v) the Eastbound API. For instance, forcing different applications to generate many conflicting flow rules may lead a controller to an unpredictable state. Besides, new flows, with no match in the flow table, requires that the data plane sends a Packet-In event to the controller to ask for new rules. Generating a large number of distinct flows leads to a large volume of packet-in events. This attack may, thus, exhaust the system resources of an SDN controller machine, as well as the channels between switches and the controller [Zha+16].

Infrastructure Layer Denial-of-Service Attacks

In this kind of attack, attackers exploit weaknesses in OF switches and the southbound API. A typical OF switch consists of a switching Application Specific Integrated Circuit (ASIC) and an OpenFlow Agent (OFA). The first component is a piece of hardware holding one or more flow tables. Then, the second is a software agent responsible for the communication with the controller through the control channel.

As mentioned, the switch needs to install new rules to handle incoming new flows. In this process, the OFA generates flow requests and sends them to the controller. In the meanwhile, the switch stores the packets belonging to those flows and waits for the controller to answer. Generally, the number of flow requests sent per time unit is limited.

Wang et al. [Wan+14] demonstrate that hardware switches like HP Procurve and Pica8 Pronto can only generate less than 1000 requests per second. Overwhelming this bottleneck saturates the control channel, overloads the memory of the switch, and forces it to drop legitimate flows.

Moreover, generating fake flows fills the flow tables with useless rules. The switch keeps those rules as long as the flows are active. Consequently, rules for normal network flows can not be stored [Sez+13].

3.3.2 Software-defined Networking for Security

SDN provides multiple opportunities, not only to revisit old security concepts but also to introduce new techniques. As mentioned in [YY15], multiple SDN features can be useful to enhance network resilience. Network-wide knowledge, cumulated in the controller, facilitates the validation of security policies. Besides, it enables quick identification and resolution of any conflicts [McB+13]. As a result, consistent security policies can be built and maintained. Fourth, SDN supports software-based traffic analysis. The latter opens the door wide for innovative ideas and can employ all kinds of intelligent algorithms, databases, and any other software tools.

The study by Chi et al. [Chi+14] presents different concepts on how to integrate the Snort IDS into an SDN-based network. Based on the traditional approach, where a switch mirrors the traffic for the IDS by copying it over to an additional port, this could also be done in the SDN context using flows. When an alarm occurs at the IDS, the SDN controller initiates a rule installation on the switch. The switch then blocks the traffic similar to a firewall. Another method presented is the implementation of an IDS as a controller application. The controller-level IDS inspects the traffic sent to the controller at a `Packet-In` event. This co-location makes it possible to forward or reject the traffic directly. This proposal is resource-intensive and is not recommended by the authors since the `Packet-In` event's purpose does not fit this type of traffic handling. A third proposal includes a parser that emulates OSI layers two to four signatures using flows. This approach installs said flows only once on the switch to filter traffic and, thereby, reduce the load on the IDS. Some of the concepts presented in the study provide feedback between the IDS and the SDN controller. This concept is also part of the approaches in this thesis.

With SDNIPS, Xing et al. present a parser [Xin+14] that uses the information from alerts about attacks and installs rules in the network from this information. In their testbed, they use a server with Xen hypervisor, Open vSwitch, and Snort as IDS to map a network node in a virtual environment. Snort generates a file in JavaScript Object Notation (JSON) format for each attack, which is evaluated by a background process. This process extracts the source and destination IP address, as well as the source and destination TCP ports, from the alarm to create new rules for this combination on the Open vSwitch. The actions of the rules are traffic diversion, filtering, and isolation. In the evaluation, the authors compare the new approach to a traditional network where Snort operates as an IDS and reports the alerts to the Linux IPtables firewall. As the number of ICMP attacks (in packets per second) increases, the detection rate of Snort in the traditional approach continuously decreases. At the same time, SDNIPS for up to $25,000$ packets per second still detects over 95% of attacks. This paper shows the achievable performance gain when transferring IDS to a software-defined approach. The article does not give exact information about the type and number of flows that the method installs on the Open vSwitch after attack detection. An indication of the required OF version is also missing. This information is essential to be able to quantify the performance requirements for Open Flow switches, which are necessary to transfer the concept to a native setup. Apart from the attack detection rate and the CPU utilization of the server, the authors name no other metrics. They neither measure the delay caused by SDNIPS and the network throughput, nor do they perform an analysis of attacks for false positives. Therefore, the evaluation is missing essential metrics about the performance of the presented approach regarding performance and security. In parts, the testbed used is similar to the scenarios used for this thesis (see Section 4.1). However, the evaluation of the approach in the paper is incomplete. Although we follow related paths in this work, they differ in the flow actions and the metrics measured.

Yoon and others have designed and implemented the security functions firewall and IDS as SDN-based applications [Yoo+15]. A `PacketIn` event triggers sending traffic to the SDN controller and processing it by a corresponding application. The firewall comprises various security policies, which result in the forwarding or filtering of traffic. The implemented IDS can read and process existing Snort signatures. In a discussion, the authors compare the advantages and disadvantages of the procedures. For this purpose, they paid attention to compliance with SDN paradigms and evaluated and weighted their violations. Finally, they evaluated the two approaches in different testbeds with different servers and native OF switches. Here, the IDS was operated both as an in-line equivalent and as a passive IDS like a mirror port in classical networks. For the evaluation, the choice fell on the byte throughput metric. The implemented stateful firewall also underwent an examination concerning the delay. Metrics regarding attack detection or CPU usage of the controller server were not measured.

3.4 Network Function Virtualization

Multiple works deal with the NFV domain. They either deal with (i) NFV frameworks, (ii) methods to benchmark NFV frameworks and VNFs, (iii) the possibilities that NFV offers in combination with SDN, or (iv) the implementation of network function as VNFs.

Regarding the different available frameworks for developing high-performance virtual network functions, Barbette et al. [BSM15] as well as Gallemnueller et al. [Gal+15] provide a detailed comparison of various development kits and deployment frameworks and present their different features. These frameworks, like DPDK, netmap, and PF_RING, provide high-throughput, low latency network IO by bypassing the network kernel of the operating system. Also, the recently proposed XDP framework [Høi+18] bases on the extended Berkeley Packet Filter. It allows the implementation of network processing functionality directly within the operating system kernel, albeit with limited processing complexity. Finally, the availability of P4-enabled devices and other Smart NICs allows the offloading of VNF functionality onto programmable hardware [KV16]. Note, that due to the complexity of the network function developed in this work, the choice fell on DPDK as the development framework due to it providing high performance while maintaining implementation flexibility.

In the context of the performance evaluation of softwarized network functions, [Chi+12] provides an extensive list of best practices and caveats and [Bon+15] and [Lan+15] performed similar performance benchmarks of softwarized network functions. Moreover, the authors of [RBR17] and [Cao+15] present different frameworks for the performance evaluation of network functions in virtualized environments.

A detailed investigation related to the use cases of SDN and NFV in the context of network security, as well as the advantages and disadvantages of these approaches, is presented in the survey by Lorenz et al. [Lor+17]. Farris et al. provide an extensive overview of emerging SDN and NFV security mechanisms in the context of the Internet of Things [Far+19].

Security services are a typical example of traditional network service functions that can benefit from adopting NFV and dynamic SFCs. First, NFV offers additional flexibility and elasticity, thus, allowing to adapt the security function to attack volume and composition. In select cases, VNFs succeed even to outperform traditional instances. To illustrate, Bremler-Barr et al. [Bre+14] extracted the DPI functionality from Snort and converted it into a common service VNF. The authors validated their approach on a single Snort-based IDS service in an emulated SDN environment. The tests demonstrate that the DPI VNF performed 67% to 86% better than two separate traditional Snort instances.

3.5 Security Function Chaining

SDN allows enabling NFV infrastructures with SFCing. In this section, we introduce several approaches to function chaining.

In addition to introducing the possibilities for dynamic SSFCs, SDN also allows augmenting them with traffic and application awareness [Li+17].

Many solutions rely on SDN and NFV in the context of DDoS resiliency. Bohatei [Fay+15] is a flexible and elastic DPS demonstrating the advantages of these emerging network management paradigms. It consists of an SSFC, running on VMs scaling elastically concerning the volume and composition of the attack. Bohatei steers suspicious traffic through the defense VMs and, at the same time, minimizes user-perceived latency and network congestion. The evaluation proves that this solution can handle attacks of hundreds of Gbps. It is also able to mitigate multiple dynamic attack strategies successfully. It is notable that Bohatei also responds to several canonical DDoS attacks in less than one minute.

Blendin et al. state in their position paper [Ble+14] that while NFV "promises high flexibility and lower costs for network operators at the same time," the chaining of network services remains a demanding challenge. This challenge arises due to frequently changing operator requirements and resources for new service functions requiring fast deployment and scaling capabilities. A significant issue is that upon the release of the work, preconfigured service chains consisting of static configurations were prevalent. Such static SFCs rely on using fixed hardware and software configurations with little to no flexibility. Thus, the adoption of existing service chains to unsteady requirements needs a time-intensive and manual reconfiguration. The authors introduce three available solutions: (i) *Network Service Headers*, (ii) *StEERING*, and (iii) *SIMPLE*. Using specialized interfaces network service headers allow realizing service chaining. *StEERING* relies on Layer 2 header modifications to accomplish a way of SFCing, not allowing mapping of packets to users and service chains. Third, the *SIMPLE* approach is to correlate packet headers on entering and exiting the traversed SFCs. This approach is complicated and requires accurate packet matching. Also, the method only applies to SFCs, where all packets that enter an SFC also exit the SFC. The authors then propose an SDN-based approach avoiding packet-based matching. They aim at simplifying the deployment process to minimize network configuration changes over the deployment by isolating service instances. The concept for SFCing applies OF and Layer 2 addresses to identify the service instances. Thus, it can map traffic to its user and service chain using ingress and egress links for reference points. The paper also includes a proof-of-concept implementation demonstrating its feasibility and usefulness.

Mohammed et al. present an "SDN controller for Network-aware Adaptive Orchestration in Dynamic Service Chaining" [Moh+16]. Emerging technologies like SDN and NFV introduced new architectural models and techniques for SFCing and the orchestration of application and network services. The problem presented by the authors is the effective deployment and governance of such scenarios. The paper's idea is to offer effective management and orchestration mechanisms through an SDN controller and provide a Northbound Interface that allows SDN applications to describe their network requirements without requiring details of low-level implementation. The proposed SDN controller underwent testing using Mininet and various topologies of the network. Results show the load balancing capabilities during complex traffic chaining function and that no packets elude elements of the chain when reconfiguring data transmission routes. However, such a feature also raises the delay and the total number of messages transmitted in the network. In summary, the paper introduced an SDN framework that handles and orchestrates SDN applications efficiently, thus providing a Northbound API for SDN apps to explain their specifications without having low-level network information.

StEERING [Zha+13] is a system for dynamic routing of traffic through any middlebox series. The problem the paper addresses is that there are no protocols available in classic networks for routing traffic through middleboxes. Providers have no choice but to use low-level and complicated software to produce the desired chaining, giving network operators the freedom to direct traffic in compliance with consumer granularity and form of traffic. The proposed framework has four design requirements:

- Efficient routing of traffic by ensuring traffic only passes through a chain specified by the network operator and without passing through unnecessary middleboxes.
- Versatility by simplifying adding or removing service chains, and supporting specific policies for subscribers, applications, and operators.
- Support for a high number of rules and the ability to scale with the growth of subscribers and applications.
- Enable every form of a middlebox to be implemented independently of its provider.

In summary, the paper describes a structure within a Software-defined Network for handling traffic across middleboxes. *StEERING*'s powerful forwarding and scalability proved themselves in a prototype implementation, which results in significant reductions in latency.

In their paper "Dynamic Chaining of Virtual Network Functions in Cloud-Based Edge Networks" [Cal+15], Callegati, et al. discuss space and time diversity regarding service chaining. Explicitly, the nature of the SDN control plane within a cloud-based edge network is studied by looking into the necessary actions provided in OF switches to accomplish dynamic service chaining inside network feature implementations of Layer 2 and Layer 3. The authors define the steps required to achieve network function chaining. The paper introduces proof-of-concept implementations for dynamic feature chaining utilizing Layer 2 and Layer 3, which demonstrates that dynamic service chaining is possible in SDN and that SDN controller architecture offers possibilities for streamlining. Numerous Mininet emulations checked the implementations, showing that dynamic function chaining functions in SDN.

Multiple papers state which topics in the area of SFCing are still open. These topics include (i) rule anomalies [Li+18], (ii) intelligent positioning [Lui+15], (iii) and effective provisioning [Sha+19].

3.6 A Security Plattform for Network Function Virtualization

Security Function Chaining (SFCing) is a significant component of this work and essential for complex NFV security frameworks. The first inspiration for this work was the "Security Position Paper: Network Function Virtualization" by Milenkoski et al. [Mil+16]. This position paper from the Cloud Security Alliance (CSA) is concerned that with the rise of cloud infrastructures, security risks have risen as well. The document, therefore, discusses security issues and concerns for NFV environments. The authors propose six NFV security challenges.

(i) *Hypervisor dependencies:* Currently, the market is dominated by only a few hypervisor distributors, with many vendors looking to become market players. Like their vendor counterparts for the operating system, these vendors must solve security vulnerabilities in their code. Proactive patching is critical. Such providers must also understand the underlying infrastructure, e.g., how packets move within the structure of the network and different types of encryption.

(ii) *Elastic network boundaries:* Throughout NFV solutions, the architecture of the network accommodates many features. The placement of physical controls depends on their position and the length of the cable. In NFV architecture, these boundaries are vanishing, complicating security issues because of the unclear boundaries. Traditionally, using Virtualized Local Area Networks (VLANs) alone does not meet the security requirements even though physical separation still can be necessary for some uses.

(iii) *Dynamic workloads:* The benefit of NFV is its versatility and dynamism. Traditional models of protection are stagnant and incapable of changing when network topology shifts to react to requests. Introducing security services into NFV architectures often encompasses having to rely on an overlay model that does not easily coexist across the seller's borders.

(iv) *Service insertion:* Since the fabric routes packets that meet configurable criteria intelligently, NFV promises elastic and transparent networks. Traditional network managers deploy security controls logically and physically in-line. With NFV, there is often no simple insertion point for security services not already integrated into the hypervisor.

(v) *Stateful versus stateless inspection:* Today's networks require redundancy at the system level and along the network path. This route redundancy creates asymmetric flows that present challenges to stateful systems that need to see each packet to provide access controls. During the last decade, security operations based on the premise that stateful inspection is more advanced and superior to stateless access controls. NFV may add additional complexity where security controls can not handle the asymmetries created by multiple redundant routes and devices in the network.

(vi) *Scalability of available resources:* As noted earlier, the appeal of NFV lies in its ability to do more with fewer data center rack space, electricity, and cooling. The dedication of cores to workloads and network resources enables the consolidation of resources. DPI technologies — for example, next-gen firewalls and TLS decryption — are resource-intensive and do not always scale without offload functionality. Security controls must be universal to be effective and often require significant computing resources.

Based on these challenges, they also infer six risks when deploying SDN and NFV in cloud environments: (i) NFV and hypervisor compatibility, (ii) system availability, (iii) SDN architecture, (iv) SDN implementation, (v) policy consistency, and (vi) compatibility with Infrastructure as a Service (IaaS)

On the other hand, they also envision opportunities for an NFV security framework. Such a framework can reduce deployment and management resources by using commodity hardware. Moreover, such a framework could offer additional flexibility regarding (i) on-Demand deployment and scalability, (ii) dynamic threat response, (iii) global and realtime view, (iv) flexible response, and (v) NFV and SDN enabling software-defined security.

Next, the authors propose a security framework that can adapt the ordering of its security appliances depending on an incoming attack. They detail an enterprise-grade architecture for such a framework.

This work inspired this thesis. Therefore, on the following pages, we will propose and evaluate VNFs solutions to integrate inside an SDN-enabled network. Next, we will present our approach to allow the dynamic adaptation of SSFCs to malicious traffic.

Chapter 4

Augmenting Single Security Functions using Software-defined Networking

The amount of attacks that security functions must take care of grows at a rapid pace. This growth exceeds the growth of computing resources relative to their price. Thus, these circumstances force service providers to increase their spending on hardware or cloud resources for security functions leaving less budget for the actual service, research, and development or even making services unprofitable.

In our opinion, the best approach to counter this effect is to increase the efficiency of single security functions and the combination of multiple security functions. This chapter deals with single functions while Chapter 5 deals with multiple security functions inside SSFCs.

We present a solution to dynamically bypass IDS to the end of increasing the throughput of the system while keeping security metrics at a high level. To this end, we leverage SDN to reroute the traffic when needed.

Furthermore, we introduce a solution to combat TCP SYN flood attacks. By using SDN and NFV, we alleviate the shortcomings of existing solutions for this attack.

Research Questions

In this chapter, we will tackle several research questions. All of the following research questions are part of the meta-research question **MRQ 4:** *How and to what extent can SDN help improve the performance of (security) network functions?*. The numbering of these research questions maps to the sections of this chapter. If a section deals with more than one research question, those questions have their number appended by ascending Latin letters.

RQ4.1a How can SDN-based approaches improve Intrusion Detection Systems?

RQ4.1b What effects do bypassing approaches have on the performance and security of Intrusion Detection Systems?

RQ4.1c How do adaptive approaches, which perform reconfigurations at runtime, compare to static approaches?

RQ4.1d How do different workload levels impact the performance and security of the SDN-based approaches?

RQ4.1e Do the SDN-based approaches change their behavior when using hardware or software switches?

RQ4.2a How can SDN-based approaches improve DDoS Protection Systems against SYN flood attacks?

RQ4.2b What is necessary to make such a solution stateless and independently deployable?

RQ4.2c How does such a solution perform compared to existing solutions?

RQ4.2d To what extent grade can parallelization improve the performance of such a solution, and how vital is parameter-tuning?

RQ4.2e Which deployment and scaling strategies suit the solution?

Chapter Structure

We at first present, implement and evaluate a solution to bypass IDS dynamically using SDN in Section 4.1. Next, we introduce TCP Handshake Remote Establishment and Dynamic Rerouting using Software-defined Networking (THREADS) — a DPDK-based and SDN-enabled security function to mitigate TCP SYN flood attacks in Section 4.2. Last, in Section 4.3, we conclude the chapter with an analysis of the gathered answers to the previously stated research questions.

4.1 Dynamic Network Intrusion Detection System Bypassing

In addition to firewalls, using NIDS has become a security standard for data centers. The constant work of security researchers and the community ensures a regular surge of new signatures for IDS to defend against attacks. However, the use of NIDS is proving inflexible for cloud solutions, which must react to new requirements within the shortest possible time. As SDN and NFV solutions are increasingly finding their way into data centers, the question arises as to whether and how active cooperation between these systems and NIDS can contribute to further gains in performance and security.

This section comprises the following contributions to evaluate the potential of SDN and NFV solutions.

We design SDN-based algorithms for handling network traffic, including the detection of attacks by the NIDS:

Adaptive Blacklisting: The traffic of specific (potentially untrusted) protocols reaches the NIDS for a defined time interval. If the traffic of a considered connection triggers an alarm in this interval, it becomes permanently redirected via the NIDS. Other traffic continues without redirection.

Adaptive Whitelisting: Only whitelisted (trusted) protocols pass directly to the destination. The remaining traffic passes through the NIDS. If the traffic from a specific connection does not trigger an alarm after a certain number of transmitted packets, the traffic of this connection is whitelisted.

Selective Filtering: Incoming traffic of selected protocols is permanently redirected via the NIDS and only then forwarded to the original destination. Outgoing and other incoming traffic usually continue to the destination without the detour via the NIDS.

We implement these algorithms as an SDN-Controller App. We then perform multiple experiments inside a testbed environment to measure the performance and security metrics throughput, successful HTTP requests, TCP handshake duration, request residence time, and detection rate of attacks by the NIDS. From these metrics, we derive further security metrics, as described in Section 2.1.2.1. We evaluate and discuss the results of the measurements and the computed metrics during the experiments.

The results are promising. The dynamic approaches can remove a majority of the negative performance impact from the NIDS. The detection accuracy remains high within the margin of error to 100% in most scenarios. However, this only applies when using a software switch. When applying the algorithms on a hardware switch, this switch reduces the improvements in performance.

Furthermore, the detection rate falls to inferior values, while the other security metrics stay high. The Selective Filtering shows that, with little effort, a static solution can improve performance. While it does not reach the performance of the other approaches, the hardware switch has only a marginal effect on the performance of Selective Filtering and does not reduce detection accuracy.

In Section 4.1.1, we present the underlying problem and our approach. We detail the implementation of the approach in Section 4.1.2 and evaluate it in Section 4.1.3. Finally, Section 4.1.4 discusses the results and concludes this topic.

4.1.1 Approach

In this section, we describe our approach aiming at increasing NIDS performance while still maintaining appropriate security metrics. We first describe why NIDS performance can often become a problem. Next, we introduce our two dynamic algorithms, "Adaptive Blacklisting" and "Adaptive Whitelisting." Finally, we present a more straightforward and static (no dynamic insertion of new rules at runtime) approach "Selective Filtering" to the problem as a baseline for comparison to our dynamic algorithms.

4.1.1.1 Problem Statement

We introduced IDS in Section 2.3.1. There, we also provide further information (e.g., classification). In the following, we will use real-time, misuse-based, and non-distributed NIDS. A NIDS is a software application or a hardware device that is generally placed inline at a point before entering into an internal network. A NIDS is responsible for monitoring and analyzing the network packets for any anomalies or to detect malicious behaviors. Based on predefined rules, the NIDS either drops malicious packets or forwards the packets, thereby protecting the entire internal network. The current implementations of NIDS usually make use of a technique called DPI, inspecting the whole network packet in detail, including headers and body. Previous studies show that the time required for DPI depends on the examined protocol as well as the type and number of rules/signatures present in the NIDS.

Consequently, such detailed analyses of the DPI technique require many computing resources, resulting in making them a likely bottleneck in the network infrastructure. The NIDS can turn into a bottleneck, as well, as they require inline placement before entering a corporate network infrastructure to also act as IPS. Also, studies have shown that under overload conditions, NIDS can experience packet loss, introduce network delays, and thus result in reduced network bandwidth.

Additionally, under overloaded conditions, employing NIDS can result in a high number of false-positive alarms making them useless and ineffective in the network. It is possible to counter these deficiencies by adding multiple NIDS instances and load balancers. However, this is (i) very expensive, since the massive performance demand from DPI requires more resources for the NIDS than for the actual protected service. We later show that for the used ruleset, it takes twelve NIDS instances to handle a 1 GBit/s link. Furthermore, (ii) it can open new vulnerabilities (e.g., in the load balancer). Also, new botnets are capable even to overload scaling security solutions [Kol+17a]. Therefore, alternatives to just adding additional resources need exploration.

Furthermore, attacking hosts launch most of the attacks as fast as possible so that the host stays in contact with the victim machine for the least amount of time. A short connection duration allows reducing the chances of an attack getting detected. This characteristic creates an opportunity to design adaptive algorithms that make use of such findings to dynamically adjust the routing of packets either to the NIDS or directly to the protected service. These bypassed packets would not require any computing resources on the NIDS freeing resources to analyze the remaining packets.

4.1.1.2 Initial Situation

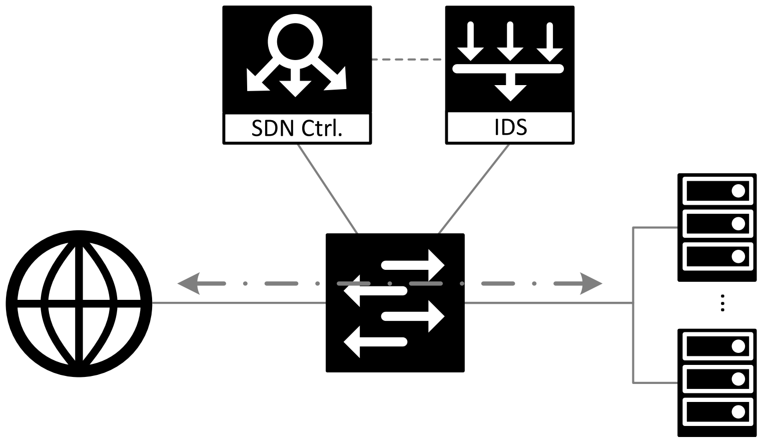


Figure 4.1: Traditional Switching: Direct routing from source to sink. NIDS inline mode is possible with a single SDN Flow.

In the initial situation, the network consists of three segments, as shown in Figure 4.1. First, an external network (in most cases, the Internet) is the source for potentially malicious network packets. Next, the security portion of our network consists of an SDN-enabled network (represented by a switch), an SDN controller, and the NIDS. Finally, the third segment is the protected internal network. In the default configuration of all components, all traffic is routed directly between the external network and the internal network. Hence, the default configuration provides no protection. Resembling the inline deployment typical for NIDS requires a single additional flow. This flow forwards all packets from the external network to the NIDS. Then, it forwards the benign packets received back from the NIDS to the internal network.

4.1.1.3 Adaptive Blacklisting

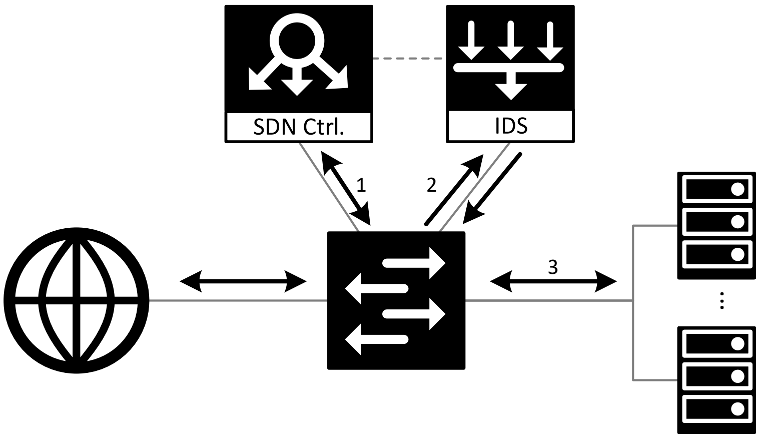


Figure 4.2: Connection establishment via adaptive Blacklisting. The switch queries new connections for observed traffic types with the controller (Flow 1). The controller creates a new flow forwarding traffic to the NIDS (Flow 2). If no alert occurs for some time, the network directly forwards traffic to the service host (Flow 3).

Adaptive Blacklisting distinguishes between blacklisted and non-blacklisted traffic. The principle behind Adaptive Blacklisting is to initially only route those packets from applications, services, and protocols via the NIDS for which the NIDS has configured signatures. Therefore, this approach puts traffic types with signatures on the blacklist. Blacklisted traffic can, but does not necessarily have to be, malicious. The SDN does not route traffic for other services via the NIDS and instead forwards it directly to its destination without redirection. This distinction eliminates the load on the NIDS from traffic for which it does not have any potentially matching signatures. **RQ 4.1a**

In contrast to the existing static blacklisting approaches (e.g., the Selective Filtering described in Section 4.1.1.5), when using Adaptive Blacklisting, connections are removed from the blacklist once they have not triggered an alarm for a certain amount of time. When a new connection supported by the NIDS arrives, the system requests for instruction from the controller, as depicted as Flow 1 in Figure 4.2. After confirming that the requested connection is indeed new and not already in the database, the controller creates two flows with different durations. The first flow forwards the network traffic to the NIDS. The figure depicts this flow as Flow 2. This flow has a higher priority but a shorter lifetime (X). Many attacks have to occur within the first few packets after establishing a new connection, so that they give an administrator the least amount of time to detect them or because the vulnerability only occurs at the beginning of a request. Therefore, once the lifetime of Flow 2 times out, a second flow (Flow 3), created at the same time as Flow 2, forwards the traffic directly to the host destination by bypassing the NIDS. This flow has a lower priority but a higher lifetime (Y). If the NIDS detects attacks in packets traversing Flow 2 before Flow 2's lifetime (X) times out, the SDN controller makes Flow 2 is permanent, and all the traffic from this host will pass through the NIDS without compromising the system.

Our solution allows configuring the lifetime timeouts described before, i.e., if to achieve higher security (larger ratio of packets routed via the NIDS) and accept lower performance, an administrator could configure longer timeouts. Lowering the timeout values results in potentially lower security (potential premature direct routing of malicious packets) and higher performance. Additionally, we can configure a timeout for the lifetime of bypassing the NIDS (Flow 3) to be either permanent or temporary. In the permanent mode, once the rules allowed the source host to bypass the NIDS, its traffic will never pass through the NIDS again. On the contrary, in the temporary mode, the state from Flow 3 will switch back to Flow 2 to recheck if the connections are still benign. Having the algorithm run in temporary mode also helps ensure that the switch clears its flow tables regularly, and does not lead to overflow of flow tables in the switches. In permanent mode, it is possible to eliminate inactive rules by setting a soft timeout. A soft timeout removes rules without traffic, triggering them for more than a configured amount of time). An alternative – if the protocol supports it – is adding another detection in the controller to detect a disconnection sequence (e.g., TCP's connection termination [DSC74]). Hence, we expect this adaptive algorithm to reduce significantly the number of packets that pass through the NIDS, which, in turn, increases the NIDS performance and keeps the CPU utilization of the NIDS host low.

4.1.1.4 Adaptive Whitelisting

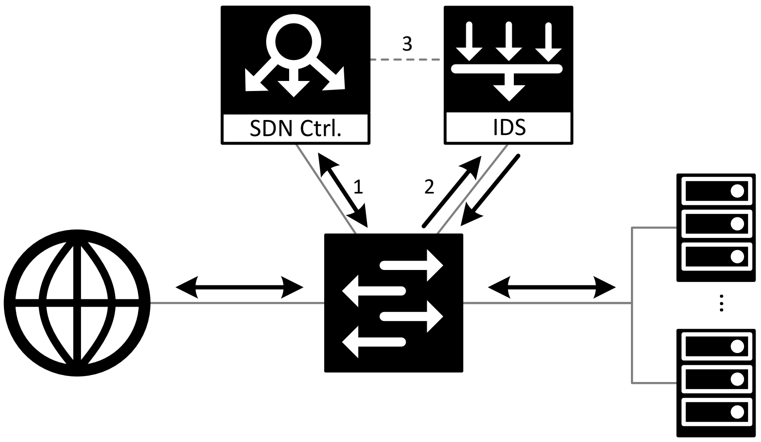


Figure 4.3: Procedure for handling a connection by Adaptive Whitelisting. The switch queries the controller for non-whitelisted traffic (1). The controller creates a flow diverting the traffic to the NIDS (2) and then queries the NIDS for the observed attacks (3). If recorded attacks lie below a threshold after a fixed time, the diversion stops.

The fundamental concept behind Adaptive Whitelisting is that unlike the Adaptive Blacklisting, it does not necessarily require any knowledge about the configured signatures in the NIDS. This approach at first routes all the network traffic except for optional explicitly whitelisted traffic types via the NIDS.

For every connection, the receiving component initially queries the SDN controller, which creates a flow via the NIDS. If, after a preset number α of packets, the NIDS has raised less than β alerts, the traffic becomes whitelisted, resulting in an additional network rule. The subsequent traffic of the tested connection no longer passes through the NIDS and instead proceeds directly to its destination. A time limit Z, is configurable after which the whitelisted traffic needs to undergo inspection again. The number of packets routed through the NIDS in order to work effectively requires empirical studies. The system requests for instructions from the controller upon arrival of a new connection. Figure 4.3 depicts that action as Flow 1. Like Adaptive Blacklisting, once the controller confirms that the requested connection is indeed new and without a previous entry, it sets up a single flow (Flow 2) with the highest priority passing all the network traffic through the NIDS. The SDN controller now communicates with the NIDS to record the packet characteristics detected for the newly created connection. The connection between NIDS and SDN controller (Flow 3) in the figure represents this communication between the SDN controller and NIDS.

If, after α packets passed via the NIDS, the SDN controller records that β or more packets have triggered alerts, it routes packets from that connection permanently via the NIDS. In our configuration $\beta = 1$, i.e., if even one alert occurs in the sample space of α packets, this suffices to mark the connection as malicious. Otherwise, if less than β alerts occur, the subsequent network traffic from this connection passes directly to the host destination by bypassing the NIDS via another flow. Like Adaptive Blacklisting, a predefined lifetime is present until the packets for a connection can bypass the NIDS. On the expiry of the lifetime, the switch deletes the rule, and the network traffic for that connection will pass through the NIDS again to prove the connection is still benign. Also, these timeouts are configurable like those explained for Adaptive Blacklisting, and a permanent mode is possible as well.

4.1.1.5 Selective Filtering

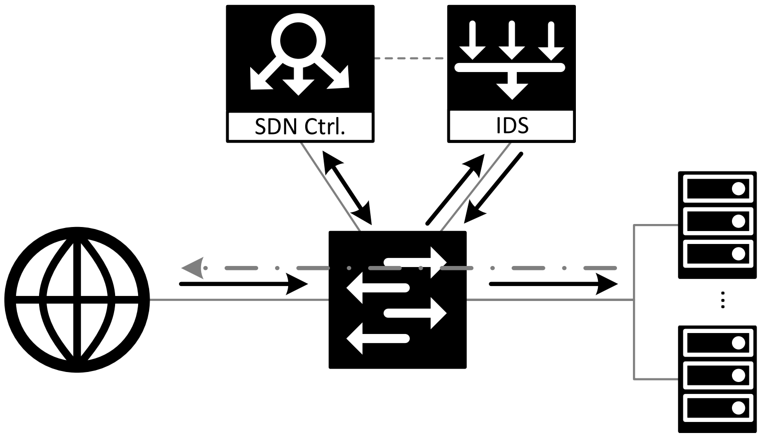


Figure 4.4: Flows for Selective Filtering for a diversion via the NIDS. This configuration directly forwards traffic that has no signatures configured at the NIDS to the service host. Traffic with configured signature has to pass through the NIDS.

Unlike Adaptive Blacklisting or Adaptive Whitelisting, Selective Filtering is not an algorithm but rather a simple SDN-based static solution that helps to baseline our dynamic approaches. Such static flows, once established on the switch upon its first connection to the SDN controller, do not change during operation. The concept of Selective Filtering requires only a few flows reducing the initial overhead of the SDN-based traffic analysis. In most production deployments, for optimal performance, most NIDS use signatures for a limited set of applications, protocols, and services. Selective Filtering attempts only to statically route traffic over the NIDS for which protocol or service signatures are available.

This distinction requires knowledge about which application workload is running on which host and protected by which NIDS. For each host server and application, Selective Filtering adds a flow entry in the switch, which redirects all incoming traffic to this combination via the NIDS. The remaining network traffic continues directly to its destination. Figure 4.4 depicts this approach.

One of the most significant advantages of the Selective Filtering approach is its simplified deployment. Nevertheless, the NIDS can continue to meet its task of detecting attacks, as potentially malicious traffic passes through the NIDS. Due to its simplicity and compactness, Selective Filtering is highly compatible with other security systems such as firewalls or Demilitarized Zones (DMZs) without adaptation.

4.1.2 Implementation

We implemented the algorithms presented in Section 4.1.1 to allow their evaluation. Additionally, we realized a load generator. This section gives a short overview of the employed technologies.

As the SDN controller, we use Ryu. Ryu is lightweight, supports basic switching and REST per default, and provides for a simplified extension using simple Python scripts. We realized every algorithm as a single Ryu module.

The choice for the NIDS fell on Snort in version 2.9.9.0. This version does not support parallel processing and, therefore, setting it under load is feasible with limited resources. Detected attacks are provided by Snort using its internal database.

Our experiment controller, written in Java, controls the service host as well as the client(s), generates the workloads, executes the experiments, and records its results.

4.1.3 Evaluation

In this section, we evaluate our approach. First, we describe the used testbed. Next, we introduce the used metrics, configuration scenarios for the testbed, and workloads. These sections also contain technical information about the acquisition of said metrics. Last, we present and assess the measured results.

4.1.3.1 Testbed

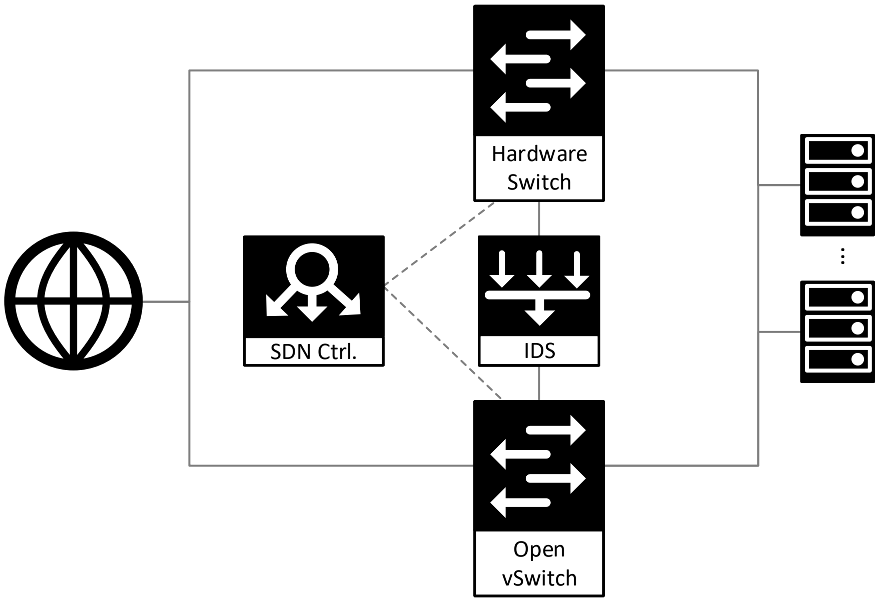


Figure 4.5: Testbed used for evaluation comprising benign and malicious traffic generation (simulating an external network), a NIDS, the SDN-enabled hardware and software switches able to send traffic either to the NIDS or service host, an SDN controller, and service hosts.

The testbed we used to evaluate the presented approach comprises multiple servers and one Open Flow-enabled hardware switch, as depicted in Figure 4.5. The servers take the roles of simulated clients, load driver, target server, software switch, NIDS, and SDN controller with no server covering multiple rules to prevent side-effects. We used HPE ProLiant DL360 Gen9 servers with an octa-core Intel Xeon CPU with enabled hyperthreading and 32GB main memory. Table 4.1 gives detailed information about the servers' configuration. The choice

Unit	**Value**
Product	HP ProLiant DL360 Gen9
CPU	Intel Xeon E5-2640 v3
Default CPU frequency	2.60 GHz
Max CPU frequency	3.40 GHz
Min CPU frequency	1.20 GHz
Cores (Threads)	8 (16)
Cache (L1/L2/L3)	512 KB/2048 KB/20480 KB
Memory size	32GB (2 x 16 GB) DDR4 Dual Channel
Memory frequency	1.866 GHz
Memory Connection	Dual Channel
Storage Model	HP VK0800GEFJK 800 GB SSD
Storage Connection	SATA III (6GBit/s)
Operating System	Ubuntu 18.04.3 LTS (x86-64)
Kernel	4.4.0-72

Table 4.1: Hardware specifications describing all servers inside the testbed.

for operating system fell on a 64-bit Linux with kernel version 4.4.0-72 for x86-64 architectures.

The selected switch is an HPE 5130-24G-4SFP+ from the Aruba series. This switch series has 24 1 Gbit/s and four 10 Gbit/s ports and allows configuration through a serial console and via SSH. It supports OF versions up to 1.3 as SDN protocols, and its hardware table can contain up to 384 entries. For Snort, we configured the “Server-Apache rules” from Snort’s “registred rules” set. This list is available for free to all registered Snort users. From these rules, we only enabled rule `1111`. For the role of the SDN controller, we used the python-based Ryu controller. All network connections supported a maximal bandwidth of 1 Gbit/s. An experiment controller connects to all devices via a separate experiment network. This experiment controller configured the servers and switches, starts and stops the measurements, monitors the experiment, and collects the metrics. The target service runs Apache as the webserver application.

4.1.3.2 Metrics and Their Acquisition

To evaluate the quality of our approach, we need to measure multiple metrics. These metrics comprise the throughput, the response time, and the accuracy of the attack detection. Section 2.1.2.1 gives further information on relevant metrics.

Network Throughput

A significant metric to assess the performance of web servers and, therefore, also their protection system, is the achieved throughput. This value measures the amount of traffic processed by the system.

We make use of the Simple Network Management Protocol (SNMP) to measure the throughput. This protocol is available on many switches and OSs. It provides access to many settings and counters (so-called Object Identifiers (OIDs)) of a system. These OIDs include the state and capabilities of the system's NICs, the CPU load, and memory usage. Furthermore, SNMP provides access to different counters of the system that contribute toward determining the throughput. These counters include the number of bytes received and transmitted through the network, and the number of incoming and outgoing packets, as well as possible packet errors. The throughput considered in our paper is the number of incoming and outgoing bytes to the interfaces and ports involved in an experiment at the switch. For this purpose, we use the OIDs for incoming (`1.3.6.1.2.1.2.2.1.10`) and outgoing (`1.3.6.1.2.1.2.2.1.10.16`) bytes.

We log SNMP values throughout our experiment. In the course, our logging solution takes into account the features of the protocol. These features include such as the irregular updating of counters, failed SNMP OID queries, and the conversion from absolute to relative values. An example is the number of incoming and outgoing bytes. In this case, SNMP transmits the consecutive, absolute number of bytes. Computing the throughput requires building delta values and scaling them according to the time passed.

Number of Sent and Completed HTTP Requests

While the throughput represents a network-level perspective, it does not give much information about service quality. In our case, even a high throughput could, in theory, result in no requests completed by the server. Thus, we augment the throughput results with the number of HTTP requests. For these requests, we differ between sent requests (as explained further on, we use different sending strategies for the different workloads), and the requests successfully served on time. The difference between both values results from requests that either failed (e.g., the server denied them), and requests that did not return successfully before the experiment ended.

These values are simple to compute since the load driver application can incorporate a counter that increments with every sent HTTP request. A second counter takes care of the successfully returning requests.

TCP Handshake Delay

In addition to throughput, the delay is another essential metric in computer networks. The use of additional components that a network packet needs to pass through in the network, such as the NIDS, leads to additional packet delays. A delay is the amount of time a packet needs from the source to the destination. For this paper, we use the time taken to establish a TCP Handshake.

In the case of the Adaptive Black- and Whitelisting algorithm, the establishment and start of a connection is a central process. In different steps, different types of delays arise from the very first to the subsequent packets. For example, to identify connections, the first packet is sent to the SDN controller via a `packet-in` event. Here the event, as well as the processing and feedback on the controller, introduce a delay. Besides, there is a possible redirection to the NIDS, introducing an additional delay for the packet inspection. Since many applications use the TCP protocol for the use of NIDS, the case is particularly interesting in which the resulting delay of an entire TCP handshake becomes visible because it is the prerequisite for a TCP connection. The observation of the time difference of the TCP handshake also has technical reasons. The time difference can be determined merely through two timestamps, before and after the call.

Despite the simple implementation, this approach also has some disadvantages. On the one hand, the measurement is sensitive to influences by scheduling algorithms and their effect on the operating system. Scheduling is used internally for the administration of the threads on the operating system level for the context switching of applications. When sending multiple outgoing packets, the order of packets cannot be guaranteed. On the other hand, this shortcoming becomes balanced by the duration and repetition of experiments as the average of the delays approaches the real mean.

Overall, all measurements must use the same measuring method. This characteristic allows for a uniform calculation of any constant delays. The method used is also stable against synchronization problems that exist with other server-spanning methods to determine the delay.

Request Residence Time

The residence time comprises the complete duration during which a request is in the systems. Thus, it tracks the request from the connection establishment to the connection termination. Like the number of HTTP requests for the throughput, the residence time adds a service metric to the network metric TCP handshake delay. Since the residence time comprises the handshake, it has to be at least as large as the handshake delay.

We measure the residence time by storing the timestamp once issuing the command to send a request. Once the request completes, we again take the timestamp. The difference between both timestamps equals the residence time.

Attack Detection Rate, False and True Positives, False Negatives

The attack detection rate is the number of attacks detected by Snort during the experiment. To determine the number, we used the barnyard2-managed MySQL database on the NIDS as the basis to count the number of attacks.

Barnyard2 inserts the attacks detected by Snort as an entry in the database. The difference between the number of database entries at the beginning and end of the experiment repetition is the number of attacks detected. This approach can be problematic if some detections by Snort enter into the database after the experiment due to excessive delay. This effect is statistically significant when writing entries to the database after the beginning of the subsequent experiment. However, post-analysis of the database can subsequently reduce or eliminate such effects.

A problem with the automated analysis of attack detection is the false-positive and false-negative analysis. Although barnyard2 writes all the packets that Snort has sent to an alarm entirely, the entries can contain some corrupted data. This limitation makes it difficult to search the database for patterns. For better detection of attacks, generated HTTP attacks contain the plain text "attack" (Snort does not have an existing rule to match against this pattern, so it is neither harmful nor helpful to the other metrics).

In overload situations, our experiments show that this pattern does not work reliably. As an example, we query the database for regular HTTP requests. Filtering by a unique feature that occurs only in harmless HTTP requests allows identifying false positives in the sample unambiguously. The database alone does not give enough information to determine the number of false negatives. However, if the attacks do not trigger a vulnerability, we assume false negatives to be zero.

Based on the detected attacks, the false and true positives, as well as the false negatives, we can then compute the precision, recall, and f-measure. Since we did not measure the true negatives (this would require an excessive amount of logging resources with little gain in significance), we can not compute the accuracy.

4.1.3.3 Scenarios

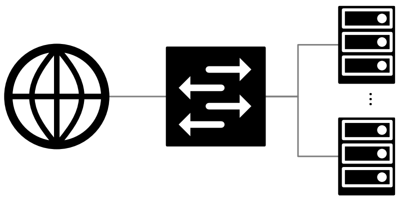
We evaluate our approach in multiple scenarios within the described testbed. These scenarios include baseline scenarios as well as scenarios testing the algorithms presented in the approach using hardware and software switches. Figure 4.6 shows the used scenarios, as described below.

Scenarios 1a and 1b: Baseline Scenarios

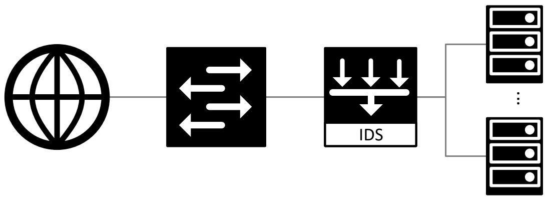
The goal of the reference scenarios is to baseline the setup on the two criteria of our interest, namely performance and security. Therefore, Scenario 1a baselines for maximum performance, whereas Scenario 1b baselines for maximum security. We use results from these scenarios as the minimum, respectively, maximum achievable values for the algorithms.

Scenario 1a consists of only a switch as a node between the source and the destination. Figure 4.6a depicts this scenario. This scenario forwards traffic directly from the external network to the service hosts and back. Hence, the results from this scenario represent the maximum data throughput and the number of successful HTTP requests combined with the lowest TCP handshake delay and request residence time. The only limit to the network capacity between the source and the destination is the maximum bandwidth of the switch.

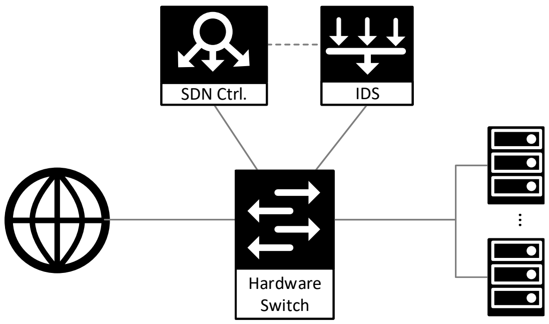
Additionally, the type of switching method used, such as cut-through and store and forward, also introduces a switching delay due to port specific queue behavior. This scenario helps to assert such behaviors precisely. Since no classification occurs, it is not meaningful to compute security metrics. Notably, the true-positive rate and the precision would result in having to divide by zero, since the total number of positive classification in zero. False-positive rate, recall, and accuracy would always yield zero, and the F-measure would not be computable since it relies on the precision value. However, when we evaluate the results, we will simply not state these values since we see little significance in them.



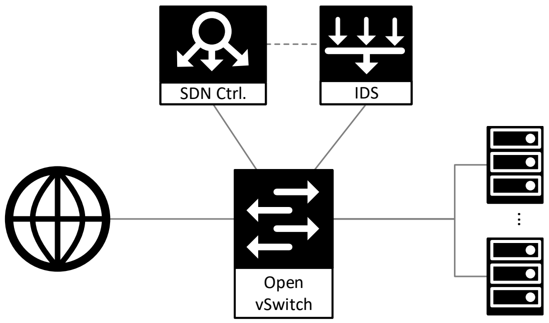
(**a**) Scenario 1a (Unprotected Baseline): Direct connection between source and sink. Baseline for maximum throughput and no security.



(**b**) Scenario 1b (Protected Baseline): NIDS between source and target. Baseline for minimum throughput and maximized security.



(**c**) Scenario 2: SDN-enabled network using a hardware switch.



(**d**) Scenario 3: SDN-enabled network using a software switch.

Figure 4.6: Scenarios used for the Evaluation

Scenario 1b consists of an inline NIDS between the source and the sink. Figure 4.6b shows this configuration. Therefore, all the network traffic is routed via and examined by the NIDS. This scenario helps us generate a reference for the performance of the network with the highest level of security by using a NIDS. Here, some of the primary influencing factors to the bandwidth of the network are the speed of the Ethernet interfaces of the host system as well as the maximum supported throughput of the NIDS that can be achieved based on scaling the NIDS's system resources such as CPU and memory.

Further possible limitations include the I/O performance of the overall system and the operating system used. Thus, the results from this scenario should represent the lowest data throughput and the number of successful HTTP requests combined with the maximum TCP handshake delay and request residence time. Due to the hard-to-predict likeliness of false positives when putting Snort at high load, it is hard to predict the resulting security metrics. However, since Snort tends to false positives rather than false negatives, it is reasonable to expect a small recall value.

Scenario 2: Hardware Switching

This scenario employs an SDN-capable hardware switch in conjunction with an SDN controller. Figure 4.6c gives an overview of this setup. The SDN controller allows us to manipulate the network's flow tables at runtime as for the Adaptive Blacklisting and Whitelisting algorithms described before. The controller also collects feedback from the NIDS's interface to obtain information about the detected attacks. The connection from the SDN Controller to the NIDS is on a separate network. Thus, it does not influence the performance of the user data flow.

This scenario is significant in various ways. Firstly, the type, size, and performance of flow tables vary significantly with different switch models. An evaluation of how the nature of flow tables affects real applications is, therefore, particularly interesting. An OF compatible switch has two types of flow tables: software and hardware tables. While a dedicated processor processes the entries of the hardware table, the CPU takes over the processing for a software table. Software tables are, therefore, significantly slower than the hardware implementations. Even between the hardware tables, there are performance differences in the priority of flows within a table. If multiple tables are present, this fact also adds inter-table prioritization.

Secondly, apart from a pure Quality of Service (QoS) perspective, there are three other performance criteria for flow tables during runtime: Adding, modifying, and deleting existing flows in a table. When managing many flows (more than 100), adding new flows can take a longer time than when there are only a few flows (less than 10). Additional delays can result in further problems, such as the additional triggering of a `packet-in` event for buffered packets.

Scenario 3: Software Switching

In the last experimental setup, a software switch, Open vSwitch, is used instead of a hardware switch that was used in Scenario 2, as shown in Figure 4.6d. In contrast to a hardware switch, a software switch has entirely different behavior. The flow capacities and the performance of the device are no longer dependent on the hardware specifications of the device. Moreover, the separation between the software and hardware tables vanishes and the complete table has consistent performance. Additionally, the size of the table is no longer dependent on the hardware. A more powerful host system leads to faster performance, and the ability to use commodity hardware guarantees flexibility. We chose this scenario to assess the performance of this switch VNF in comparison to the hardware switch.

4.1.3.4 Workloads

We use two different workloads to evaluate the performance and security properties of our applications. The first workload puts the system under a constant load while the other workload targets excessive load.

Workload 1: Constant Load

This workload focuses on evaluating the feasibility of our algorithms. Furthermore, it allows concluding the behavior under constant load.

To achieve this constant load, we query the server with HTTP requests. Exactly 170 requests are open at any time. We have chosen this number to ensure that this load would not exceed the hardware flow table of the employed switch specified at 384 entries. Once the client receives the completion of a request, it starts a new one. This limitation ensures that the load $P(X)$ is constant. Each request consists of an HTTP POST-Request for a two Mebibyte[1] file on an Apache webserver. This size orients itself at the size of an average website [Sta18]. Additionally, to this benign requests, our load generator starts five attacks every ten seconds. The NIDS has signatures configured for these attacks.

[1] 1 Mebibyte = 1024 Kibibyte = 2^{20} Byte = 1 048 576 Bit

Workload 2: Overload

This workload features requests at a rate higher than the maximum capacity of the system. Thus, we create 60 requests per second (value determined after the previous testing) and an attack every two seconds. This behavior ensures that more requests enter the system than leave it. Thus we ensure $a = \frac{\lambda}{\mu} > 1$. The rate of 60 requests per second is about the limit that the service can handle without a NIDS. Due to technical limitations of the used library, it is necessary to cap the limit of simultaneous open requests at 3 000. The other parameters stay the same as for Workload 1.

4.1.3.5 Performance and Security Results

We execute every workload for every scenario for at least 30 times. Every execution takes five minutes. **RQs 4.1b 4.1c 4.1e**

Workload 1: Constant Load

Table 4.2 shows the results for Workload 1. Furthermore, we have computed possible derived metrics (as described in Section 2.1.2.1).

Throughput: Regarding the two baseline scenarios, it becomes evident that Snort in inline mode results in a significant reduction in throughput. Figure 4.7 shows that while Scenario 1a achieves the theoretical maximum of 940 MBit/s [Ryg17], the routing via the NIDS results in a drop to 72 MBit/s or by 92%. Thus, not using the NIDS is 13 times as fast as using the NIDS. Scenario 2 shows that all bypassing algorithms increase the throughput in comparison to the baseline scenario with inline NIDS. While Adaptive Blacklisting and Whitelisting both reach approximately the same results with 434 MBit/s (Blacklisting) and 427 MBit/s (Whitelisting), the Selective Filtering reaches 573 MBit/s. In Scenario 3, the balance changes. Both adaptive approaches reach the theoretical maximum throughput of 940, respectively 941 MBit/s (exceeding the theoretical threshold is related to general inaccuracies when measuring, e.g., imprecise timers). In comparison, Selective Filtering gains only a minimal improvement by 15 MBit/s in throughput to a total of 588 MBit/s.

Successful HTTP Requests: When regarding the successful HTTP requests, the dimensions of the results are similar to the throughput. As seen in Figure 4.8, Scenario 1a again presents the maximum of all scenarios with a total of 17 625 requests. Adding the inline NIDS in Scenario 1b reduces this

Metric	**Scenario**							
	1a	**1b**	**2 (HW Switch)**			**3 (SW Switch)**		
	Baseline		**BL**	**WL**	**Filter**	**BL**	**WL**	**Filter**
HTTP-Requests[1]	17625	1271	9508	9333	10238	16714	16752	10392
Throughput[2] [MBit/s]	-	2.7	10.9	10.8	6.5	14.2	14.5	6.6
Throughput[2] CI	±0.1 [MBit/s]							
Throughput[3] [MBit/s]	940	72	434	427	573	941	940	588
Throughput[3] CI	±1 [MBit/s]							
Residence Time [ms]	2898	37389	5034	4939	4852	3038	3028	4777
TCP Handshake [ms]	20	4657	451	532	2034	167	211	2015
No. Attacks	150	150	153	150	155	155	155	154
No. Snort Alerts	0	223	35	39	154	149	149	156
No. Snort Alerts CI	0	0	±2	±5	±9	±10	±10	±12
True Positives	0	223	35	39	153	149	149	154
False Positives	0	0	0	0	1	0	0	2
False Negatives	0	0	0	0	0	0	0	0
Dupl. Classifactions	-	73	0	0	0	0	0	0
Attacks Lost	-	0	117	111	1	6	6	0
Precision [%]	-	100	100	100	99	100	100	99
Recall [%]	-	100	100	100	100	100	100	100
Accuracy [%]	Can not be computed due to missing total negatives							
F-Measure [%]	-	100	100	100	100	100	100	99

Table 4.2: Measurement results and derived metrics for constant load in Workload 1. Confidence intervals result either from known measurement errors or are calculated using inverse Student's t-distributions with $\alpha = 0.05$.
[1]: Number of successful benign HTTP Requests by the client that were answered by the server.
[2]: Traffic from the Client to the Server
[3]: Traffic from the Server to the Client

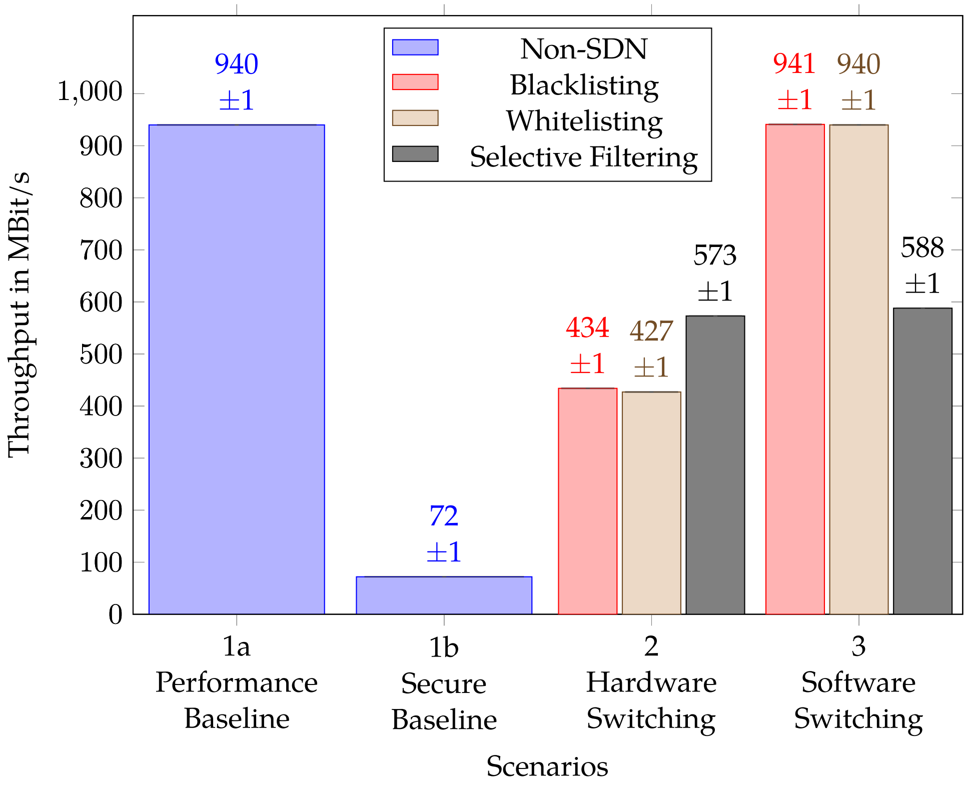


Figure 4.7: Throughput in MBit/s under constant load in Workload 1.

to a mere 1 271 requests. This decrease represents a reduction of 93%. Thus, the number of successful requests in Scenario 1a is 14 times higher than in Scenario 1b. Introducing our bypassing algorithms yields similar results as for the throughput. In Scenario 2, Adaptive Whitelisting can increase the successful requests to 9 333, and Adaptive Blacklisting reaches 9 508 requests. Thus, both adaptive approaches perform within each other's margin of error since the confidence intervals overlap. Again, Selective Filtering outperforms both adaptive approaches in Scenario 2, with a total of 10 238 requests. With the addition of software switching in Scenario 3, both adaptive approaches receive massive performance boosts. Adaptive Whitelisting yields 16 752 and slightly outperforms Adaptive Blacklisting yields 16 714 requests.

Nevertheless, this difference is within the margin of error. With this result, the adaptive approaches get close to the maximum performance baseline established from Scenario 1a. Last, Selective Filtering only gains another 154 requests from the software switch resulting in 10 392 requests.

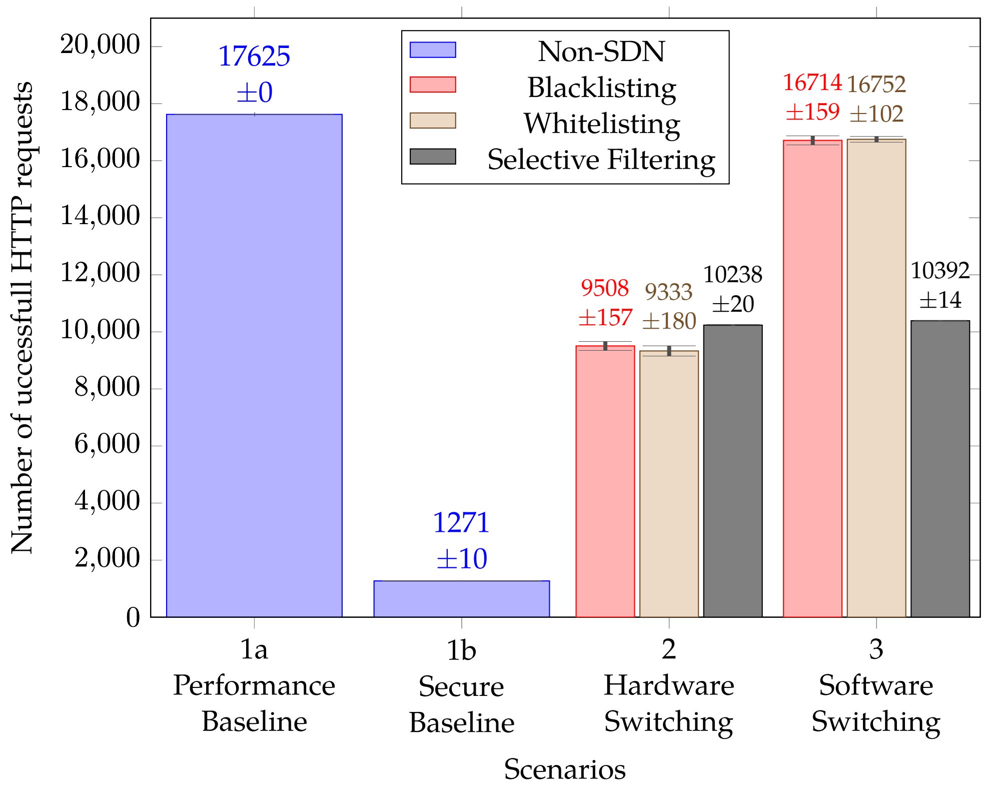


Figure 4.8: Number of successfull requests under constant load in Workload 1.

TCP Handshake Duration: Figure 4.9 shows the effect of the various scenarios on the TCP handshake duration. With a direct connection in Scenario 1a, the handshake only takes 20 ms. The addition of the inline NIDS increases the delay to 4 657 ms in Scenario 1b. In Scenario 2, we see a significant reduction in the delay for all bypassing approaches. Unlike the throughput and the number of successful HTTP requests, dynamic approaches achieve a smaller delay at 451 ms (Blacklisting), respectively, 532 ms (Whitelisting) than the Selective Filtering at 2 034 ms. Another difference to the throughput and the number of successful HTTP requests the results for the adaptive approaches no longer lie within each other's margin of error. Selective Filtering again only marginally improves its performance by 19 ms upon adding the software switch in Scenario 3 to 2 015 ms. At the same time, the dynamic approaches further improve more than halving their delay to 167 ms, respectively, to 211 ms.

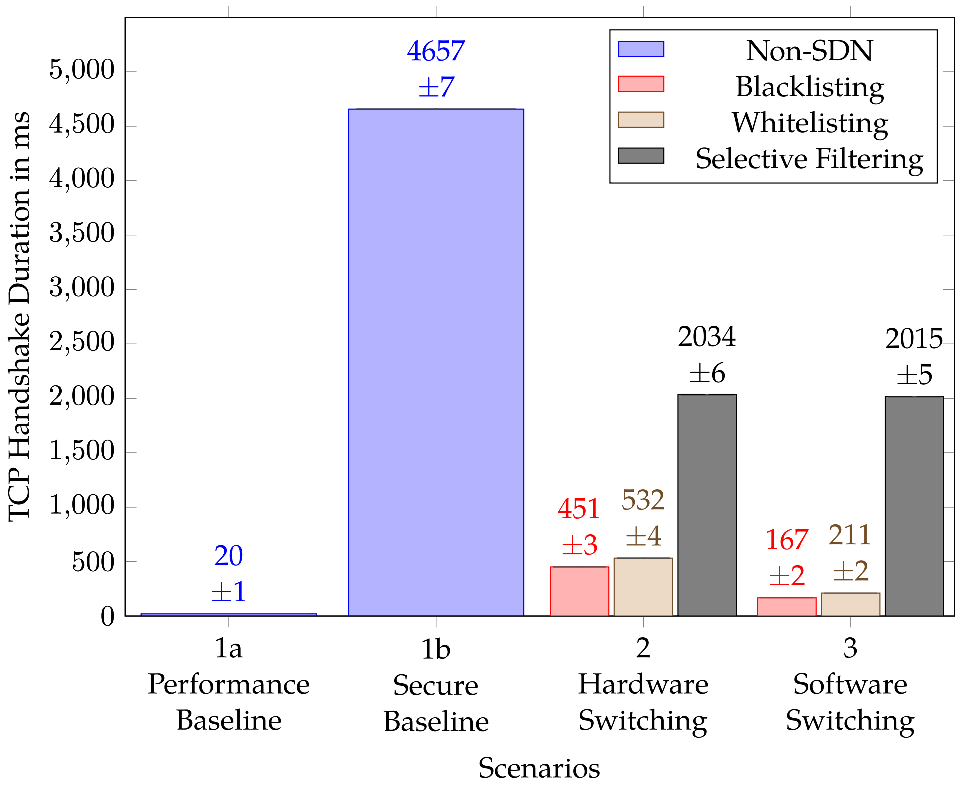


Figure 4.9: TCP handshake durations in milliseconds under constant load in Workload 1.

Residence Time: Regarding the residence time, we receive results comparable to the TCP handshake duration. Since the handshake is part of the residence time, this behavior is as expected. Section 4.1.3.2 shows the results for the residence time. The handling of the request takes 2 898 ms in Scenario 1a without a NIDS. Adding the NIDS in Scenario 1b increases the residence time to 37 389 ms. This increase equals a factor of thirteen (which is equal to the reduction in throughput and close to the reduction of handled HTTP requests). The use of the adaptive SDN-based algorithms in Scenario 2 significantly reduces the residence time by 1 996 ms to 5 034 ms (Adaptive Blacklisting) and by 1 911 ms to 4 939 ms (Adaptive Whitelisting). Selective Filtering only performs slightly better, and its results of 4 852 ms are within the margin of error of Adaptive Whitelisting. When using a software switch in Scenario 3, the residence time further decreases for both adaptive algorithms reaching values of 3 038 ms (Adaptive Blacklisting) and 3 028 ms (Adaptive Whitelisting). These results

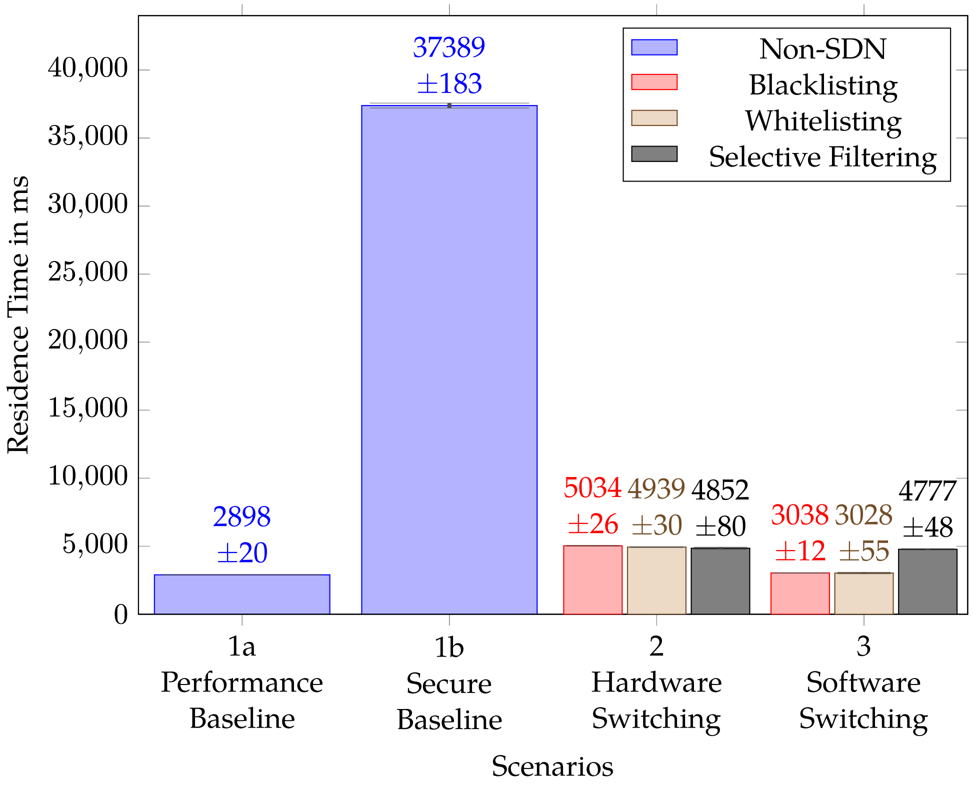


Figure 4.10: Residence time in milliseconds under constant load in Workload 1.

are within each other's margin of error and close to the performance baseline. Adaptive Whitelisting in Scenario 3 takes only 4% longer than Scenario 1a to complete a request. Similar to the previous results, Selective Filtering hardly profits from the introduction of the software switch and reaches 4 777 ms – a reduction by 75 ms.

Attack Detection Ratio: Figure 4.11 shows the ratio between the number of detected attacks and the number of executed attacks. Apparently, without a NIDS, Scenario 1a detected no attacks. Scenario 1b adds the NIDS and already shows around 149%. So, more attacks are detected than are executed. Scenario 3 shows a significant drop in the detection rate for the two dynamic approaches. Adaptive Blacklisting achieves only a 23% detection rate, and Whitelisting is only slightly better at 26%. Selective Filtering achieves a detection rate of 99%, which is within the margin of error to the ideal 100% rate. Using the software switch in Scenario 5 increases the ratio to 96% for both dynamic approaches

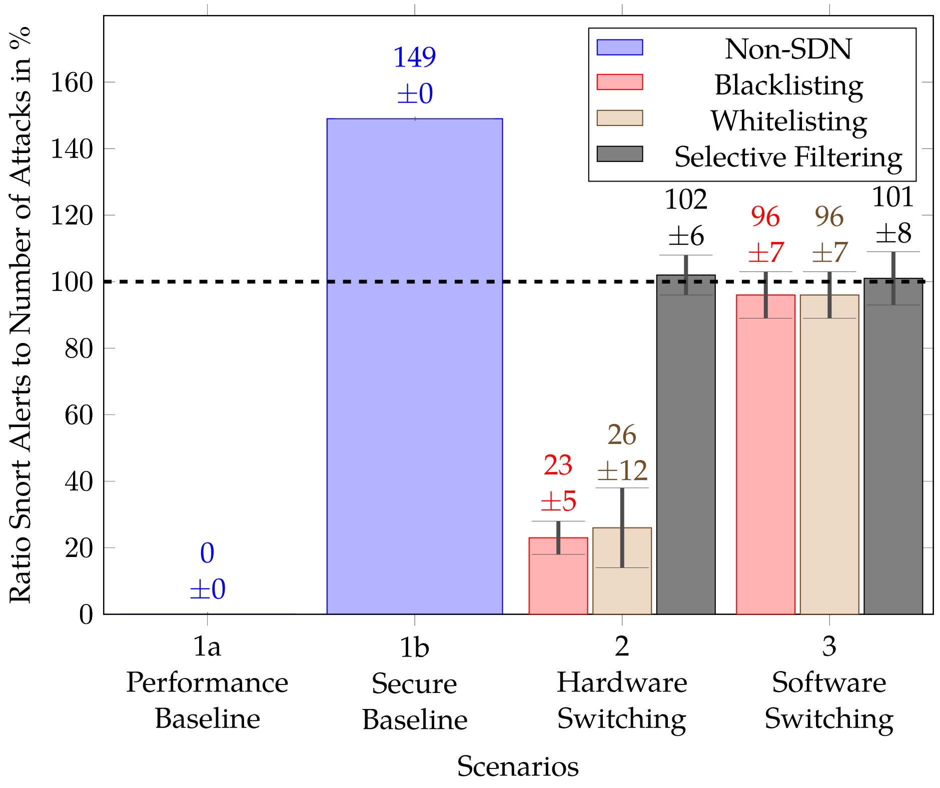


Figure 4.11: Number of detected attacks relative to the expected number under constant load in Workload 1.

and 101% for Selective Filtering. All three approaches are within the margin of error of 100%. When examining the attacks in Snort's database, they all contain the sequences that should trigger the signatures except for one packet for Selective Filtering in Scenario 2 and two packets for Selective Filtering in Scenario 3. Thus, all attacks appear to be correctly detected, but false positives occur.

Security Metrics: Next, we take a look at the security metrics computed from the measured values. Despite deviations in the number of detected attacks, all approaches and all scenarios (except of course Scenario 1a, which does not perform classification) yield a perfect recall. The precision for the secure baseline and the adaptive approaches also reaches the maximum value. The Selective Filtering has a very high, yet not perfect, precision. This issue also results in a reduced F-Measure, at least for Scenario 3. Since the adaptive approaches achieve maximum precision and recall, they also realize the maximum value for the F-measure.

Workload 2: Overload

RQ 4.1d Table 4.3 shows the results for Workload 2. Furthermore, we have computed possible derived metrics (as described in Section 2.1.2.1). In the following, we describe these results and the differences to Workload 1.

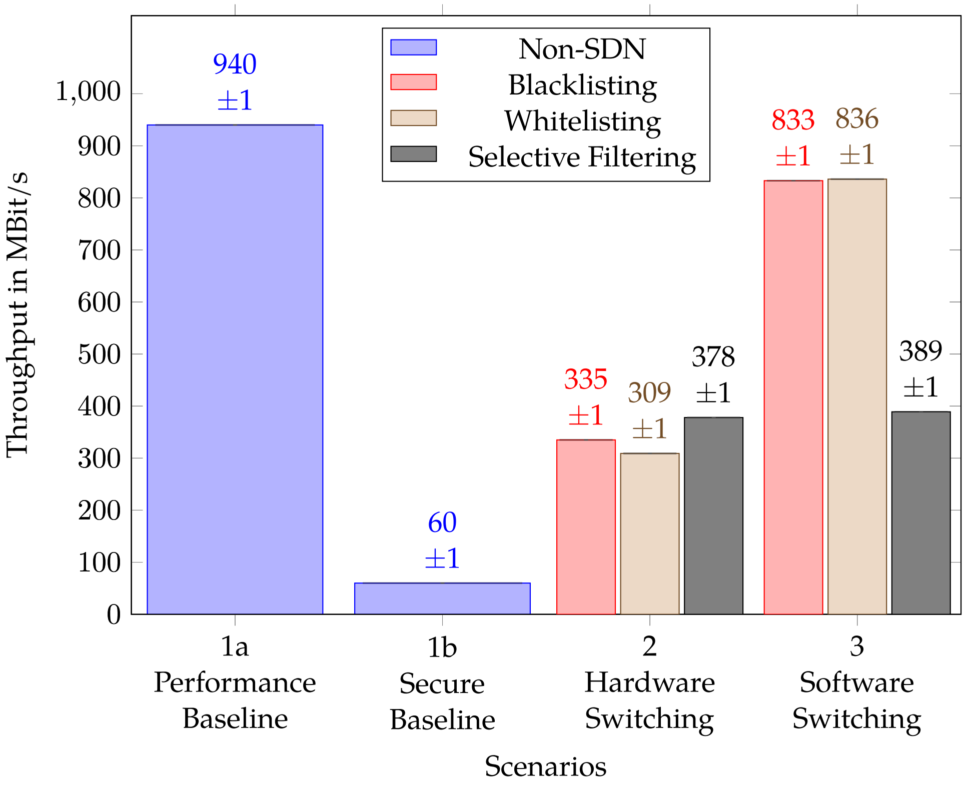


Figure 4.12: Throughput in MBit/s under overload in Workload 2.

Throughput: Figure 4.12 shows the throughput when the system is in an overload situation. Throughput in Scenario 1a remains the same as the theoretical maximum of 940 MBit/s. In Scenario 1b, throughput drops down to 60 MBit/s, which is 12 MBit/s below the result from Workload 1. This reduction represents a drop by 94%, making the throughput in Scenario 1a 16 times higher than in Scenario 1b. For Scenario 2, adding the SDN-based algorithms again

Metric	**Scenario**							
	1a	**1b**	**2 (HW Switch)**			**3 (SW Switch)**		
	Baseline		**BL**	**WL**	**Filter**	**BL**	**WL**	**Filter**
HTTP-Requests[1]	18000	7014	14649	13831	11564	18051	18049	11704
completed on time[1]	17880	1271	9432	8626	7943	16699	16597	8154
Throughput[2] [MBit/s]	-	2.6	6.7	6.0	5.8	12.19	12.6	5.8
Throughput[2] CI	± 0.1 [MBit/s]							
Throughput[3] [MBit/s]	940	60	335	309	378	833	836	389
Throughput[3] CI	± 1 [MBit/s]							
Residence Time [ms]	1021	106781	23638	24644	39800	9868	11508	39216
TCP Handshake [ms]	208	65457	21118	21908	33466	6961	8720	33032
TCP Handshake / Residence Time [%]	20	61	89	89	84	71	76	84
No. Attacks	150	22	59	57	44	147	140	47
No. Snort Alerts	0	48	29	31	72	131	130	79
No. Snort Alerts CI	0	0	± 9	± 8	± 68	± 8	± 16	± 61
True Positives	0	48	29	31	52	128	128	57
False Positives	0	0	0	0	20	3	2	22
False Negatives	0	0	0	0	0	0	0	0
Dupl. Classifactions	-	26	0	0	8	0	0	10
Attacks Lost	-	0	30	28	0	16	10	0
Precision [%]	-	100	100	100	72	98	98	72
Recall [%]	-	100	100	100	100	100	100	100
Accuracy [%]	Can not be computed due to missing total negatives							
F-Measure [%]	-	100	100	100	84	99	99	84

Table 4.3: Measurement results and derived metrics for overload in Workload 2. Confidence intervals result either from known measurement errors or are calculated using inverse Student's t-distributions with $\alpha = 0.05$.
[1]: Number of benign HTTP Requests sent by the client.
[2]: Traffic from the Client to the Server
[3]: Traffic from the Server to the Client

increases the throughput. With 335 MBit/s, Adaptive Blacklisting outperforms Adaptive Whitelisting, which increases the throughput to 309 MBit/s. Again Selective Filtering outperforms both adaptive approaches in Scenario 2 with 378 MBit/s. However, the advantage is smaller than for Workload 1. Also, all approaches perform worse than in Workload 1 with throughput values reduced by 99 MBit/s (Adaptive Blacklisting), 125 MBit/s (Adaptive Whitelisting), and 195 MBit/s (Selective Filtering). For Scenario 3, again, the adaptive algorithms profit most. Both Adaptive Blacklisting and Adaptive Whitelisting provide very close results with 833 MBit/s and 836 MBit/s. These results equal an increase of 498 MBit/s (Adaptive Blacklisting) and 527 MBit/s. Like for Workload 1, Selective Filtering only takes a small profit – here 20 MBit/s – from the addition of the software switch. Similar to Scenario 2, the results are lower than for Workload 1 by 108 MBit/s (Adaptive Blacklisting), 104 MBit/s (Adaptive Whitelisting), and 199 MBit/s (Selective Filtering).

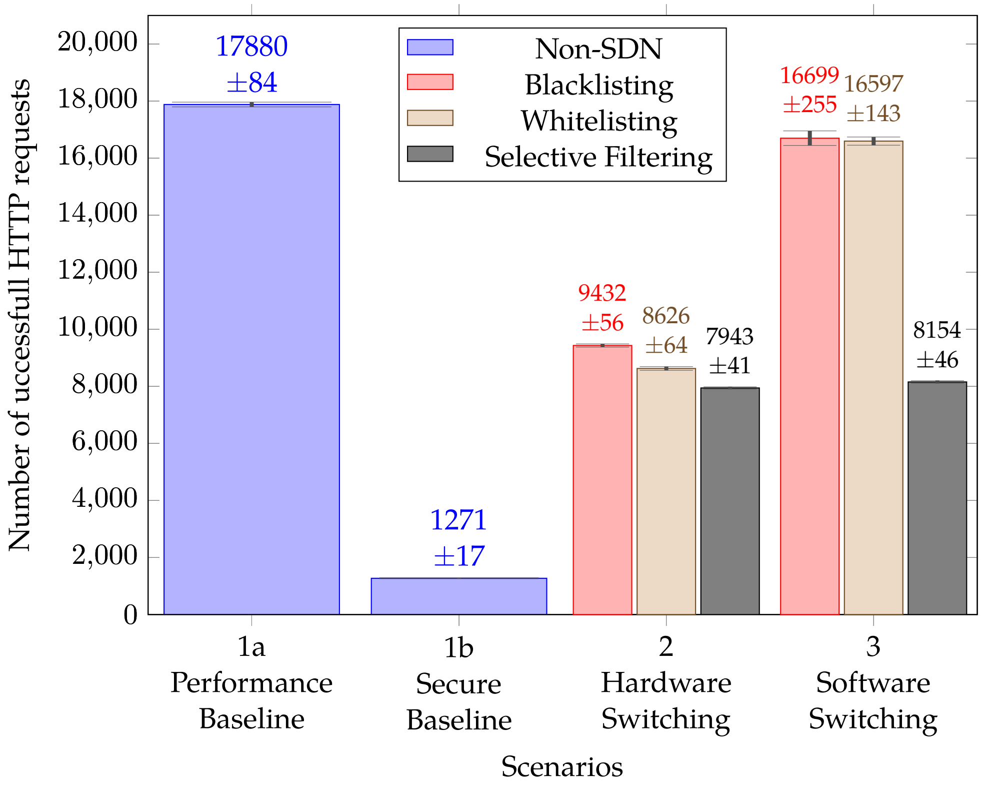


Figure 4.13: Number of successfull requests under overload in Workload 2.

Successful HTTP Requests: Figure 4.13 shows that the behavior of the successful request metric is similar to the behavior of the throughput. This relation also corresponds to a similar observation from Workload 1. Putting the system in an overload situation increases the number of requests for the performance baseline in Scenario 1a slightly by 155 to 17 880. Possible causes for this increase are short-time performance surges by, e.g., CPU frequency boosts. When adding the NIDS in Scenario 1b, the number of successful requests stays the same as for Workload 1 at 1 271. The addition of the SDN-based algorithms in Scenario 2 increases the number compared to the secure baseline. Adaptive Blacklisting reaches 9 432 and Adaptive Whitelisting 8 626 requests. This reduction is a decrease compared to Workload 1 of 76 (Adaptive Blacklisting) and 707 (Adaptive Whitelisting) requests. Unlike for the throughput in both workloads and the number of successful requests in Workload 1, Selective Filtering falls behind the adaptive approaches with only 7 943 successful requests – a reduction by 2 281 compared to Workload 1. Again, the adaptive approaches take a substantial profit from the addition of the software switch in Scenario 3. Adaptive Blacklisting reaches 16 699 and Adaptive Whitelisting 16 597 requests. These are only a small decrease of 15 (Adaptive Blacklisting), and 155 (Adaptive Whitelisting) requests compared to Workload 1. For Adaptive Blacklisting, this is even within the margin of error and equals to 93% of the performance baseline. Once more, Selective Filtering only marginally profits from the software switch gaining another 211 requests to a total of 8 154.

TCP Handshake Duration: Figure 4.14 presents the results for the TCP handshake duration under overload. In general, all values increase compared to Workload 1. Still, the performance baseline from Scenario 1a yields the lowest duration at 208 ms (still tenfold the result from Workload 1). In Scenario 1b, introducing the NIDS increases the average handshake duration to 65 457 ms – over a minute. The usage of SDN in Scenario 2 decreases these values to 21 118 ms for Adaptive Blacklisting and 21 908 ms for Adaptive Whitelisting. These results are a massive increase by 20 667 ms for Adaptive Blacklisting and 21 376 ms for Adaptive Whitelisting compared to the results from Workload 1. Like in Workload 1, Selective Filtering falls short of the adaptive approaches in this metric with 33 466 ms - an increase of 31 432 ms. As before, the adaptive algorithms profit from the software switch added in Scenario 3. This addition takes off another 14 157 ms for Adaptive Blacklisting resulting in 6 961 ms and 13 188 ms for Adaptive Whitelisting, resulting in 8 720 ms. While these results are still a massive increase from Scenario 1a, the adaptive algorithms reduce the handshake duration by up to 89% compared to Scenario 1b. Selective Filtering

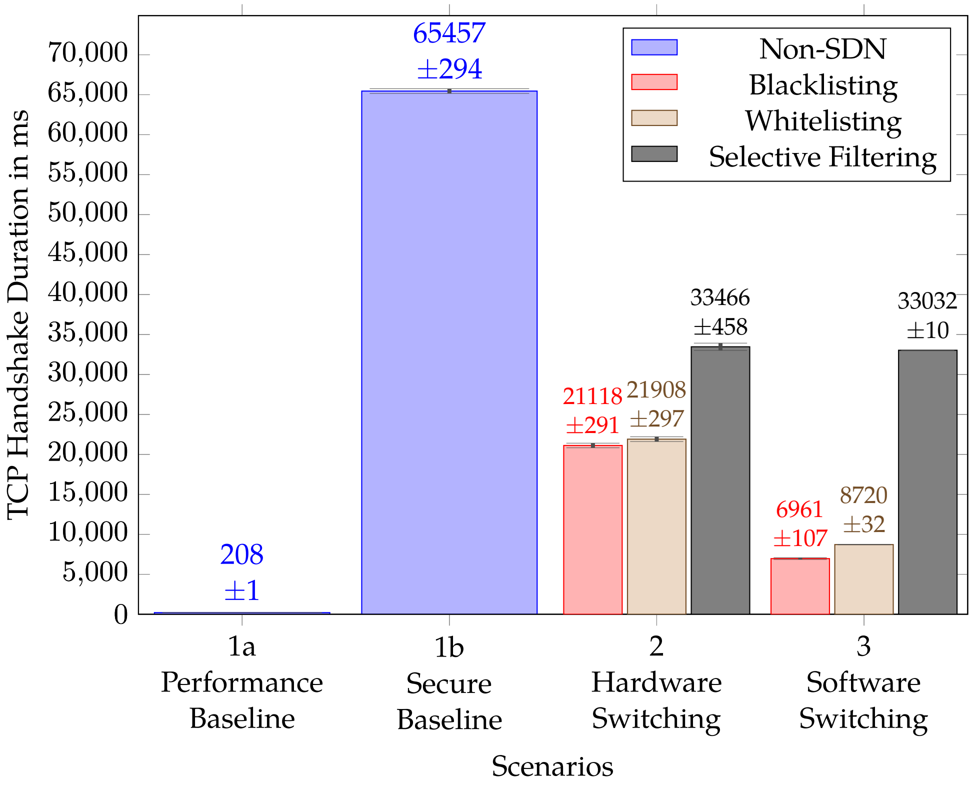


Figure 4.14: TCP handshake durations in milliseconds under overload in Workload 2.

only receives a small boost of 434 ms from the software switch, which is even within the margin of error from Scenario 2.

Residence Time: As mentioned for Workload 1, the residence time comprises the TCP handshake duration and therefore exceeds it. Figure 4.15 shows the measurement results for the residence time under Workload 2. As before, the performance baseline in Scenario 1a gives the best result as desired with 1 021 ms. This result is 1 877 ms lower than for Workload 1. Possible causes for this behavior might be short-time CPU frequency boosts or batch-processing. The introduction of the NIDS in Scenario 1b results in an inflation of the residence time to 106 781 ms – 69 392 ms more than in Workload 1. Once more, in Scenario 2, the SDN-based approaches significantly reduce the residence time. Adaptive Blacklisting yields 23 638 ms, and Adaptive Whitelisting achieves 24 644 ms. These results are an increase of 18 604 ms for Adaptive Blacklisting

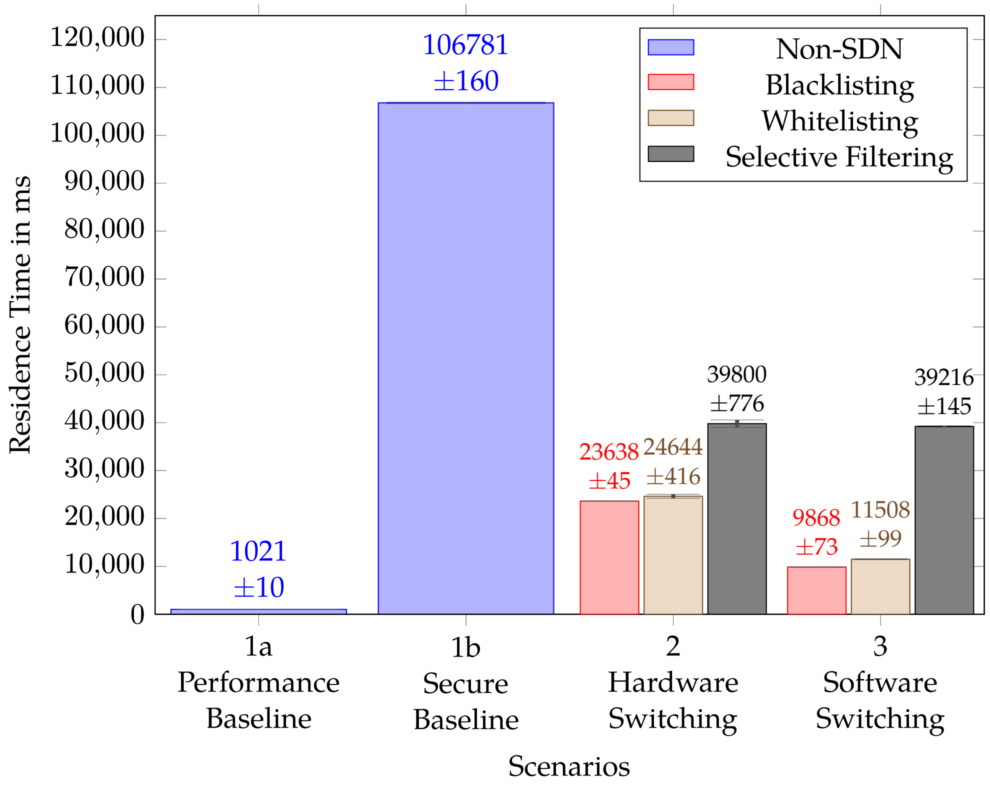


Figure 4.15: Residence time in milliseconds under overload in Workload 2.

and 19 705 ms for Adaptive Whitelisting compared to the results from Workload 1. Selective Filtering results in an average residence time of 39 800 ms – an increase over Workload 1 by 34 948 ms. The software switch again benefits both adaptive algorithms reducing their residence time by 13 770 ms to 9 868 ms for Adaptive Blacklisting and by 13 136 ms to 11 508 ms for Adaptive Whitelisting. This improvement is still an increase of 6 830 ms for Adaptive Blacklisting and 8 480 ms for Adaptive Whitelisting. However, Adaptive Blacklisting can reduce the residence time to 9% of the residence time for the secure baseline from Scenario 1b. It is also worth mentioning that in scenarios using the NIDS, the share of the TCP handshake duration on the total residence time is significant, with 61% to 89%.

Attack Detection Ratio: Figure 4.16 shows the results for the attack detection ratio. In Scenario 1a, no classification occurs. The inline NIDS in Sce-

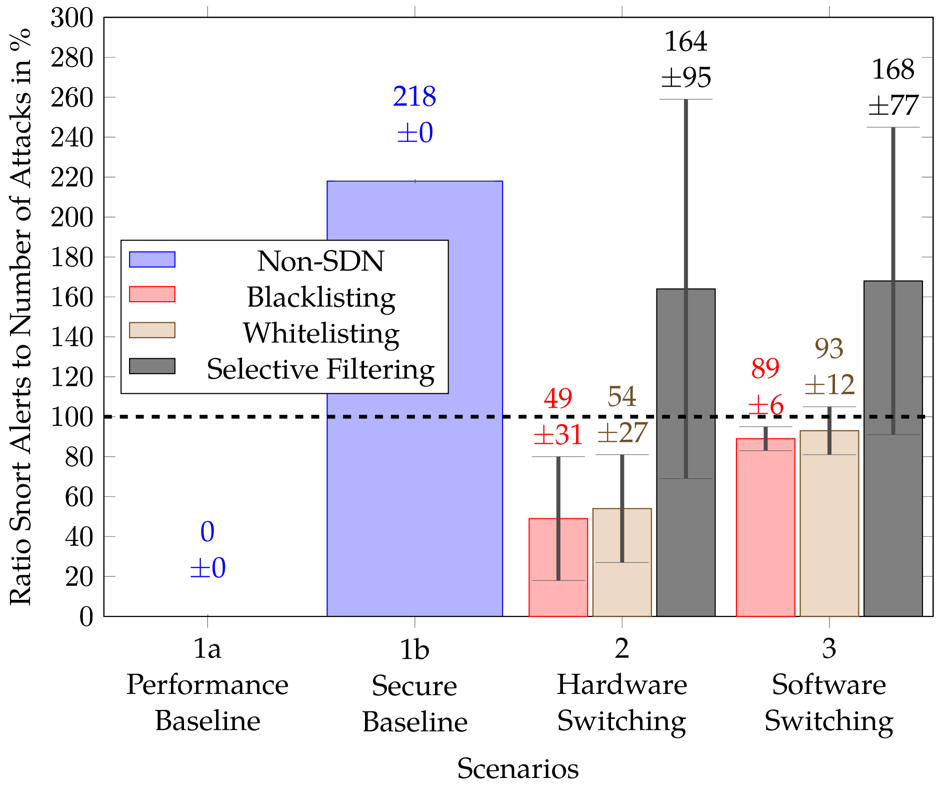


Figure 4.16: Number of detected attacks relative to the expected number under overload in Workload 2.

nario 1b is overloaded and yields 218% or more than twice as much attack detections than sent attacks. However, these detections are duplicate classifications. Using the SDN-based algorithms in Scenario 2 yields 49% for Adaptive Blacklisting and 54% for Adaptive Whitelisting of attacks detected. This missed target is mainly due to many attacks – 51% for Adaptive Blacklisting and 49% for Adaptive Whitelisting – being lost in the network. However, this is unexpectedly an improvement of 26% for Adaptive Blacklisting and 28% for Adaptive Whitelisting compared to Workload 1. Selective Filtering yields a 164% attack detection rate. This extra 64% contains 45% of false positives and 19% of duplicate classifications. However, these results are highly volatile, reducing their meaningfulness. Using the software switch in Scenario 3 boosts the detection rate of Adaptive Blacklisting by 40% to 89% and of Adaptive Whitelisting by 39% to 93%. Adaptive Whitelisting is thereby within the margin of error of the goal of 100%. Adaptive Blacklisting accounts for 2% false positives, and 11% lost attacks, and Adaptive Whitelisting has 1% false positives, and 7% lost attacks. Selective Filtering this time can not only not profit from the software switch but even has a higher surplus in attack detection by an additional 4% summing up to 168%. These 68% surplus comprise 47% of false positives and 21% of duplicate classifications.

Security Metrics: Last, we take a look at the security metrics computed from the measured values. Despite deviations in the number of detected attacks, all approaches and all scenarios (except of course Scenario 1a, which does not perform classification) yield a perfect recall. The precision for the secure baseline, and the adaptive approaches with the hardware switch also reaches the maximum value. With the software switch, these approaches suffer a slight loss of 2% in precision. The Selective Filtering has a mediocre, precision of 72% for both scenarios. This shortcoming also results in a reduced F-Measure of 84%. Since the adaptive approaches achieve maximum precision and recall when using the hardware switch, they also realize the maximum value for the F-measure. The reduced precision also affects the F-measure, reducing it to 99%, when using the software switch. Due to the close results – except for the Selective Filtering – a figure for the security metrics was omitted.

4.1.4 Discussion

Criterion	**Approach & Workload**					
	Dynamic Blacklisting		**Dynamic Whitelisting**		**Selective Filtering**	
	WL 1	**WL 2**	**WL 1**	**WL 2**	**WL 1**	**WL 2**
Hardware Switch						
Improved Performance	✓	✓	✓	✓	✓	✓
Metrics $> 95\%$	✓	✓	✓	✓	✓	✗
Loss Rate $< 10\%$	✗	✗	✗	✗	✓	✓
Software Switch						
Improved Performance	✓	✓	✓	✓	✓	✓
Metrics $> 95\%$	✓	✓	✓	✓	✓	✗
Loss Rate $< 10\%$	✓	✗	✓	✓	✓	✓

Table 4.4: Overview of the results for the used approaches and scenarios.

The comparison between Scenario 1a and 1b confirms the motivation for this paper. Adding an inline NIDS reduces the throughput dramatically and increases the latency. Furthermore, an inline NIDS under high load triggers more alerts than actual attacks occurred.

Table 4.4 gives an overview of the achieved goals of the NIDS-bypassing approaches and their fulfillment. As expected, bypassing the NIDS increases the performance. However, the two dynamic approaches behave differently than Selective Filtering. While in Scenario 2, they increase the performance relative to the inline NIDS, they decrease the rate of detected attacks to about a fourth of the actual attacks. These are unacceptable security characteristics.

When replacing the hardware switch with a software switch in Scenario 3, the performance of the adaptive approaches increases even further to the theoretical throughput maximum and a latency acceptable for a web server. The attack detection ratio increases as well. The detection of all executed attacks is within the margin of error, and precision, recall, and F-Measure remain at 99% to 100%.

On further investigation of this behavior, it seems that in some conditions, the hardware switch starts using the software table. This behavior is unexpected since we chose the number of active connections below the hardware table capacity. The switch becomes significantly slower and frequently does not react to requests from the controller when using the software table. This characteristic could explain both the performance and detection impact. When the installation of the rerouting flows fails or executes incorrectly, traffic could either be permanently sent via the NIDS (reduced performance) or directly to the service host (no attack detection). Packets for unhandled packet-in events (e.g., because no flow is present and the controller can not add it due to the switch not responding) can also become lost in the network.

The results for the Selective Filtering show that bypassing with simple rules can yield an increased performance compared to inline mode. Especially without overload, Selective Filtering presents good results and can be a simple solution for small performance issues.

Workload 2 proves that the effect of Snort on performance under overload is even higher than under normal circumstances. The number of surplus attack detections also increases.

Again, the dynamic approaches prove their utility by increasing throughput and reducing latency. In Scenario 2, the detection rate becomes unreliable. Like in Workload 1, the addition of the virtual switch further increases the detection rate and the performance.

Again in Workload 2, Selective Filtering improves throughput and latency.

It also reduces the surplus detections on average, but the high deviation reduces the reliability of this observation. The dynamic approaches retain high precision, accuracy, and F-Measure.

In summary, the two bypassing algorithms show promising results. They can improve performance up to an extent where the introduction of the NIDS has no impact on performance. Attack detection is within the margin of error to 100% under typical load for both algorithms and under overload for Adaptive Whitelisting. For the constant load workload, all algorithms, and in the overload workload, the dynamic algorithms present excellent F-Measure, precision, and recall metrics. The successful use of both dynamic algorithms depends on the use of the software switch since their reliability and performance suffer when using the hardware switch. When using the software switch, the Adaptive Whitelisting fulfills all requirements.

A limitation is that our framework at the moment allows no direct tracking of what happens at the hardware switch when it becomes unresponsive in Scenario 2. Any assertion which flows get redirected and which do not will require this functionality.

4.2 TCP Handshake Remote Establishment and Dynamic Rerouting using Software-defined Networking

In Section 2.2.1.1, we introduced the threat of the SYN flood attack. To counter this attack in Section 2.3.2.1 and Section 2.3.2.2, we detailed the existing solutions SYN Cookies and SYNPROXY as state-of-the-art DPSs. However, as mentioned, both systems have significant shortcomings. SYN Cookies, on the one hand, require co-location with the service and can not be scaled independently from the service. SYNPROXY, on the other hand, is stateful and must forward all traffic after connection establishment.

To alleviate the issues mentioned above, we present THREADS (TCP Handshake Remote Establishment and Dynamic Rerouting using Software-defined Networking), a novel SYN flood mitigation solution using SDN. In the following, we specify the requirements that lead to our design decisions and describe our approach.

THREADS needs to address the shortcomings of SYN cookies and SYNPROXY specified above. Thus, THREADS needs to be:

(1) stateless,

(2) independently deployable

(3) scalable, and

(4) must not require traffic to make a detour after establishing a connection.

4.2.1 Approach

RQs 4.2a 4.2b THREADS is a stateless service, hosted on a dedicated machine, to realize a solution fitting these requirements. Packets have to directly reach this machine during connection establishment, while all traffic of established connections proceeds to the server without passing through the dedicated THREADS machine.

4.2.1.1 Architecture

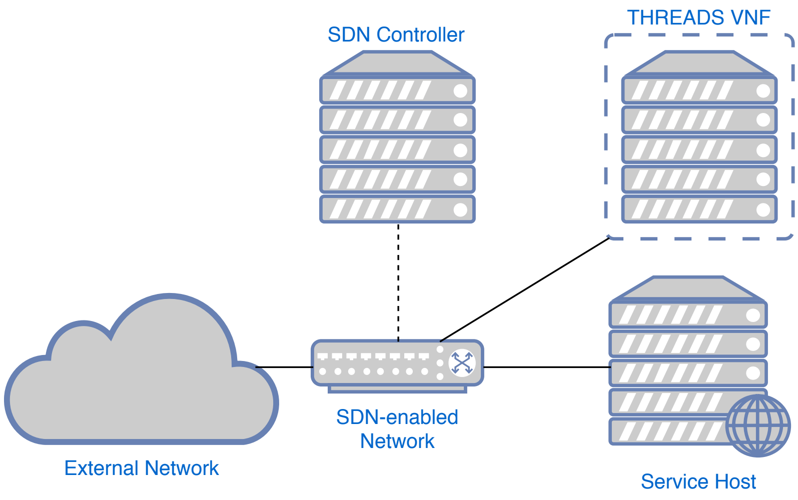


Figure 4.17: Minimal network architecture of THREADS.

Our approach takes advantage of the functionality to program flows depending on their source and target provided by SDN. THREADS deploys inside an SDN-enabled network as a two-part solution, consisting of a VNF hosted on a separate machine as well as a kernel module loaded by the machine hosting the protected service. Figure 4.17 shows the network architecture for THREADS deployment.

Virtualized Network Function

A VNF separated from the service host handles the entire connection establishment process and informs the service host after completing a handshake. Hence, the VNF shields the service host from potentially malicious traffic, while forwarding legitimate connections after their successful establishment.

Kernel Module at the Service Host

Our kernel module augments the service host with additional capabilities. The provided extension of the network stack allows the service host to take over legitimate connections established by the VNF. Furthermore, the service host is capable of triggering network reconfigurations through the SDN controller, redirecting packets belonging to established connections, so that packets on these connections reach their target directly without detouring via the VNF host. At first, we used the VNF to communicate with the SDN controller. However, the solution using the service host proved more efficient during attack scenarios.

4.2.1.2 Connection Establishment

In the freshly deployed configuration, the data plane starts with three preconfigured rules:

A) Forwards all traffic from external networks is to the VNF.

B) Forwards all traffic from the VNF to the gateway to the external network.

C) Sends all traffic from the VNF addressed to the service host to the service host.

Thereby, the last rule has a higher priority than the other rules, overriding the general rule for all traffic from the VNF.

Connection establishment follows the procedure described below:

(1) The client sends a SYN packet to the server.

(2) The data plane diverts the packet to the VNF (Rule A).

(3) The VNF generates a SYN+ACK packet using the same algorithm as used in SYN cookies and sends this packet back to the client.

(4) The network forwards the packet to the external network (Rule B).

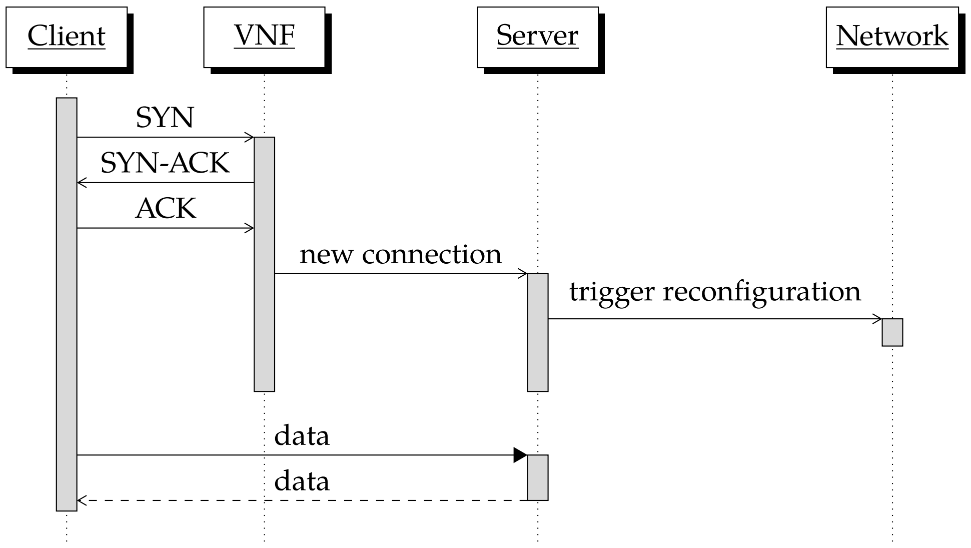


Figure 4.18: Connection Establishment Process Using THREADS

   a) If the packet was part of a SYN flood attack, the establishment process ends here since either the client IP will be spoofed or the attacking client will not respond.
   b) Else the packet reaches the client.

(5) The client replies with an ACK packet.

(6) The network again diverts the packet to the VNF (Rule A).

(7) The VNF validates the ACK packet. From the client's perspective, this step concludes the connection establishment.

(8) The VNF sends a so-called SYN+ packet to the server. This packet contains the negotiated values for sequence and acknowledgment numbers, as well as the other required parameters that define the connection.

(9) The network forwards this packet to the server (Rule C).

(10) The server accepts the packet and starts listening on the specified connection. Furthermore, it notifies the SDN controller to create two new rules:

   D) The switch sends all traffic from the client to the server on the negotiated connection directly to the server.

E) The switch sends all traffic from the server to the client on the negotiated connection directly to the client.

Rule D requires a higher priority than Rule A. THREADS sets the priority for Rules D and E to the same value for simplification. Both rules have a soft timeout. If no packets transit the connection for a preset amount of time, the rules are deleted and would require a new connection establishment process. An alternative is to delete the rule once the switch detects a TCP connection termination. This approach is an alternative for long-living connection with variable data transmissions like SSH sessions.

(11) The client and server can now exchange data directly without involving the VNF.

Figure 4.18 gives an overview of the connection establishment process.

4.2.1.3 Requirement Fulfillment

Requirement	SYN cookies	SYNPROXY	THREADS
Stateless	✓	✗	✓
Independently Deployable	✗	✓	✓
Scalable	✗	✓	✓
Direct Routing	✓	✗	✓

Table 4.5: Requirement fulfillment of THREADS.

Using this approach has many advantages compared to existing solutions and thus fulfills the previously stated requirements. Separation of the service host and the protection is possible (req. 2), similar to SYNPROXY, but established connections no longer have to pass through the added machine (req. 4) and instead continue directly to the server. Thus, it eliminates the proxy as a possible network bottleneck and freeing CPU resources at this machine to handle attacks. The VNF itself is stateless (req. 1). Combined with the separation of the service and the protective environment, this property allows adding VNFs if required. This goal only requires simple (preferably SDN-based, e.g., [GK17; GC18; QW15]) load balancing solution. Every THREADS VNF can verify an ACK packet that is a response to a SYN+ACK packet from itself or other THREADS VNFs. Thus, the protection system becomes independently scalable (req 3.) without the need to modify the server application, as would be the case with SYN cookies. Table 4.5 gives an overview of the requirement fulfillment by THREADS.

4.2.2 Implementation

After detailing the concept of our approach, this section covers the implementation of the two essential components of THREADS. We first introduce the VNF implementation and then present the kernel module deployed on the service host.

4.2.2.1 Virtualized Network Function

We implemented the VNF in the C language using DPDK. The Data Plane Development Kit (DPDK) is an open-source project designed to accelerate software packet processing by bypassing the network stack provided by the Linux kernel. It contains a set of network interface controller drivers as well as data plane libraries. DPDK creates an Environment Abstraction Layer (EAL) hiding environment specifics. To eliminate interrupt overhead, DPDK accesses devices using polling. Therefore, DPDK uses run-to-completion scheduling. The choice for DPDK in comparison to similar solutions fell due to our previous experience with DPDK, its excellent performance, and its wide adoption. We presented more information about DPDK in Section 2.5.3.2.

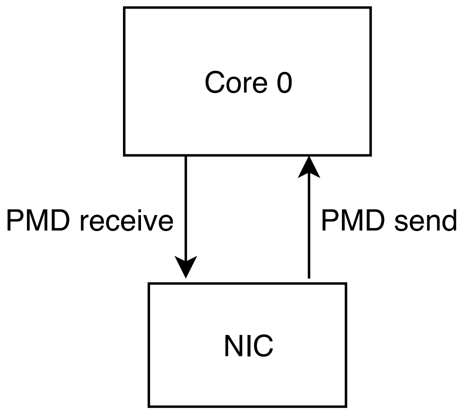


Figure 4.19: General architecture for the THREADS VNF.

For the proof-of-concept implementation, we use a simple approach, as seen in Figure 4.19. The THREADS VNF uses a Poll Mode Driver (PMD) as it allows for direct access to the configuration of the network devices as well as their receive and send queues. THREADS receives the packets using the PMD receive API provided by DPDK and processes received packet sequentially. Finally, THREADS sends the pending outgoing packets using the PMD transmit API.

When processing SYN packets, the THREADS VNF replies with a matching SYN+ACK packet. To generate this SYN+ACK packet, THREADS relies on the same approach used by SYN cookies. On processing ACK packets, the VNF validates whether the packet matches a previously sent SYN+ACK packet. If this is the case, THREADS sends a SYN+ packet (as described above) with the required information encoded to the service host.

4.2.2.2 Service Host Kernel Module

The goal of the kernel modification is to allow to remotely open TCP connections for the server with the possibility to tell the kernel which ISN it has to use. Therefore the kernel module replaces the `tcp_v4_init_seq` kernel function with a modified version. If the function processes a regular SYN packet, the behavior stays the same as for the non-modified version. Thus, it generates a random ISN and sends out a SYN+ACK packet. When receiving a SYN+ packet, the function decodes the ISN and other parameters from this packet. Next, the modification opens the matching socket and starts listening for data.

Additionally, the module handles the notification of the SDN controller. Therefore, once a SYN+ packet is received, the module opens a UDP socket and sends the required message to the SDN controller.

The choice fell on a kernel module to minimize performance overhead, and because regular connection establishment also occurs in kernel space. UDP offers the ability to trigger a faster reconfiguration of the network. If connection establishment duration is less critical in some use-case that focuses more on reliability, TCP can replace UDP. Another alternative would be the controller's REST interface. However, communication with this interface is not only slower but also more complex and would, in most cases, require to exit the kernel space.

4.2.3 Evaluation

To validate our approach, we perform experiments to assert that (1) connection establishment works for our approach, and (2) our approach can handle SYN flood attacks. To this end, we perform measurements in a dedicated testbed and present the results of these experiments. Finally, we provide a discussion of the obtained results.

Unit	**Value**
Product	Dell PowerEdge R210 II
CPU	Intel Xeon E3-1230 v2
Default CPU frequency	3.30 GHz
Max CPU frequency	3.70 GHz
Min CPU frequency	1.60 GHz
Cores (Threads)	4 (8)
Cache (L1/L2/L3)	64 KB/256 KB/8192 KB
Memory size	16GB (2 x 8 GB) DDR4
Memory frequency	1.600 GHz
Memory Connection	Dual Channel
Storage Model	HGST Ultrastar A7 K2000 500 GB@7200 rpm
Storage Connection	SATA II (3GBit/s)
1st NIC (Controller & Backend)	2 Port Broadcom Limited NetXtreme II BCM5716 Gigabit Ethernet
2nd NIC (Experiments)	Intel X520 10-Gigabit SFI/SFP+ Network Connection
Operating System	Ubuntu 18.04.3 LTS (x86-64)
Kernel	4.4.0-72

Table 4.6: Hardware specifications of all servers inside the testbed except the VNF.

4.2.3.1 Testbed Description

For the testbed setup, we recreate the architecture from Figure 4.17. For all servers, except the THREADS VNF, we use a four-core (8 threads) Intel Xeon E3-1230 V2 CPU at 3.30 GHz equipped with 16 GB RAM. Machines of the same type simulate the external network with a benign client and an attacker. Table 4.6 gives additional detail on these machines.

The VNF runs on a six-core (12 threads) Intel Xeon E5-2420 V2 CPU at 1.90 GHz, also equipped with 16 GB RAM. Table 4.7 further describes this machine.

A single HPE 5130-24G-4SFP+ Switch from the Aruba series takes the role of the SDN component in the architecture. It supports OF 1.3, and its hardware table can contain up to 384 entries. For the SDN controller, we used the python-based Ryu controller. All network connections support a maximal bandwidth

Unit	**Value**
Product	Dell PowerVault NX400
CPU	Intel Xeon E5-2420 v2
Default CPU frequency	1.90 GHz
Max CPU frequency	1.90 GHz
Min CPU frequency	1.20 GHz
Cores (Threads)	6 (12)
Cache (L1/L2/L3)	64 KB/256 KB/12288 KB
Memory size	16GB (2 x 8 GB) DDR4
Memory frequency	1.600 GHz
Memory Connection	Dual Channel
Storage Model	HGST Ultrastar A7 K2000 500 GB@7200 rpm
RAID	RAID 5 with four drives
Storage Connection	SATA II (3GBit/s)
1st NIC (Controller & Backend)	2 Port Broadcom Limited NetXtreme II BCM5716 Gigabit Ethernet
2nd NIC (Experiments)	Intel X710 10-Gigabit SFI/SFP+ Network Connection
3rd NIC (Experiments)	Intel X520 10-Gigabit SFI/SFP+ Network Connection
Operating System	Ubuntu 18.04.3 LTS (x86-64)
Kernel	4.4.0-72

Table 4.7: Hardware specifications of the VNF server in the testbed.

of 10 Gbit/s using Intel X520 controller cards. Furthermore, an experiment controller connects to all devices via a separate experiment network. An Apache webserver application runs on the service host. Within the same setup, we deploy SYN cookies and SYNPROXY for comparison measurements.

4.2.3.2 Experiments

We perform multiple experiments to assert successful connection establishment as well as performance under attack.

Connection Establishment

The first experiment aims to validate connection establishment while using THREADS. We, therefore, connect to the service host multiple times. We test for the path of the packets using Wireshark [Com19] dumps. We also test the connections by downloading a 32 MB file form the server.

In a second experiment, we perform 50 HTTP requests and assert the time until a successful answer is received. TCP retransmits packets for which it has not received an acknowledgment after a predefined amount of time. This time frame is called the "Retransmission Timer." It mainly depends on the round-trip time within the network [PA00]. Thus, we measure the time for the connection establishment for various delays between 0 ms and 250 ms. To create these artificial delays, we use Linux's NetEm.

Performance

To evaluate performance, we put the system under the stress of a simulated SYN flood attack. Therefore, the attacker machine creates SYN packets with spoofed addresses using hping3 [San19]. In this mode, the machine can saturate the 10 Gbit/s link up to its theoretical maximum of 14.88 Mpps (mega packets per second). We then measure how many packets were answered by the THREADS VNF, SYN Cookies, or SYNPROXY, respectively. We execute this experiment for one minute per test run and perform at least ten runs per configuration. For this experiment, we chose to set the NICs queue size to 64 packets, and the burst size to 32 packets for THREADS.

Additionally, we performed another experiment to evaluate our capability to establish benign connections during an attack. Hence, we perform an attack from one host and try to download a website from a protected apache webserver 50 times from another host. Since we need to track every packet to ensure the correct functionality of THREADS, the setup for this experiment is limited to 124 kpps.

4.2.3.3 Experiment Results

After detailing the performed measurement studies, this section presents the obtained measurement results. In the following section, we will then discuss these results.

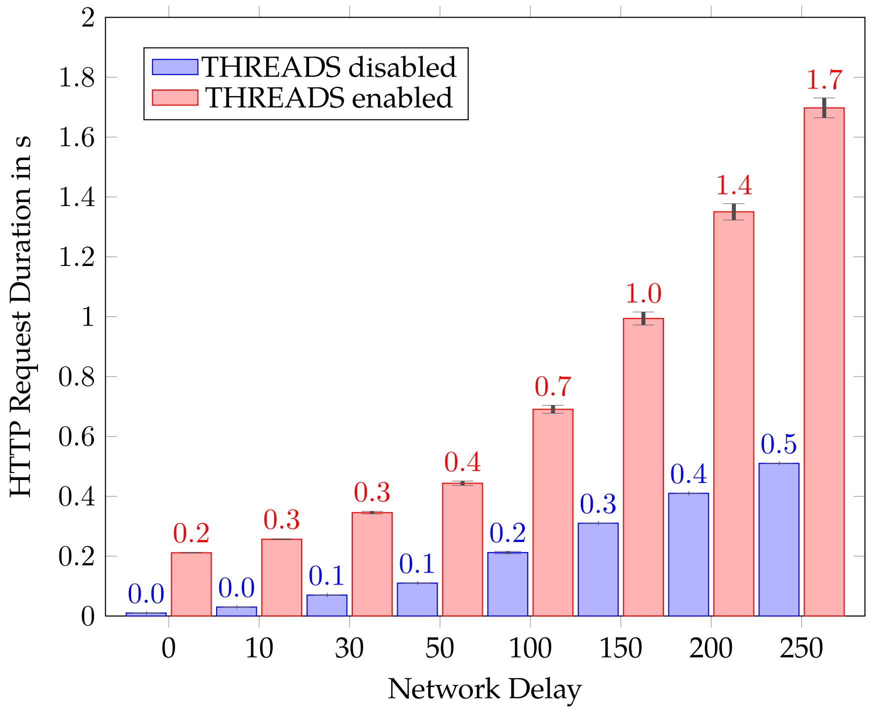


Figure 4.20: Duration until the first HTTP request is processed.

Connection Establishment

Connection establishment works as expected. As described earlier, the VNF handles initial connection establishment, and hands successfully established connections over to the server. The server successfully opens the socket and listens for data. The Wireshark dumps show that all SYN, SYN+ACK, and ACK packets take the correct route.

We found that the client starts transmitting immediately after sending the ACK packet. At this point, the installation of the rules D and E (cf. Section 4.2.1.2) might not have completed. This delay leads to the first segment arriving at the VNF instead of the server. Since the VNF does not listen on that socket, the VNF discards the segment, and the client must retransmit it. This latency creates an additional delay for the first segment. Our second experiment analyzes this effect. To this end, we measured the additional delay induced by THREADS for different artificially added latencies between the client and the SDN switch when downloading a website via HTTP. Figure 4.20 shows the results. It shows that THREADS adds an extra delay of 0.2 seconds without delay and up to 1.2 seconds with 250 ms of delay. Wireshark dumps show that all following segments proceed regularly fashion without additional delays.

Performance

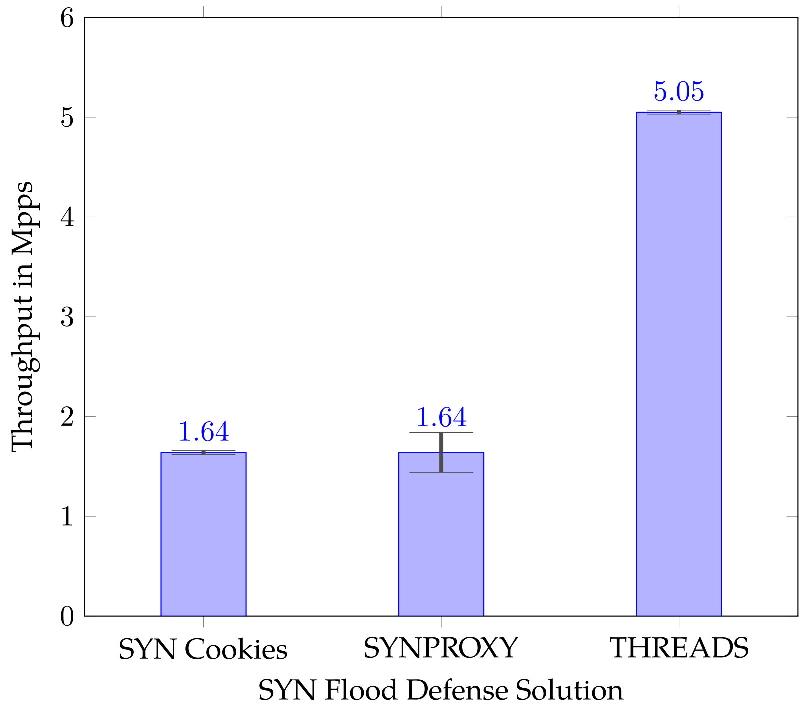


Figure 4.21: Performance of Basic THREADS Implementation

RQ 4.2c When it comes to performance values, we evaluate the total throughput in Mpps as one of the key performance indicators. Furthermore, we compare the throughput achieved by THREADS to values obtained using SYNPROXY as well as SYN cookies.

Figure 4.21 shows the observed values. The y-axis shows the mean throughput in Mpps, and the x-axis distinguishes the different mechanisms. The whiskers represent the 95% confidence intervals. The annotations describe the exact mean value of each bar.

The results show that both SYN cookies and SYNPROXY perform similarly with an average throughput of 1.64 Mpps with SYNPROXY exhibiting slightly larger confidence intervals. THREADS, on the other hand, achieves significantly more throughput with an average of 5.05 Mpps while exhibiting small confidence intervals.

In the other experiment, THREADS was able to establish all benign connections, even at the maximum attack load for this experiment. The attack load did not affect the connection establishment duration.

4.2.4 Discussion

In this section, we present a brief discussion regarding the obtained results and compare the values to similar work from the area.

4.2.4.1 Connection Establishment

Our evaluation has shown that THREADS can perform regular connection establishment. However, the second experiment shows that THREADS adds a delay to the transmission of the first segment. Since the retransmission timer depends on the round trip time, this effect increases when adding additional delay. This behavior, in some situations, can prove harmful to performance for latency-critical applications. However, TCP is a protocol primarily chosen for its reliability. Many applications are not that time-critical for the first packet, as long as the overall process duration does not significantly increase. For example, a Google study shows that the average page load time for mobile webpages is over eight seconds [An17]. Even if the timing is a critical issue, there are three possible solutions.

a) The VNF could forward received packets to the server for a short amount of time until the SDN rule installment completes. However, either the VNF must forward all packets that are neither SYN, SYN+ACK, or ACK without distinction, or this modification would make the VNF stateful.

b) The switch can use an additional flow to delay packets that are neither SYN, SYN+ACK, or ACK if there is no specific rule configured for these packets.

c) The third solution is to use the server's regular TCP connection establishment under normal circumstances. The server triggers THREADS to be enabled once the backlog at the server reaches a pre-configured threshold. In an attack situation, the small extra delay is more acceptable than under normal circumstances.

4.2.4.2 Performance

THREADS shows the capability to handle up to three times as many connection requests as SYN cookies and SYNPROXY. Thus, it is capable of defending against more massive and more violent attacks using the same amount of resources. So far, we can only speculate that this performance increase is partially due to DPDK's low overhead and hardware acceleration from the Intel network cards. Furthermore, we have shown that THREADS can establish new benign connections during attacks.

Unfortunately, it was not possible to directly compare our approach to the competing approaches from [Zhe+18] and AVANT-GUARD from [Shi+13b]. Both approaches do not provide artifacts to re-implement their solutions. However, while [Zhe+18] only provides evaluation regarding SYN Flood detection [Shi+13b] provides some performance evaluation. Their experiments show that AVANT-GUARD has a response time impact of 1.86% in idle mode and 2.1% with a SYN Flood of 1000 pps. Since our approach shows a performance in Mpps, and we evaluated benign connections for up to 124 kpps of attack load, it would be frivolous to interpolate over three to four orders of magnitude. Thus, as we have shown that THREADS can perform in the range of Mpps, we can not draw any conclusion from the underlying data regarding whether AVANT-GUARD can or cannot perform at this level as well.

4.2.5 Parallelization and Parameter Tuning

So far, we only deployed a single-core version of THREADS. Furthermore, THREADS only underwent benchmarks using a default configuration. This section introduces two parallelization approaches for THREADS: (i) locked access to the queues, and (ii) a ringbuffer distribution. Furthermore, we take a look at parameters that are common to all approaches as well as approach-specific parameters. Finally, we perform a parameter study for both approaches using a two factor ANOVA.

4.2.5.1 Joint Parameters

Both approaches have joint parameters that can be modified. The first parameter is the *queue size* inside the network card. This size determines how many packets can reside in the cache before dropping further packets. The next parameter is the *NIC's burst size*. This parameter determines the number of packets sent at once. Next, *mbuf_pool_size* determines the size of DPDK's message buffer. This parameter is of interest because DPDK's documentation suggests setting it to a value of $n = (2^q - 1)$ where q is an arbitrary value. We want to evaluate the gain in following this rule. Finally, we modify the core count.

4.2.5.2 Locked Access

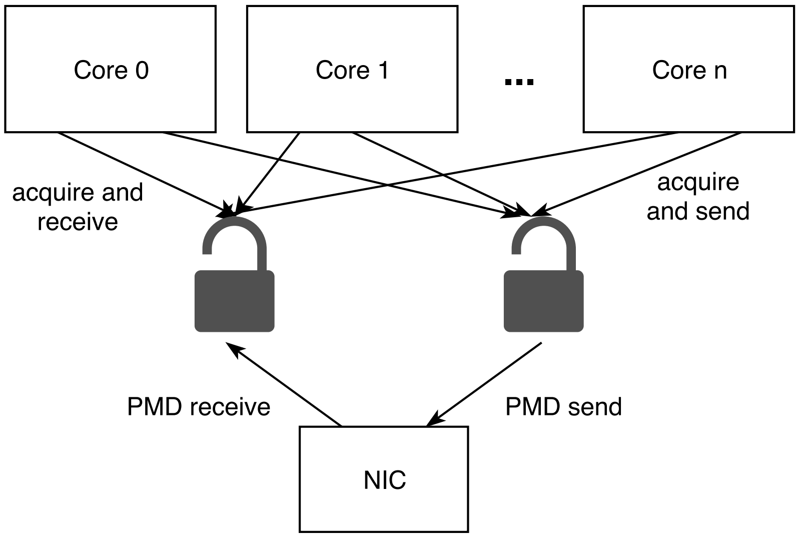


Figure 4.22: Locked Access Multi-threading

Parameter	Values
queue size	64, 2048
burst size	4, 32
mbuf_pool_size	8191, 9192
core count	2, 4, 8, 12

Table 4.8: Variation for Locked Access

The first solution extends the basic approach while adding only minimal additional complexity. We introduce simple locks (true/false-semaphores) to allow multiple cores for packet processing so that only one core at a time can access the receive or the send queue. Figure 4.22 shows this architecture.

Each thread now goes repeatedly through the following steps:

1) Lock the receive queue

2) Receive burst from the queue
3) Unlock receive queue
4) Process packets
5) Lock send queue
6) Send computed packets
7) Unlock the send queue

If a lock is currently not unlocked when trying to lock, the thread waits for it to unlock. When multiple threads are waiting for a lock to unlock, the waiting threads get the lock in an FCFS fashion. Table 4.8 shows the variations used in the parameter study for the locked access.

4.2.5.3 Ringbuffer Distribution

Parameters	Values
queue size	64, 2048
ringbuffer size	32, 512
burst size	4, 32
mbuf_pool_size	8191, 9192
core count	2, 3, 4, 10

Table 4.9: Variation for Ringbuffer Access

A more sophisticated approach is to use a ringbuffer instead of the locking mechanism, as described above. The idea behind this approach is to prevent locking overhead. To achieve this goal, we add a dedicated receiving and sending core for the respective queue. The receiving core takes packets from the receive queue and stores them on the receiving ringbuffer. The processing cores can then take the packets from the receiving ringbuffer. These cores then place processed packets on the sending ringbuffer. The sending core takes the packets from the sending ringbuffer and transmits them. All operations on the ringbuffer are designed to be atomic and therefore require no locking. Figure 4.23 shows the architecture using the ringbuffer.

For this approach, we vary the size of the ringbuffer, and the number of packets taken and written per access.

Table 4.9 shows the variations used in the parameter study for the ringbuffer distribution.

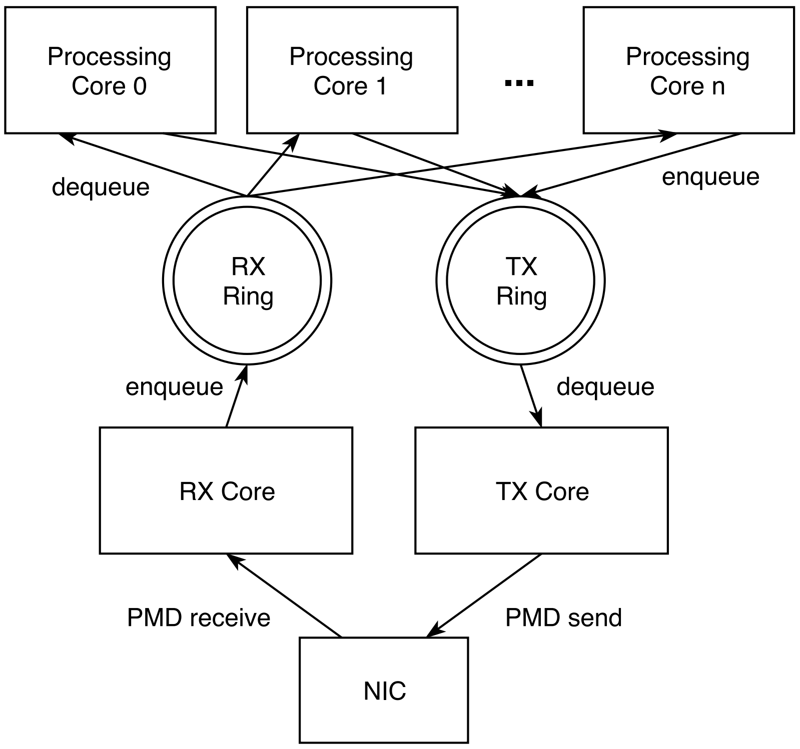


Figure 4.23: Ringbuffer Distribution for Multi-threading

4.2.5.4 Parameter Study

In the following, we present the results obtained during a parameter study regarding the two previously described mechanisms. In both cases, we present the baseline measurements of 5.05 Mpps, as shown in Section 4.2.3.3, as well as the optimal and worst parameter combination. **RQ 4.2d**

Locked Access

First, we evaluate the Locked Access mechanism and identify its key performance parameters. To this end, we perform multiple measurement studies combining the different parameters presented in Table 4.8. For brevity reasons, we omit the detailed results of each of the parameter studies. We instead present the results for both the best as well as the worst parameter combination. We also compare both to the baseline measurement presented before. Figure 4.24 depicts this comparison.

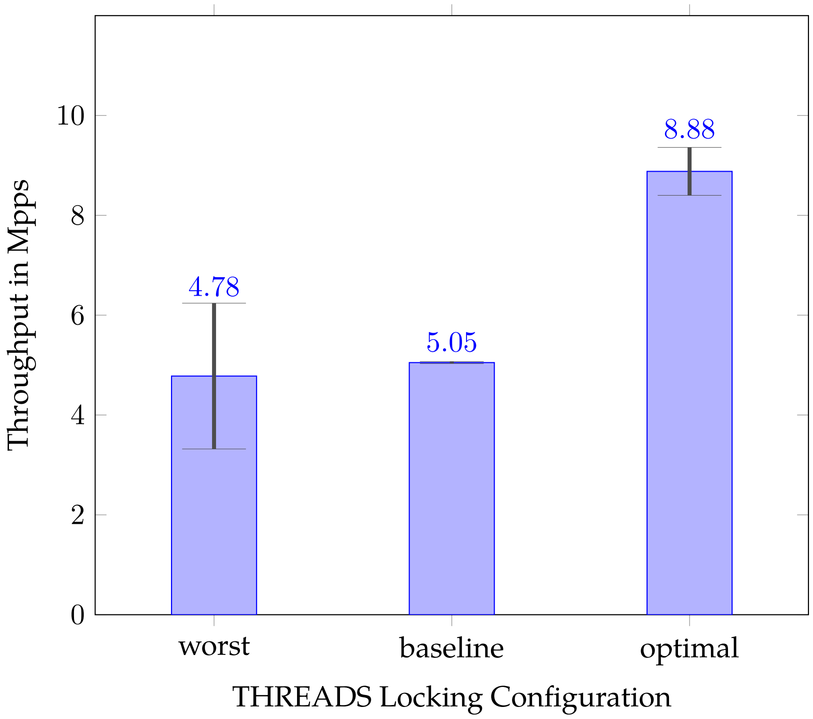


Figure 4.24: Performance Effect of THREADS Configuration Using Locked Access

The figure, again, shows the mean throughput in Mpps along the y-axis and the different scenarios along the x-axis. The whiskers represent the 95% confidence intervals. The baseline measurement represents the single-core scenario presented before.

Applying eight cores, a mbuf_pool_size of 8191, a burst size of 32, and a queue size of 64 results in the worst performance. The achieved performance at 4.78 ± 1.46 Mpps, on average, even underperforms the single-threaded solution. The highest observed performance gain occurs when using four parallel cores, a mbuf_pool_size of 8191, a burst size of 4, and a queue size of 2048. The achieved performance at 8.88 ± 0.48 Mpps is significantly faster than the single-threaded baseline and outperforms the worst-case performance by a factor of almost two. These results show that the configuration is essential, and sub-optimal configurations can lead to worse performance than single-threaded operation.

Evaluating the various parameter combinations has shown that the queue size has a significant impact on the performance since the larger queue size leads to performance increases of up to 21%. The NIC's burst size also has a significant impact, with the smaller burst size increasing performance by up to 11% compared to the larger burst size. The mbuf_pool_size parameter has no significant impact as the difference in all cases is below 1% and does not reach the threshold for significance. Similarly, increasing the core count only results in performance gains for up to four parallel cores. Increasing the core count even further does not yield performance gains and even reduces the performance in some cases.

Ringbuffer Distribution

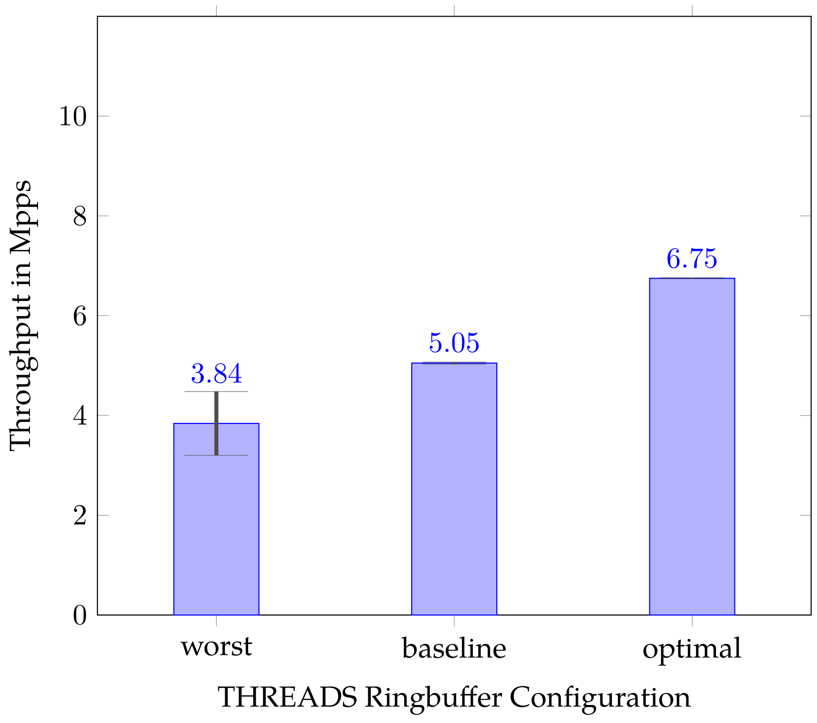


Figure 4.25: Performance Effect of THREADS Configuration Using Ringbuffer Distribution

Benchmarking the ringbuffer mechanism detailed before leads to similar observations.

The worst-case performance occurs when using ten cores, a mbuf_pool_-size of 8192, a NIC burst size of 4, and a queue size of 2048. In this scenario, we configured the ringbuffer to have a size of 32 and a burst size of 1. The achieved performance at 3.84 ± 0.64 Mpps, on average, underperforms the single-threaded solution once more.

The best performance shows when using four parallel worker cores (6 total cores due to RX and TX core), a mbuf_pool_size of 8191, a burst size of 4, and a queue size of 64. We configured the ringbuffer to hold 512 packets and feature a burst size of 32. The achieved performance at 6.75 ± 0.01 Mpps is significantly higher than the single-threaded baseline and outperforms the worst-case performance.

Figure 4.25 shows the differences between baseline, worst-case configuration, and optimal configuration. Similar to the locked access, it shows that the configuration is essential, and sub-optimal configurations can lead to worse performance compared to single-threaded operation. Here, the performance degradation of the worst configuration is even more substantial compared to the locked access solution.

Investigating the impact of single parameters has shown that the influence of the queue size falls below the threshold of significance, with only an average advantage of 2% for the smaller ringbuffer size. The same can be said for the ring size, as the higher value on average has a non-significant advantage of 2%. The impact of the NIC burst size and the mbuf_pool_size is also negligible (0%). The ring burst size, on the other hand, has a significant impact as selecting the higher size value results in a performance boost of 31% on average. The optimal average core count in this scenario is two active cores, but the advantage over four cores is within the limits of significance. For higher core counts, the performance decreases.

4.2.5.5 Multi-threading Performance and Scaling

We have introduced two approaches to multi-threading for the THREADS implementation. Figure 4.26 shows that both approaches offer an increase in performance compared to the single-threaded baseline. However, the speedup is limited. For the locking approach, a speed increase of 64% occurs when using four cores. The ringbuffer approach only yields a speedup of 34% when using six cores. However, with the locking approach, a single THREADS instance with four cores can tackle an attack that can flood a connection for more than half of the theoretical limit of a 10G network.

RQ 4.2e In general, THREADS does not scale well vertically (adding additional CPU cores). For deployment in real environments, it is, therefore, advisable to use small or medium instances and deploy multiple instances of THREADS instead of using large instances. These instances can be easily load-balanced since THREADS is stateless.

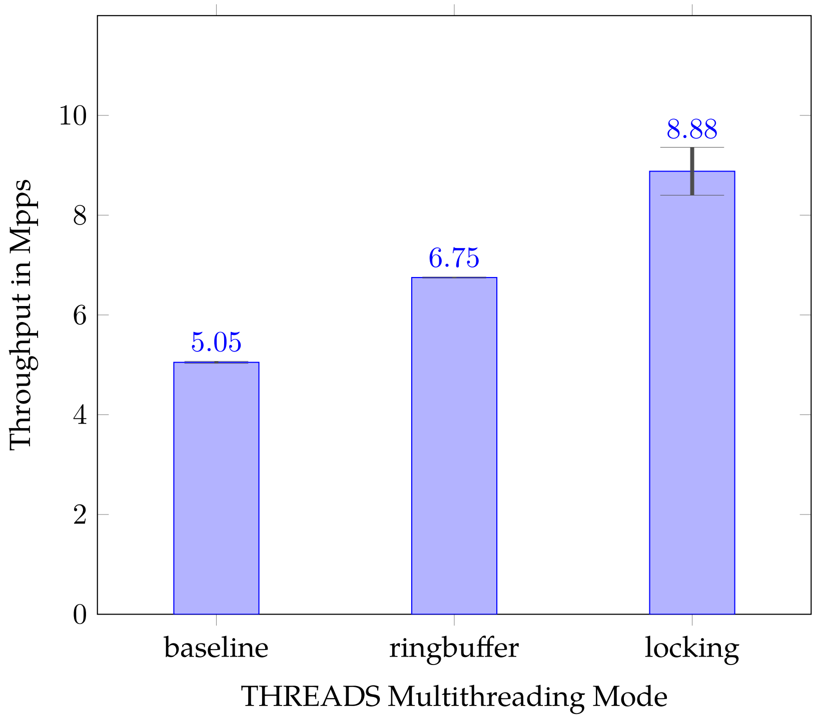


Figure 4.26: Comparison of different multi-threading approaches implemented for THREADS

4.3 Summary and Evaluation of Research Questions

In this chapter, we presented two approaches to increase the performance of single security functions inside a network using SDN. These solutions included a bypassing solution for IDS and a state-less mitigation solution for TCP SYN flood attacks.

RQ4.1a How can SDN-based approaches improve Intrusion Detection Systems?

In general, SDN allows us to use dynamic or static rules to redirect traffic based on its properties. IDS require a significant amount of computing resources for each packet they inspect. Furthermore, a high load level can lead to IDS wrongly classifying incoming data units as either false positives or negatives. Thus, SDN can improve the performance and security metrics of IDS by redirecting irrelevant data away from it and, thereby, relieving it from this load.

RQ4.1b What effects do bypassing approaches have on the performance and security of Intrusion Detection Systems?

Compared to an inline IDS, all static and adaptive bypassing approaches increase throughput and the number of successful HTTP requests. Furthermore, they reduce the latency, residence time, the number of double classifications, and the lost packets. None of the approaches leads to false-negative classifications, but, in some cases, can increase the number of false positives.

RQ4.1c How do adaptive approaches, which perform reconfigurations at runtime, compare to static approaches?

When considering their optimal configuration, the adaptive approaches have better performance metrics than the compared static approach. In these configurations, the adaptive approaches can even match the throughput performance of a system without the IDS, albeit with higher latencies. The static approach has a lower number of lost packets, but, on the other hand, generates a higher number of false positives. However, the performance of the adaptive approaches largely depends on the employed switch (see also **RQ4.e**).

RQ4.1d How do different workload levels impact the performance and security of the SDN-based approaches?

Increasing the workload beyond the level the system can handle without inducing a queuing or dropping behavior impacts the performance. Overall, throughput decreases, latencies and the number of false positives grow, and the system can no longer handle all requests successfully. In general, all previously state qualitative observations stay true. Nevertheless, the scale of these observations change. An especially noteworthy change is the decrease in precision for the static approach. In general, the SDN-based approaches suffer less from the overload situation than an inline IDS.

RQ4.1e Do the SDN-based approaches change their behavior when using hardware or software switches?

We found that the performance of the adaptive approaches strongly depends on the used switch type. When using a software switch instead of a hardware switch, this can more than double the throughput. While the traditional security metrics are hardly affected, the number of lost attacks is severely reduced — especially in non-overload situations. In comparison, the static approach does neither profit nor suffer from changing the switch type.

RQ4.2a How can SDN-based approaches improve DDoS Protection Systems against SYN flood attacks?

SDN can counter shortcomings of existing DPS solutions against SYN flood attacks. These shortcomings include the absence of independent scalability or the need for all traffic to pass through the security function even after connection establishment. By using SDN, it is possible to create a state-less DPS VNF.

RQ4.2b What is necessary to make such a solution stateless and independently deployable?

Our solution THREADS requires two components inside an SDN-enabled network: (i) a stateless VNF taking care of the TCP handshake and forwards successful handshakes to the protected service, and (ii) a modification to the service to the protected service to accept connections from the VNF and instruct the SDN controller to stop directing packets of this connection via the VNF.

RQ4.2c How does such a solution perform compared to existing solutions?

THREADS performs up to three times as fast as the existing solutions without parameter tuning. However, it introduces a significant delay for connections for the first data packet. We presented multiple solutions to this issue.

RQ4.2d To what extent can parallelization improve the performance of such a solution, and how vital is parameter-tuning?

We presented two parallelization approaches: (i) a locking approach with semaphores for the send and receive queues, and (ii) a ring buffer. These approaches show that they can improve the performance relative to the non-parallel and non-optimized baseline by another 76%. The more straightforward locking approach outperforms the ringbuffer approach and requires fewer cores at its optimal configuration than the ringbuffer. As a result of this, parameter-tuning is very important, since poorly configured parallel deployments of THREADS produce a performance below the single-threaded baseline.

RQ4.2e Which deployment and scaling strategies suit the solution?

THREADS does not scale very well vertically with optimal performance at four (locking) and six (ringbuffer) cores. Thus, for deployment in more extensive infrastructures or cloud environments, we suggest focussing horizontal scaling using many small or medium instances. Since THREADS is stateless, load-balancing between the instances is easily attainable.

Chapter 5

Performance Modeling for Security Service Function Chain Orders

Today's network attacks rely on massive bot networks. Their attacking power raises as the number of online devices rapidly grows in times of the IoT. The ending of Moore's Law [TW17] (promising doubling computing resources every two years) limits the opportunity to throw in additional resources to fight attacks. Moreover, booking additional resources on demand is very costly, also considering that the owners of bot networks do not have to pay for their attack resources.

Information systems that provide services via the Internet offer various attack vectors. We presented common attack types in Section 2.2. For each type of network attack, there are dedicated security functions to defend the system against that attack. For example, a firewall fights HTTP floods, and DPSs like THREADS (see Section 4.2) can mitigate SYN floods. Multiple security functions together comprise SSFCs to protect systems against a set of several attack types. Section 2.6 shows an example of SSFCs.

For most systems, a direct correlation between consumed resources and the number of processed packages exists. In contrast, security functions (and therefore SSFCs) stand out, as they (i) behave differently under various traffic conditions (package characteristics and overload situations [TL12]), and (ii) drop packets deemed as malicious reducing the load on subsequent security functions.

Security systems can be taken out of service by attacks — when in a suboptimal configuration — long before purposefully utilizing all available resources. As an example, we illustrate this by a chain of two security functions, a firewall, and a DPS. We assume that each security function can handle a throughput of 100 MBit/s while accurately filtering malicious packages. The described system is now the target of a DDoS attack on a port not filtered by the firewall. The total traffic rate is 1 GBit/s, and 90% of traffic is malicious. The throughput and required resources of the SSFC for this attack highly depend on the security functions' order.

If we put the firewall first, there would be no filtering, and all packets proceed to the next stage. Consequently, ten instances of each security function type would be required to handle all traffic. These resource requirements sum up to a total of 20 instances. Otherwise, if we place the DPS in front, the first layer would still require ten instances, but the DPSs would drop the malicious 90% of traffic before reaching the second layer. Instead of ten instances, a single firewall instance can handle the remaining 10% percent of traffic. The use of a total of eleven instances makes it possible to survive the attack, which means a 45% reduction in the resources required compared to bringing a firewall forward. Another attack could be on a port blocked by the firewall. Then the placement of the firewall in front of the DPS inverts the efficiency discrepancy. While dropping rates might differ for different security functions and traffic compositions, our illustrative example demonstrates potential efficiency gains when tailoring the order to the incoming traffic.

While a traffic-aware reordering would provide benefits, today's operation of SSFCs relies on a static order of security functions. We propose a self-aware approach to automatically reorder security functions based on incoming traffic to address this deficiency. Therefore, the security functions report detected attacks to a central instance, the Function Chaining Controller (FCC). We model the behavior of different security function types and configurations based on measurements as well as different traffic types for attacks, benign workloads, and combinations of them. Based on these models, we then infer a configuration tailored to incoming traffic, which can be instantiated by dynamically reordering the SSFC.

The benefit of our approach is to improve the efficiency of traffic processing inside the SSFC. This improvement results in an increased throughput to resources ratio. Since overload situations are avoided or at least mitigated, we additionally expect a reduction of the false-positive and false-negative rate. Finally, the chances of the security system being permanently disabled due to an attack shrink.

Research Questions

In this chapter, we tackle several research questions. All of the following research questions are part of the meta-research question **MRQ 5:** *To what extent can we improve security systems by introducing dynamic function chain reordering?*. The numbering of these research questions maps to the sections of this chapter.

If a section deals with more than one research question, those questions have their number appended by ascending Latin letters.

RQ 5.1 What components and capabilities define a Security Service Function Chaining framework?

RQ 5.2a How do single security functions perform under attack load?

RQ 5.2b What is the impact of the ordering when combining different security service functions?

RQ 5.3a How to model single security functions for the reordering decision?

RQ 5.3b How to model security function chains for the reordering decision?

RQ 5.3c What strategies are suitable for determining a better order?

Chapter Structure

To answer these research questions, we first present the general idea behind dynamic function chain reordering in more detail and sketch a coarse architecture in Section 5.1. Next, in Section 5.2, we analyze the effect of security function ordering by first analyzing the performance of single security functions and then moving on to combinations of security functions. Then, Section 5.3 discusses decision making regarding the desired ordering, and we conclude the chapter in Section 5.4 — including an analysis of the gathered answers to the research questions.

5.1 General Idea

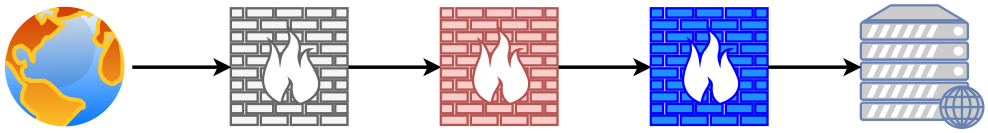


Figure 5.1: Classic architecture of security systems.

As described in Section 2.6, different services have a different set of security functions that defends them against attacks from an external network. Figure 5.1 shows an example with three different security functions forming an SSFC. In most current security architectures, those SSFC are hard-wired or — if using SDN for interconnection — interconnected using a fixed order.

In the following, we make four assumptions:

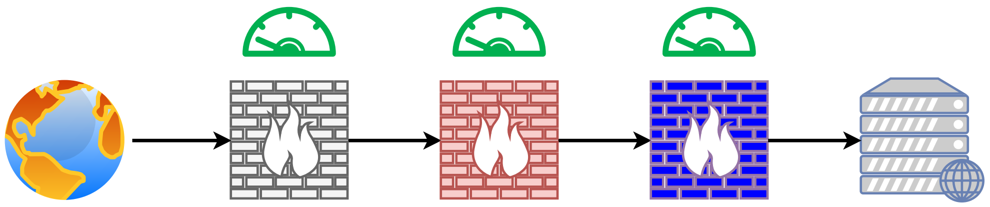
- A security function can defend against precisely one class of attacks.
- The attack classes associated with different security functions are mutually exclusive.
- Security functions drop packets deemed malicious.
- Traffic — benign and malicious — passing through a security function creates a resource demand.

Figure 5.2 presents multiple configurations of the setup from Figure 5.1 under different types of load. Figure 5.2a shows the setup under standard load without an attack occurring. In this scenario, the SSFC order is not relevant. All packets are benign and, therefore, have to pass through all security functions in the SSFC. Reordering the functions would not change the resource demand generated by the benign packets.

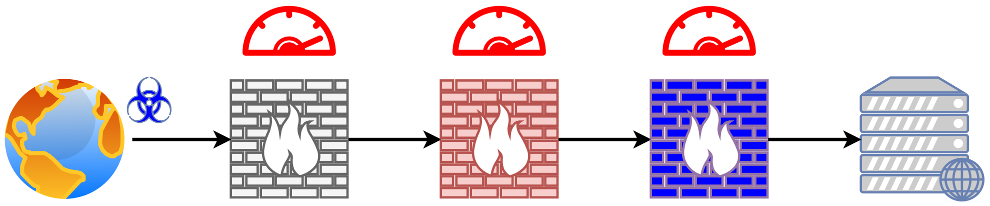
In Figure 5.2b, an attack occurs from the external networks. The attack corresponds to the attack class against which the blue security function defends. With the initial configuration, the traffic passes through all security functions in the SSFC. Thus, it creates resource demands at every step until the last security function stops it. This order would require scaling the white and red security functions as well as the blue security function, to handle the load. Changing the order of the red and white security function would not affect the resource demands since all traffic would still pass through both security functions.

In Figure 5.2c, the blue security function moves up to the middle of the SSFC. Thus, the attacks belonging to the blue attack class must only pass through the white security function until the blue security function discards them. The malicious traffic, therefore, creates resource demands at the white and blue security functions but not at the red function. If scaling is necessary, it would only apply to the white and blue security functions. Switching the red and white security functions can have an impact on the total resource demand. If one of the two security functions has a higher resource demand per processed packet, placing this function at the back of the SSFC would reduce the total resource demand compared to the reversed SSFC order.

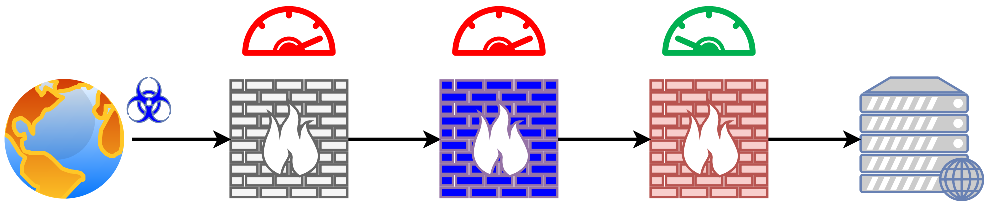
Finally, in Figure 5.2d, the blue security function is at the front of the SSFC. This SSFC order leads to the first security function dropping all malicious traffic immediately. Thus, traffic does not pass on to the white and red security functions and only creates resource demands at the blue security function. Therefore, potential scaling as a reaction to the attack is only necessary for the blue security function. The ordering of the white and red security functions is



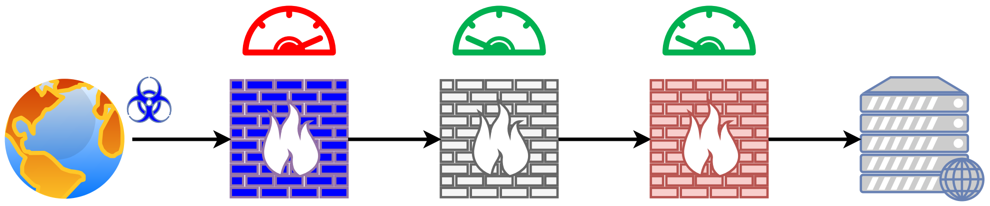
(**a**) Example setup at regular operation with a benign load. All security functions perform at the green level.



(**b**) Example setup under attack. Attacks are of blue attack class. All security functions are stressed because malicious traffic passes through all security functions.



(**c**) A partially reordered setup under attack. Attacks are of blue attack class. Only the white and blue security functions are stressed because the blue function drops malicious traffic before it reaches the red security function.



(**d**) Fully reordered setup under attack. Attacks are of blue attack class. Only the blue security function is stressed because the blue security function drops malicious traffic before it reaches the other security functions.

Figure 5.2: The example setup under different load patterns with and without reordering including load levels per security function.

Ordering	red		white		blue		total
	rel. load	no of inst.	rel. load	no of inst.	rel. load	no of inst.	no of inst.
red – white – blue	1000%	10	500%	5	667%	7	22
red – blue – white	1000%	10	667%	7	25%	1	18
white – red – blue	500%	5	1000%	10	667%	7	22
white – blue – red	500%	5	667%	7	50%	1	13
blue – red – white	667%	7	50%	1	25%	1	9
blue – white – red	667%	7	25%	1	50%	1	9

Table 5.1: Example calculation of the resource demands (in instances) for different SSFC orders. Throughput per instance: red 100 MBit/s, blue 150 MBit/s, white 200 MBits/s. The load profile is 950 MBit/s of malicious traffic matching the blue security function and 50 MBit/s of benign traffic.

not relevant for the total resource demand since the malicious traffic does not reach these two security functions, and all benign traffic passes through both security functions.

We calculate the resource demand of the six possible permutations of the example SSFC to illustrate the impact of the SSFC order. We assume three example security functions. One instance of the red security function can handle 100 MBit/s, one instance of the white security function can handle 200 MBit/s, and one instance of the blue security function can handle 150 MBit/s. As a sample workload, we assume 50 MBit/s of benign traffic and 950 MBit/s of malicious traffic. We calculate the relative load to one instance of each security function, the number of required security function instances per security function, and the total number of required instances.

Table 5.1 shows the computed results. The results for the example provide that optimal configurations (in this case, two exist) can reduce the total number of required instances from 22 to 9 or by 59%. It also confirms our previous statements regarding the order of the security functions that do not defend against the occurring attack.

In practice, attacks more and more occur in conjunction with other attacks. Mainly DDoS attacks frequently serve as a so-called smokescreen attack to hide other attacks like intrusions. Thus, in many cases, malicious traffic comprises a mix of attacks. To assert the optimal SSFC order for such a mix again requires a computation like in Table 5.1. A significant difference for attack mixes is that the order of all security functions in the SSFC becomes essential. Unlike in the previous example, only one optimal configuration remains.

RQ 5.1 **Function Chaining Controller:** The gathering of the required information to approximate the traffic composition, the modeling, and decision for the ordering of the SSFC, and the enforcement of the desired ordering require an entity taking up these responsibilities. This entity we call the Function Chaining Controller (FCC).

Figure 5.3 shows a coarse description of the FCC's operation. The FCC connects to the security functions and the service host to access their performance statistics. Based on this information, the FCC decides whether a reconfiguration is beneficial. At first, in Figure 5.3a, no attack occurs. Therefore, no reconfiguration is necessary.

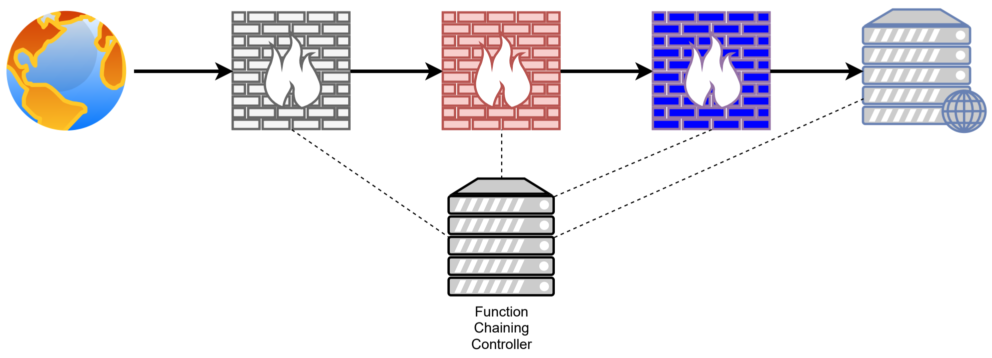
Once an attack occurs, as seen in Figure 5.3b, the FCC deducts the composition of the malicious traffic from the available information. Based on this composition, the FCC computes the optimal order for the SSFC. The FCC then enforces that ordering.

Once the reordering completes (Figure 5.3c), the FCC continues monitoring the traffic. Every time the approximated composition of the malicious traffic changes, the FCC computes whether another reconfiguration is necessary or not.

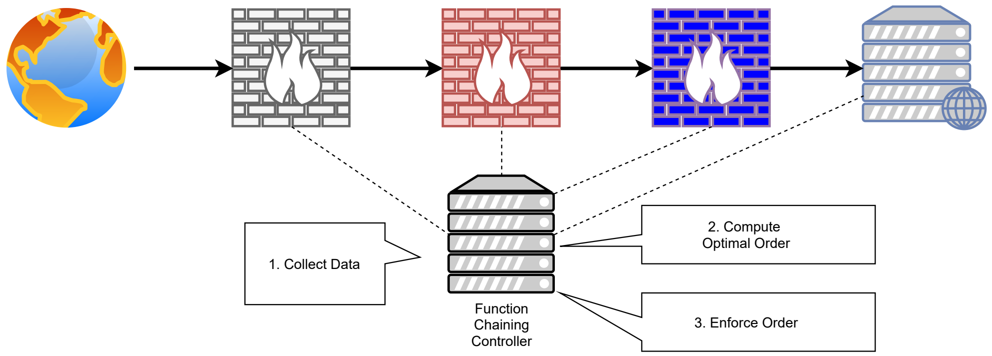
We design the FCC as a self-aware system. Thus it implements the self-aware LRA-M loop, as shown in Figure 5.4. Based on its state and phenomena (e.g., data about the security functions' states) comprise empirical observations. The FCC uses these observations to learn and reason in combination with a model to achieve its goal (highest throughput with the least amount of resources). Based on this reasoning, the FCC acts (enforcing a new order for the SSFC).

5.2 Effect of the Security Service Function Chain Order

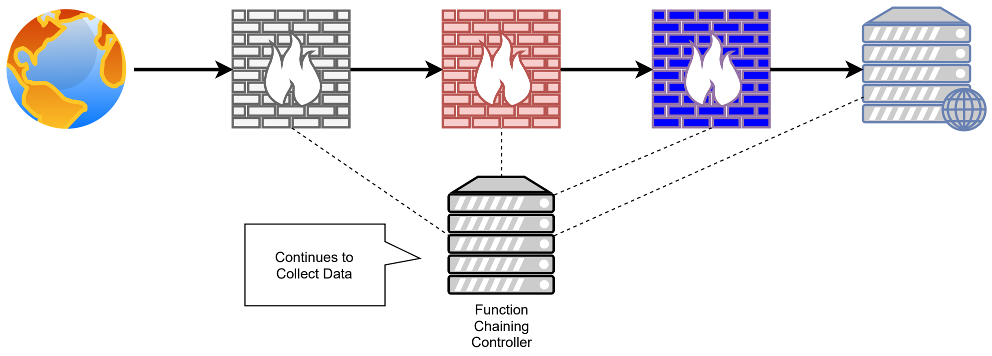
In the introduction to this chapter and the previous section, we claimed that the order of security functions inside the SSFC influences its performance. In this section, we evaluate security functions and SSFCs with different security function orders to assert our claim. At first, in Section 5.2.1, we present the used evaluation environment. We then measure the performance of three different security functions under benign and malicious traffic in a standalone deployment in Section 5.2.2. The measured security functions are a firewall, a DPS, and an IDS. In Section 5.2.3, we put these security functions in SSFCs consisting of two such functions and vary their order. Last, in Section 5.2.4, we



(**a**) Normal operation. The FCC connects to the service hosts and the security functions to gather information.



(**b**) An attack occurs. The FCC detects the attack based on the available information. It then computes the optimal ordering and enforces ist.



(**c**) After applying the new ordering, the FCC continues monitoring the system to decide whether further reordering is necessary.

Figure 5.3: Workflow for the Function Chaining Controller from normal operation over the attack occurrence to the reordering execution.

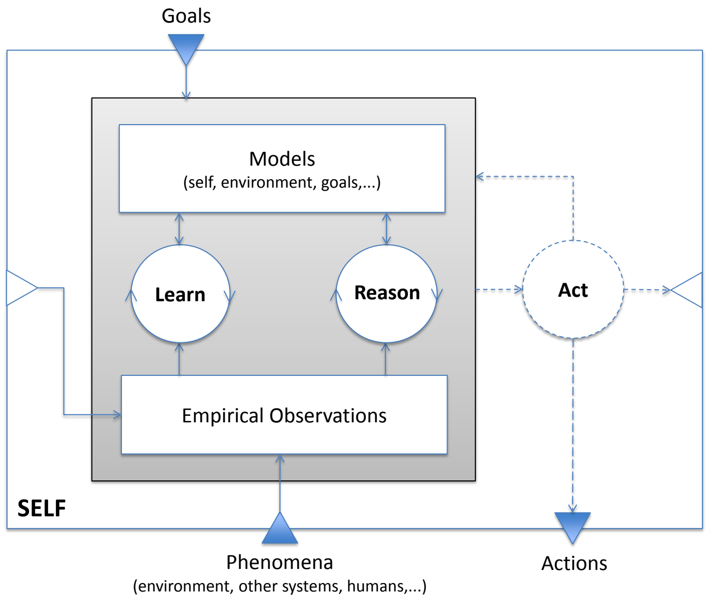

Goals
Models
(self, environment, goals,...)
Learn
Reason
Act
Empirical Observations
SELF
Phenomena
(environment, other systems, humans,...)
Actions

Figure 5.4: Self-aware LRA-M (Learn, Reason and Act based on Models) Loop [Kou+17].

discuss the results and the conclusions that we must consider for modeling and decision making in Section 5.3 and incorporate into the reordering framework in Chapter 6.

5.2.1 Evaluation Environment

We designed a testbed that can incorporate benign and malicious workloads to evaluate security function performance, using single security functions and composite SSFCs with modifiable security function orders and different server applications. Figure 5.5 shows the architecture of that testbed.

5.2.1.1 Hardware Components

We use a total of six physical servers to incorporate this setup. These take the roles of (i) a client and attacker, (ii) an application server (the protected application), (iii) a DDoS Protection System (DPS), (iv) a firewall, (v) an Intrusion Detection and Prevention System (IDPS), and (vi) an SDN, Experiment, and Function Chaining Controller.

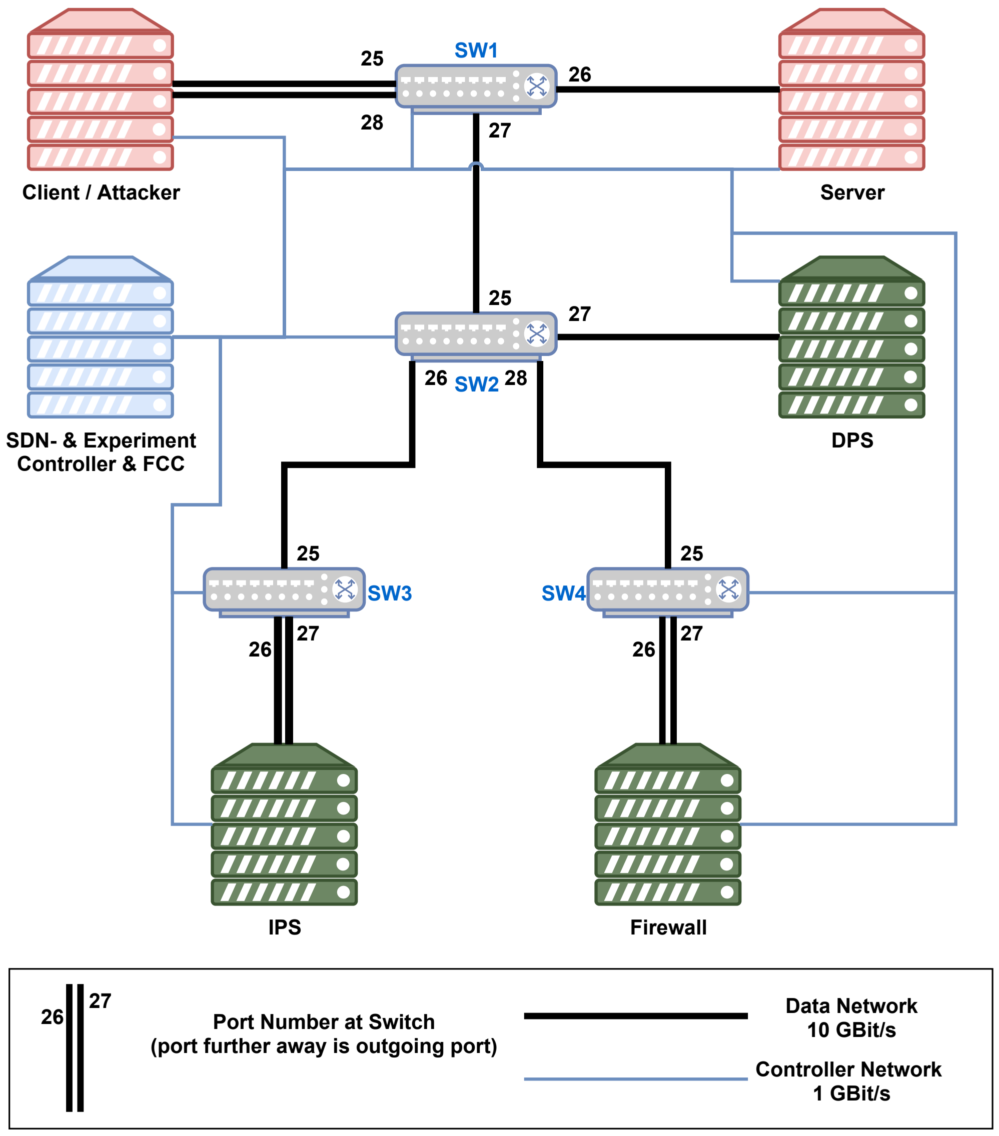


Figure 5.5: Evaluation setup for different security functions and SSFC orderings.

Unit	Value
Product	Dell PowerEdge R210 II
CPU	Intel Xeon E3-1230 v2
Default CPU frequency	3.30 GHz
Max CPU frequency	3.70 GHz
Min CPU frequency	1.60 GHz
Cores (Threads)	4 (8)
Cache (L1/L2/L3)	64 KB/256 KB/8192 KB
Memory size	16GB (2 x 8 GB) DDR4
Memory frequency	1.600 GHz
Memory Connection	Dual Channel
Storage Model	HGST Ultrastar A7 K2000 500 GB@7200 rpm
Storage Connection	SATA II (3GBit/s)
1st NIC (Controller & Backend)	2 Port Broadcom Limited NetXtreme II BCM5716 Gigabit Ethernet
2nd NIC (Experiments)	Intel X520 10-Gigabit SFI/SFP+ Network Connection
Operating System	Ubuntu 18.04.3 LTS (x86-64)
Kernel	4.4.0-72

Table 5.2: Hardware specifications for all servers inside the evaluation environment

For all servers, we use a four-core (8 threads) Intel Xeon E3-1230 V2 CPU at 3.30 GHz equipped with 16 GB RAM. The machines connect to a backend network (e.g., for downloading software packets) and the controller network (controlling experiments, gathering data) using a Broadcom 1 Gb/s controller. Additionally, all servers — except the controller — also connect to a 10 GBit/s network using Intel 10 GBit/s cards. Servers running pass-through applications like the IDPS and the firewall connect to this network using two ports. The client & attacker machine is serving two roles and, therefore, also uses one link for each role. Table 5.2 gives additional detail on these machines.

Two standard non-programmable 1 GBit/s HPE ProCurve 3500yl-24G switches provide the connectivity for the backend and controller network. Four HPE 5130 24G 4SFP+ EI SDN switches span the network for the experimentation data. Since these switches only have four 10 GBit/s network ports (numbers 25 through 28), we require four of them to interconnect all servers, as shown in Figure 5.5. The switches provide enough backplane switching capacity to ensure that this setup does not become a bottleneck. Figure 5.5 shows which cable connects to which 10 GBit/s port on the switches to facilitate the reproducibility of this setup All 1 GBit/s links use Cat 6a copper cables, and all 10GBit/s links use fiber-optic cables.

5.2.1.2 Software Components

Traffic Generator (Benign): On the first 10 GBit/s interface of the client & attacker server, we generate benign HTTP traffic. For this purpose, we use HTTP Load Generator [KDK18][1]. HTTP Load Generator can generate HTTP loads. The tool supports constant load levels, as well as varying load intensities. It is possible to manually configure these levels or retrieve them from a tool like LIMBO [Kis+17]. Furthermore, the traffic generator supports the integration of power meter devices. HTTP load generator comprises two components: a director and a generator. The *director* component controls one or multiple generator components and directs when they send packets. The *generator* component performs the actual packet generation. We deploy the director component on the experiment controller host and the generator component on the client & attacker server.

Traffic Generator (Malicious): We use the second 10 GBit/s interface of the client & attacker server to create malicious packets. On it, we create SYN, UDP, and IDS floods using Cisco's Trex [Sys20] generator. Trex is an open-source stateful and stateless traffic generator. For the chosen attacks, we do not require the stateful generation and, therefore, use only the stateless mode. Trex uses the DPDK to create high-volume loads. DPDK binds the whole interface to the program. Therefore, the benign HTTP traffic needs to use a different interface. To generate HTTP floods, we employ BoNeSi [Gol18] — a BotNet Simulator. BoNeSi can create high-volume HTTP floods by emulating spoofed IP addresses.

Intrusion Detection and Prevention System: The IDPS host runs the Snort IDPS in version 2.9.7. Snort is a popular, open-source IDPS developed by Cisco and also is the foundation of Cisco's commercial IDPS solutions. Snort uses both 10 GBit/s interfaces, one for the incoming, and one for the outgoing traffic.

[1]available at `https://github.com/joakimkistowski/HTTP-Load-Generator`

For the measurements, we extended the standard Snort Community signatures by several rules. Those are available in Appendix A.1.

Firewall: Like the IDPS, the Firewall uses one interface for incoming and one for outgoing traffic. We interconnect both interfaces using a Linux bridge, and Netfilter/iptables rules accomplish the packet filtering. The used rules are available in Appendix A.2.

DDoS Protection System (DPS): As a DPS, we use a modified version of THREADS. The main difference to the version presented in Section 4.2 is that the VNF is in charge of issuing network modifications at the SDN controller. Also, we made some further modifications to this application. In the following lines, we give a summary of this version of THREADS. For more information about THREADS' architecture, consult Section 4.2.1. THREADS comprises two components. A DPDK application attached to a single 10 GBit/s interface is handling incoming packets. It forwards successful TCP handshakes to a Python application via a named pipe that is generating the needed calls to the SDN Controller. A few changes to the original version were necessary to deliver more predictable performance. First, we send all packets that reach the DPS erroneously back on the same interface instead of dropping them. This forwarding is necessary because the required network change after a successful handshake is too slow (see also the experimentation results in Section 4.2.3.3) and, therefore, some of the early data packets reach the DPS instead of the protected service. Second, the TCP packets generated by the DPDK application did not honor the TCP options set by the sender/receiver. Especially setting an explicit MSS improved the system's performance drastically. Appendix A.3 shows these modifications.

Protected Service: The target server runs TeaStore. "The TeaStore is a micro-service reference and test application to be used in benchmarks and tests. The TeaStore emulates a basic web store for automatically generated, tea and tea supplies. As it is primarily a test application, it features UI elements for database generation and service resetting in addition to the store itself. The TeaStore is a distributed micro-service application featuring five distinct services plus a registry. Each service may be replicated without limit and deployed on separate devices as desired. Services communicate using REST and using the Netflix Ribbon client-side load balancer. Each service also comes in a pre-instrumented variant that uses Kieker to provide detailed information about the TeaStore's actions and behavior. [Kis+18]" A small kernel modification to the underlying apache web server is necessary to work with THREADS, We applied the modification to version 4.4 of the Linux kernel.

SDN Controller: We use Ryu 4.15-0 as the SDN controller. Its `ryu.app.ofctl_rest` module provides a REST-based interface for deploying flows.

Monitoring and Metrics Collection The testbed measures and records the following metrics from various sources:

- CPU usage of each server in various states: user, iowait, softirq, system
- Total number of sent benign HTTP requests
- Number of successful benign HTTP requests
- Average ICMP latency between sender and receiver
- ICMP packet loss between sender and receiver
- Average TCP SYN latency
- TCP SYN packet loss

Telegraf's [Inf20b] `inputs.cpu` plugin [Inf20c] collects CPU usage statistics and sends them to an InfluxDB [Inf20a] running on the experiment controller. It reports the usage of the CPU in various states, of whom `user`, `system`, `iowait`, and `softirq` are of interest. We use Grafana [Lab20] to visualize the gathered data. We use a moving average of 10 seconds to help to interpret the data. The client and attacker machine sends a load of x HTTP requests and an attack of intensity y to the receiver for a time t to measure the CPU usage of a certain chain/attack combination. The unit for the intensity is MBit/s for SYN, UDP, and IDS floods, and requests per second for the HTTP flood.

Telegraf's `inputs.mem` plugin [Inf20d] collects the memory usage. Processing and interpretation of memory usage follow the approach for CPU usage.

The HTTP traffic generator reports the number of total and successful requests during the run. We store the output in a file and analyze it afterward.

The ping command allows measuring the latency and packet loss between the sender and receiver. We run an attack of intensity x for a time t and continue pinging during that time.

We measure the SYN latency and packet loss using the hping3 [San20] command. Hping3 sends a TCP SYN request and measures he time until the corresponding SYN+ACK answer arrives. To determine this value, we run an attack of intensity x for a time t.

```
# clone git repository (adapt directory to your wishes)
git clone git@github.com:bladewing/SSFC_testbed.git
↪  /tmp/dynamic-chaining
cd /tmp/dynamic-chaining

# instantiate the vagrant machines - this can take some time
vagrant up

# ssh to the controller machine
vagrant ssh master
```

Listing 5.1: Script to bring up all virtual machines in the environment and ssh to the controller machine.

```
# go to the shared folder containing the ansible scripts
# the folder /vargrant holds the git repository
cd /vagrant/ansible

# make the bootstraping script runnable
chmod +x bootstrap_ansible.sh

# install required packages
./bootstrap_ansible.sh

# run the ansible playbook
ansible-playbook -i hosts-vagrant-master playbook.yml
```

Listing 5.2: Script to deploy all required services, tools, etc. on all machines using ansible from the controller machine.

5.2.1.3 Deployment

We provide the configuration files for our setup for download at `https://github.com/bladewing/SSFC_testbed`. Since most people do not have the exact switching and benchmarking hardware, like the one we use, we provide a fully virtualized environment to deploy our solution, where we replace the four hardware switches with a single Open vSwitch instance.

To use the environment, a user must install Vagrant [Has20]. We tested all of our scripts with Vagrant 2.2.7.

Listing 5.1 shows how to bring up all required virtual machines, including the virtualized network. In the last step, we connect to the controller VM. Next, Listing 5.2 bootstraps ansible on the controller machine and then deploys all required services and tools to the various machines in the testbed. After these steps, all machines are ready, and we can start running experiments.

Chain	Average ICMP response	Packet loss	Average SYN response	SYN packet loss
Direct	0.191ms	0%	4.5ms	0%
Firewall	0.343ms	0%	3.8ms	0%
DPS	0.194ms	0%	4.7ms	0%
IDPS	0.340ms	0%	4.4ms	0%

Table 5.3: Latency and packet loss for single security functions with a benign workload.

5.2.2 Single Security Function Performance

Before evaluating the impact of SSFC ordering on performance, it is necessary to evaluate the single security functions. With this knowledge, we can then assert the impact of the SSFCs ordering.

5.2.2.1 Benign Workloads

At first, it is necessary to build a baseline of how the security functions behave under benign load. It is possible to make statements on the differences in behavior under attack using that information.

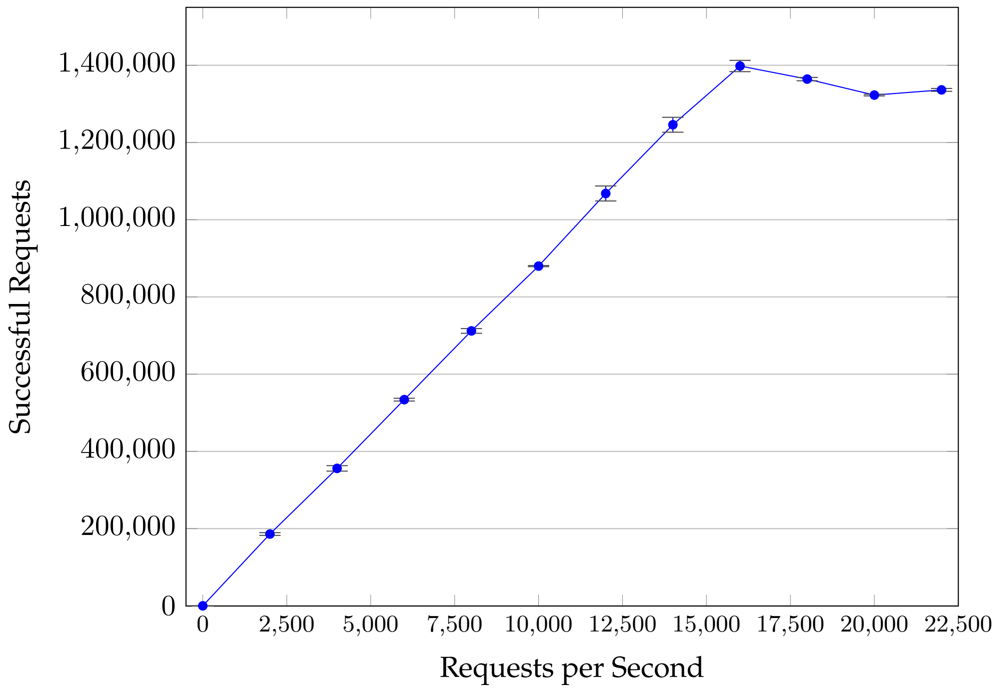


Figure 5.6: Successful requests without any security functions.

Baseline without a Security Function

We require a baseline that establishes a realistic maximum performance that is attainable by the service host. To this end, we query the service host with a varying benign load. Thereby, we determine the maximum amount of HTTP requests per second the service can handle.

Figure 5.6 visualizes the results for this experiment. Table B.1 lists the values for the figure. The results show that the service scales linearly and beginning with 16 000 requests per second the number of successful requests stalls. There is even a small decrease in throughput afterward. This decrease is probably attributable to, e.g., queuing, swapping, and context switching effects. At that point, the target service has reached its limit. Since the two servers directly connect to the same switch, the first data row in Table 5.3 shows very low latency and no packet loss. These results serve as a baseline for evaluating how the appliances impact the latency and data loss.

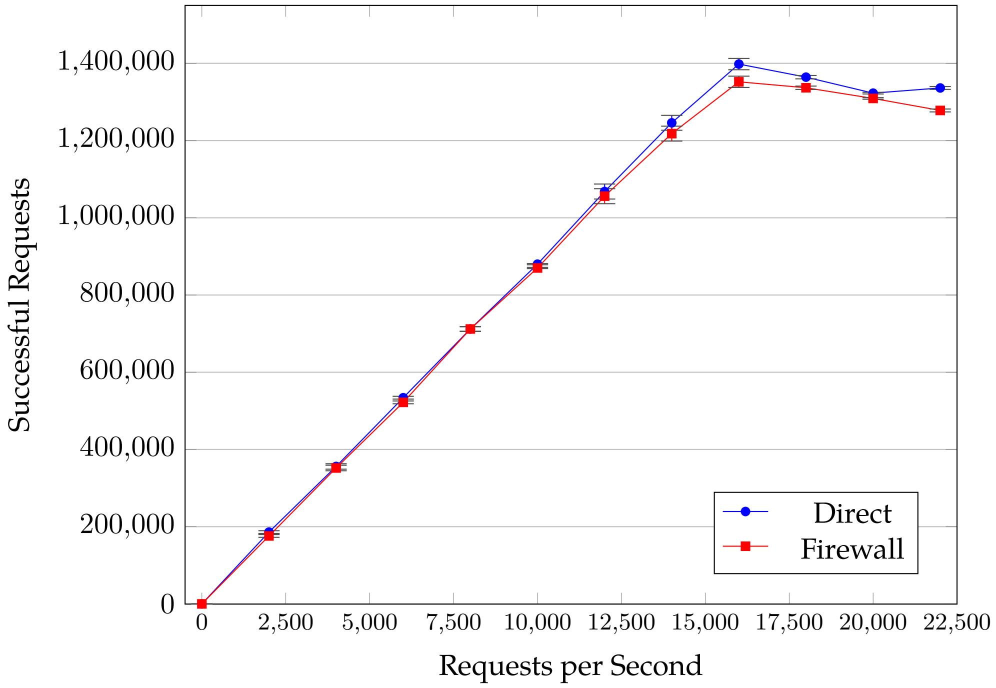


Figure 5.7: Successful requests with only a firewall enabled.

Firewall

We repeat the same experiment as for the direct connection with a firewall between the client and the service host. In the evaluation environment (Figure 5.5), this means adding the SDN rules that route traffic from the client to the server via `SW2`, `SW3`, and the firewall connected to `SW3`. Table A.1 lists the necessary SDN rules for all involved switches.

Figure 5.7 visualizes the results for the firewall (values from Table B.2). Regarding the results, the firewall has little effect on the number of successful requests. Again, the number grows linearly with the workload up to 16 000 requests per second, and then stalls out and slowly drops when in overload. As Table 5.3 shows, the firewall has a small impact on the ICMP response. A significant factor in this increase is the routing via three different switches. The SYN response even drops slightly.

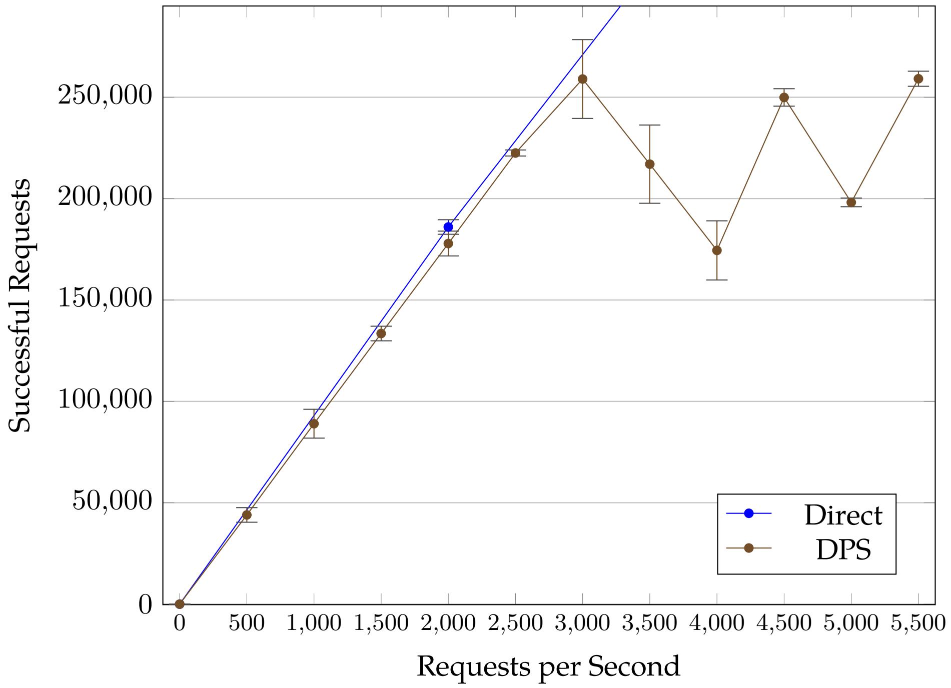


Figure 5.8: Successful requests with only a DPS enabled.

DDoS Protection System (DPS)

Again, we repeat the experiment with a DPS between the client and the service host. In the evaluation environment (Figure 5.5), this means adding the SDN rules that route traffic from the client to the server via `SW2`, and the DPS connected to `SW2`. Table A.2 lists the necessary SDN rules for all involved switches.

Figure 5.8 shows the results for the DPS (values from Table B.4). Regarding the results, the DPS has a significant effect on the number of successful requests. In the beginning, the number grows linearly. Nevertheless, unlike the direct connection and the firewall, the limit is reached at 3 000 requests per second, and then stalls out and drops significantly when in overload. For higher loads, the throughput zigzags afterward but does not exceed its previous maximum. As Table 5.3 shows, the DPS has almost no impact on the ICMP response when comparing to the direct connection. This effect is attributable to only one necessary hop. The SYN response slightly increases. We expected such a small increase since the DPS performs additional actions during SYN handshakes, including a complex computation (see SYN Cookies in Section 2.3.2.1).

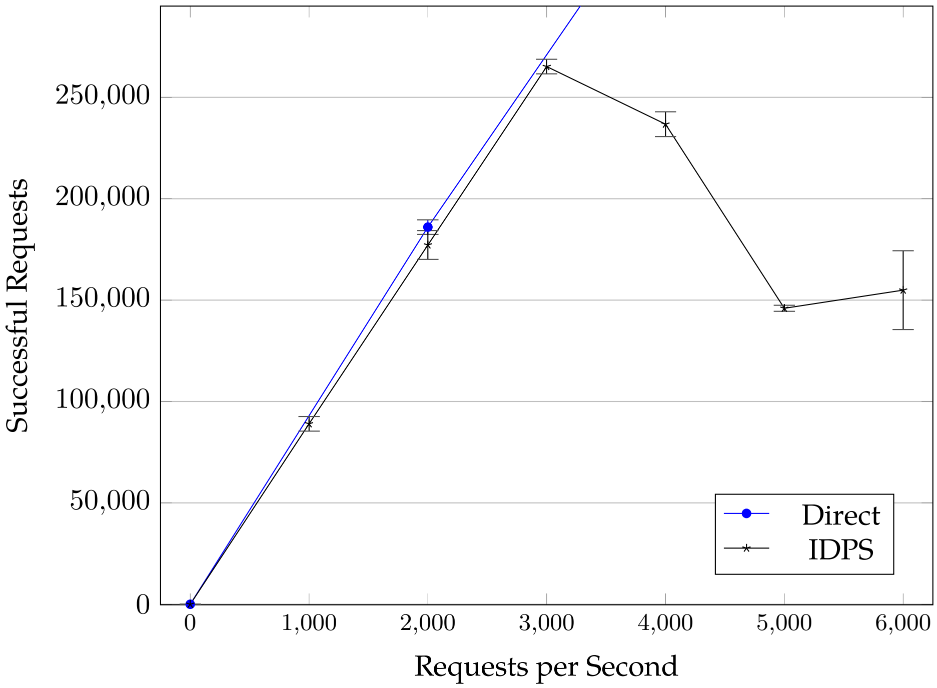


Figure 5.9: Successful requests with only an IDPS enabled.

Intrusion Detection and Prevention System (IDPS)

We performed the last iteration of our benign experiment using an IDPS between the client and the service host. In the evaluation environment (Figure 5.5), this means adding the SDN rules that route traffic from the client to the server via `SW2`, `SW3`, and the IDPS connected to `SW3`. Table A.3 lists the necessary SDN rules for all involved switches.

Figure 5.9 shows the results for the IDPS (values from Table B.5). Regarding the results, the IDPS has a significant effect on the number of successful requests. Until its peak, the IDPS behaves similarly to the DPS. The number of handled requests grows linearly. Like the DPS, the IDPS reaches its limit at 3 000 requests per second, and then stalls out and drops significantly when in overload. Unlike the DPS, the performance does not zigzag back up but stays low. As Table 5.3 shows, the IDPS has some impact on the ICMP response. The visible increase is similar to the increase in the latencies for the firewall. This similarity strengthens the assumption that the number of hops is significant. Like the firewall, the IDPS requires two hops from the client as well as to the service host. The SYN response time does not change significantly.

5.2.2.2 Malicious Workloads

RQ 5.2a Evaluating the performance of the service and the various security functions under benign workloads gives a reliable performance baseline for comparison with attack workload scenarios. In this section, we put the various functions under attack (attacks are matching the function that defends against them) using malicious workloads and evaluate their performance.

Firewall (UDP Flood)

As a first benchmark for the firewall, we perform a UDP flood attack. The firewall rules (see Listing A.2) block all received UDP traffic, and the SDN rules from Table A.1 take care of diverting the traffic via the firewall. We scale the UDP flood attack in steps of 500 MBit/s up to 5 000 MBit/s. To assert the QoS for the benign packets, we perform 2 000 benign HTTP requests per second for one minute. We measure metrics – other than the throughput – at 500 MBit/s flood strength.

We visualize the results for the successful HTTP requests under the UDP flood in Figure 5.10 using the data from Table B.6 Up until 500 MBit/s both the direct and the protected system perform at peak efficiency. From thereon, for both systems, the number of successful requests drops. The firewall's throughput drops faster than the unprotected system's throughput. At the

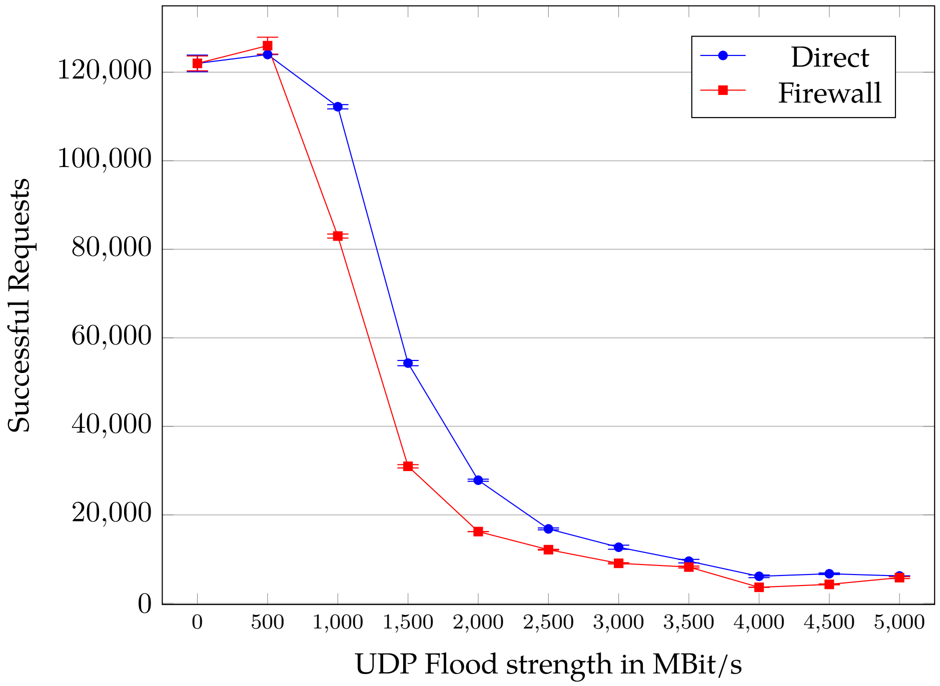


Figure 5.10: Successful requests during a UDP flood attack with direct connection or only the firewall enabled.

worst point, the firewall can only serve 3.6% of requests, and the unprotected system manages only 5.0%.

Chain	Average ICMP response	Packet loss	Average SYN response	SYN packet loss
Direct	0.679ms	0%	4.2ms	0%
Firewall	0.229ms	0%	4.2ms	0%

Table 5.4: Latency and packet loss during a UDP flood with and without the firewall enabled.

For the other metrics, as seen in Table 5.4, we observe that the SYN response is not affected by the protection. However, when using the firewall, the ICMP response is reduced by two thirds. A factor to this might be that the server does not have to handle the UDP requests and can focus on other requests from the client.

Firewall (HTTP Flood)

As the other benchmark for the firewall, we perform an HTTP flood attack. Section 2.2.1.2 gives more details on this attack type. The firewall blocks HTTP requests from malicious sources (see Listing A.2). We continue to use the SDN rules from Table A.1 for diverting the traffic via the firewall. We scale the HTTP flood attack in steps of 1 000 requests per second up to 14 000 requests per second. Again, we perform 2 000 benign HTTP requests per second for one minute to assert the performance for benign workloads. We measure metrics – other than the throughput – at a flood strength of 5 000 requests per second.

Figure 5.11 shows the successful requests during the HTTP attack using the data from Table B.7 The firewall-protected system always stays at peak efficiency. However, the unprotected system can hold this state only up to 1 000 requests per second and then is no longer able to perform all requests. After a first steep drop at 2 000 requests per seconds, the throughput continues to drop until it flattens out at 14.9% of handled requests from an attacking load of 10 000 requests per second and above.

Table 5.5 shows that the firewall increases the ICMP response time. A possible factor here is the additional hops to the firewall.

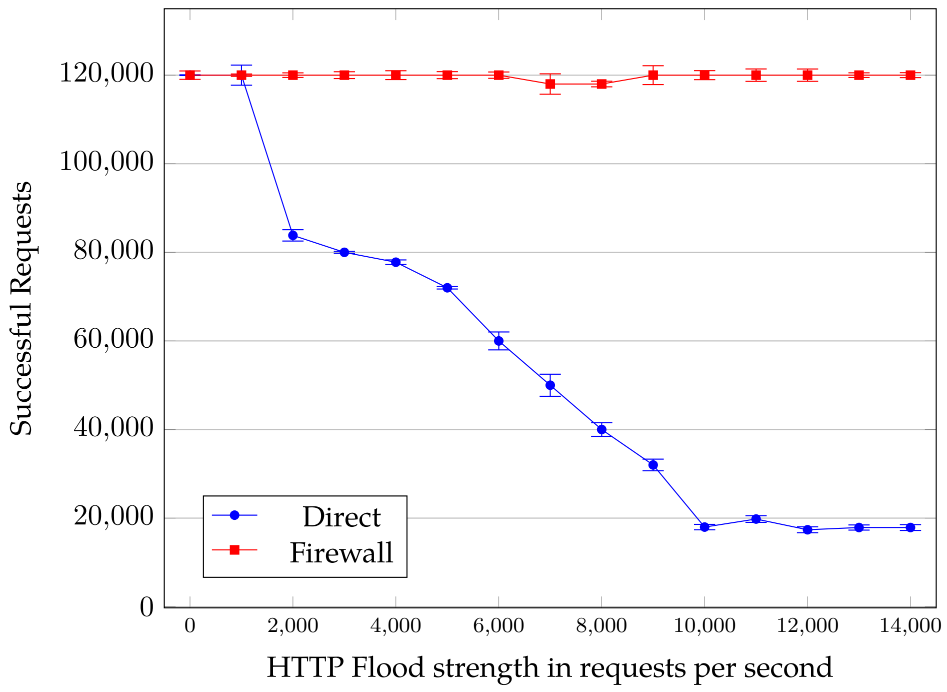


Figure 5.11: Successful requests during an HTTP flood attack with direct connection or only the firewall enabled.

Chain	Average ICMP response	Packet loss
Direct	0.154ms	0%
Firewall	0.266ms	0%

Table 5.5: Latency and packet loss during an HTTP flood with and without the firewall enabled.

DDoS Protection System (DPS) (SYN Flood)

We perform a SYN Flood attack (for the concept of this attack see Section 2.2.1.1) to assert the performance of the DPS under a malicious workload. The rules from Table A.2 redirect the traffic via the DPS. For each run, the amount of SYN packets is increased by 500 Mbit/s, up to 6 500 Mbit/s. We generate a load of 2000 benign HTTP requests per second for one minute to evaluate the successful requests during a SYN flood. We measure metrics other than the successful requests at 5 000 MBit/s attack load.

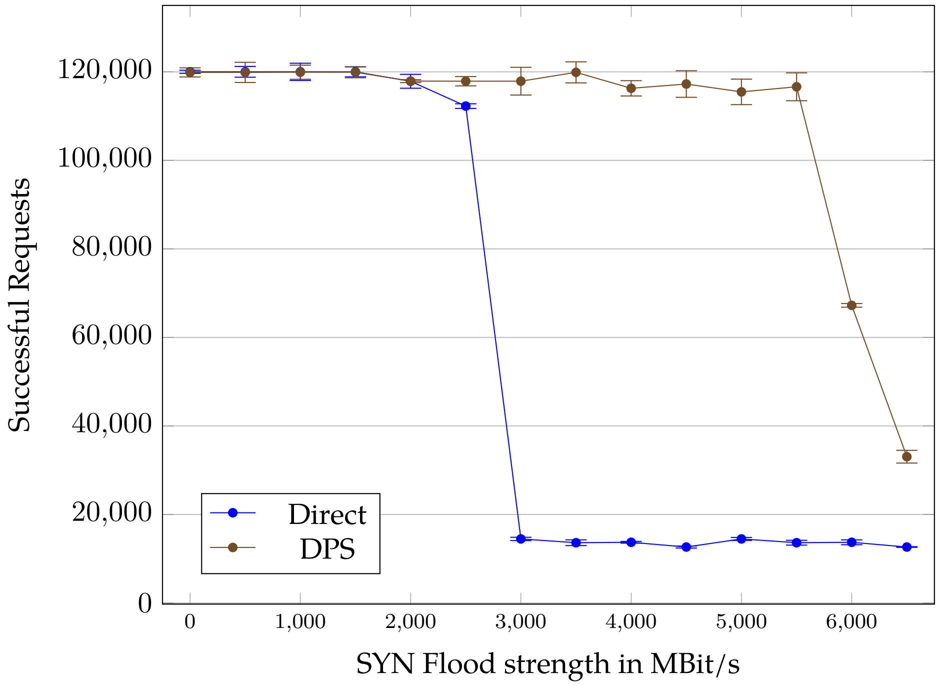


Figure 5.12: Successful requests during a SYN flood attack with direct connection or only the DPS enabled.

Figure 5.12 shows the throughput results with and without the DPS enabled (values from Table B.8). The unprotected service operates at peak efficiency for up to 2 000 MBit/s, but from 2 500 MBit/s on, it is no longer able to handle all benign requests. At 3 000 MBit/s, the service drops to around 12,3% of successful requests. With the DPS enabled, full throughput is possible until 5 500 MBit/s (2.75 times the load the direct service can handle). After that point, the number of successful requests drops. However, that drop is less steep than for the direct service, and even at 6 500 MBit/s, the DPS-protected system still completes around 2.5 times as many requests as the unprotected system.

Chain	Average ICMP response	Packet loss	Average SYN response	SYN packet loss
Direct	0.108ms	0%	3.9ms	0%
DPS	0.146ms	0%	1.1ms	0%

Table 5.6: Latency and packet loss during a SYN flood with and without the DPS enabled.

When taking a look at further stats presented in Table 5.6, the effect of the DPS is a slight increase in the ICMP response time. However, the SYN response is much faster with the DPS enabled than for the unprotected service (reduced by 71,8%).

Intrusion Detection and Prevention System (IDPS) (Intrusion Flood)

The last attack is an intrusion flood. Section 2.2.2 explains the working of this attack type. To create the flood, we use UDP packets containing a signature that matches the IDPS rules from Appendix A.1. The SDN rules from Table A.3 take care of diverting the traffic via the IDPS. We perform the intrusion flood for up to 5 000 MBit/s scaling in steps of 500 MBit/s. We measure further metrics at an attacking load of 1 000 MBit/s.

As seen in Figure 5.13 (with data from Table B.9), both configurations keep peak efficiency only until 500 MBit/s. Afterward, for both configurations, the throughput drops significantly. The throughput drops faster for IDPS than for the unprotected system. For higher loads, both systems drop to 5%.

Chain	Average ICMP response	Packet loss	Average SYN response	SYN packet loss
Direct	3.973ms	0%	8.5ms	0%
IDPS	35ms	32%	39ms	11%

Table 5.7: Latency and packet loss during an intrusion flood with and without the IDPS enabled.

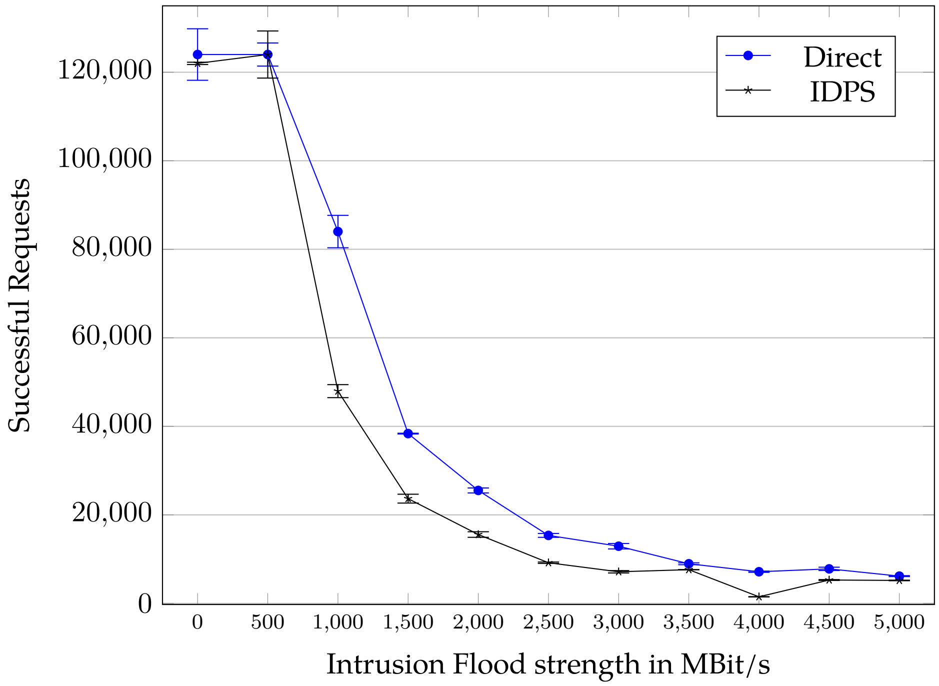


Figure 5.13: Successful requests during an intrusion flood attack with direct connection or only the IDPS enabled.

Table 5.7 shows a massive increase by one order of magnitude for the ICMP response time when using the IDPS. The SYN response time quadruples. Also, the packet loss rates grow from 0% to 32% (ICMP) and 11% (SYN). This effect fits the patterns we analyzed when optimizing an IDS in Section 4.1.

It is worth mentioning that the intrusion flood is also a UDP flood since it uses UDP packets as a matter of transport. Thus, the performance of the unprotected service correlates with its UDP flood performance. Hence, it is little surprise that both floods have a similar impact on the service.

5.2.3 Security Service Function Chain Performance

RQ 5.2b After evaluating the single security functions in the previous section, we now proceed to combinations of security functions. Therefore, we use the same attacks as before and combine pairs of security functions for each attack and switch their ordering for comparison.

The scope of these experiments is not limited to evaluate the impact on Quality of Experience (QoE) relevant metrics (throughput, latencies, and package loss) but also includes the CPU load for an approximation on the resulting computing resource demands. Therefore, we measured the CPU load throughout the experiment and for the various SSFC orders. We not only record the load during the experiment but also during the ramp-up and ramp-down phases. Three CPU load metrics are relevant for our experiments:

`user` **Mode:** This load level tells us the amount of the time the processor is spending running user-space processes. A user-space program is any process that does not belong to the kernel. Most programs associated with the desktop are all user space processes. For most applications, the CPU should spend the majority of its time running user space processes.

`system` **Mode:** This metric describes the sum of time for which the CPU has been operating the kernel. The Linux kernel manages both the processes and the system resources. If no user-space process wants anything from the system, such as when it has to assign resources, execute any I/O operations, or fork new child processes, the kernel operates. The scheduler that decides which operation will operate next is an integral part of the kernel. The amount of time spent in the kernel should be as low as possible. However, some applications partially or wholly run in system mode. Examples for this behavior are the filter modules used by firewalls.

`softirq` **Mode:** This statistic shows the amount of time spent on handling software interrupts. While hardware interrupts come from various peripherals like disks and network interfaces, software interrupts come from processes running on the system. Hardware interrupts cause the CPU to interrupt (hence the name) its current task and first handle the interrupt. A software interrupt occurs at the kernel level, not at the CPU level.

We measure the CPU load levels for the IDPS and the firewall. The DPS uses busy-waiting (not using sleep) to reduce latency and, therefore, always runs at 100% load. On the charts on the following pages, a load of 0.5 would mean that one CPU thread spends 50% of its time in this state. The maximum for our setup is 8.0 (four cores with two threads each). If the load exceeds this value, the system is overbooked. The IDPS is running single-threaded. Thus, it enters an overloaded state at a load level of more than one. In select cases, we specify loads in percent instead of the notation described above. There are some issues when comparing load levels. While it is safe to compare very high load levels, at low levels, CPU frequency scaling approaches might come into play. Thus, a load at 0.6 might be a load at the maximum or the minimum CPU clock rate. In our testbed, the minim clock rate is at 43% of the maximum clock rate. Therefore, we can only use the upper and lower boundary statements. E.g., for our CPUs, a load of 4.0 equals at most 50% and at least 21.5% (50% of the 43%) of the supported load.

5.2.3.1 UDP Flood

The first attack to benchmark SSFC orders is the UDP flood attack. The device that protects against this attack is the firewall. Thus we use the firewall rules from before (see Listing A.2) to block all received UDP traffic. Again, we scale the UDP flood attack in steps of 500 MBit/s up to 5 000 MBit/s. As before, we perform 2 000 benign HTTP requests per second for one minute to assert the QoS for the benign packets. We measure metrics — other than the throughput — at 500 MBit/s flood strength.

Firewall $\longleftrightarrow$ IDPS

We compare two SSFC orders: (i) IDPS $\rightarrow$ Firewall, and (ii) Firewall $\rightarrow$ IDPS. To enforce these SSFC orders, we use the SDN rules from Table A.4. It is notable that the reordering between the two security functions only requires the modification of the flow tables of a single switch (`SW2`).

Figure 5.14 shows the performance of both SSFC orders using data from Table B.10. Without an attack, both SSFC orders can perform all requests. Already at only 500 MBit/s, the performance of the IDPS $\rightarrow$ Firewall SSFC order

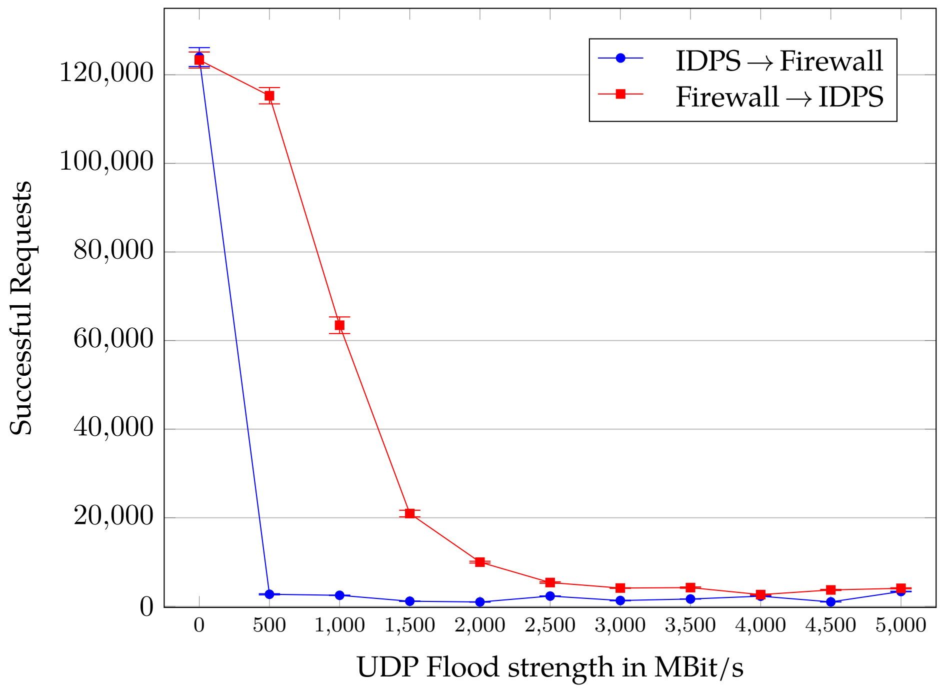


Figure 5.14: Successful requests during an UDP flood attack with the firewall and the IDPS enabled in different orders.

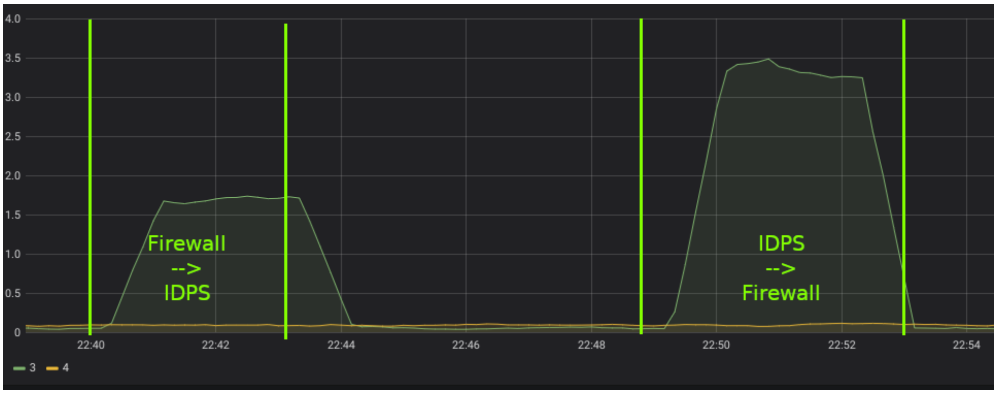
drops by 98% to 2 720 successful requests per second while the Firewall → IDPS SSFC order only loses 7%. From thereon, the performance of the IDPS-headed chain stays below 3 500 requests per second. In contrast, the firewall-headed chain only step-by-step loses performance, and only at 4 000 requests falls below the other SSFC order's limit. In general, the Firewall → IDPS SSFC order at any load level has a higher rate of successful requests per second than the IDPS → Firewall chain.

Chain	Average ICMP response	Packet loss	Average SYN response	SYN packet loss
IDPS → Firewall	350ms	27%	636ms	18%
Firewall → IDPS	0.355ms	0%	4.4ms	0%

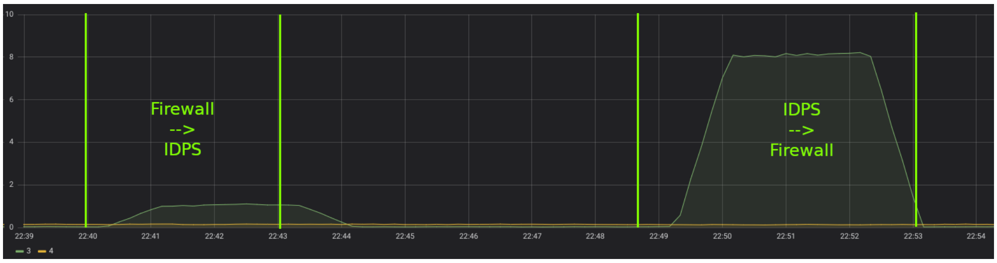
Table 5.8: Latency and packet loss during a UDP flood with the IDPS and the firewall enabled in different SSFC orders.

Table 5.8 confirms these findings of better performance when putting the firewall first. The IDPS → Firewall SSFC order results in almost a thousandfold of the ICMP response time of the other chain and two orders of magnitude increase in the SYN response. This chain also creates a massive packet loss compared to no loss for the firewall-headed SSFC order.

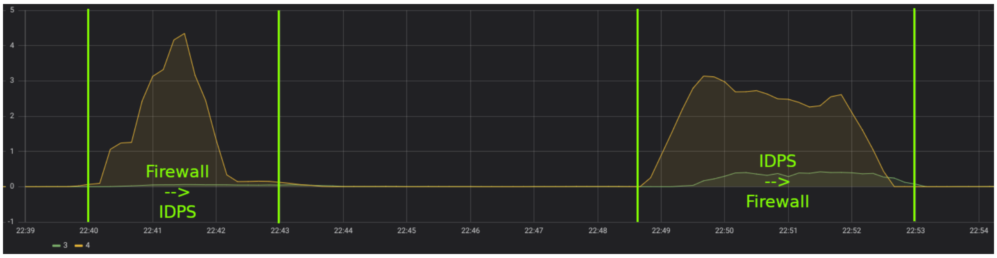
Considering the CPU load in Figure 5.15 shows that the load on the IDPS in `user` and `system` state is significantly smaller for the Firewall → IDPS SSFC order. As described before, the IDPS overloads at a load level higher than 1.0. The user load fulfills that criterion. Thus, the IDPS also is a bottleneck in the Firewall → IDPS chain. Furthermore, a significant `softirq` load is present for the IDPS-headed chain. Also, the IDPS load continues long after the attack termination, leading to the assumption that some packets still reside in the IDPS's internal queue. We confirmed that assumption using Wireshark. The firewall, on the other hand, does not encounter significant `user` and `system` state. However, the firewall creates much load in the `softirq` mode. The load is below the maximum of 8.0 for both SSFC orders. This observation leads to the assumption that the IDPS hinders the firewall. When putting the firewall first, the load peaks higher than for the other SSFC order. This effect is attributable to the higher throughput through the firewall in this scenario. Albeit, the load stays on longer, when putting the IDPS first. The IDPS's queuing behavior causes the packets to arrive later and more continuously at the firewall.



(**a**) CPU load in user state.



(**b**) CPU load in system state.



(**c**) CPU load in softirq state.

Figure 5.15: CPU load on the IDPS (3) and firewall (4) during a UDP flood with different SSFC orders.

5.2.3.2 HTTP Flood

The second benchmark is an HTTP flood attack. Section 2.2.1.2 describes this attack type. The firewall is the defending security function and blocks HTTP requests from the malicious sources (see Listing A.2). As for the standalone firewall, we scale the HTTP flood attack in steps of 1 000 requests per second up to 14 000 requests per second. Once more, we perform 2 000 benign HTTP requests per second for one minute to assert the performance for benign workloads. As before, we measure metrics – other than the throughput – at 5000 requests per second flood strength.

Firewall $\longleftrightarrow$ IDPS

We compare the same SSFC orders as for the UDP flood attack: (i) IDPS → Firewall, and (ii) Firewall → IDPS. We continue to use the SDN rules from Table A.4 introduced for the UDP flood for diverging the traffic via the IDPS and firewall and vice-versa.

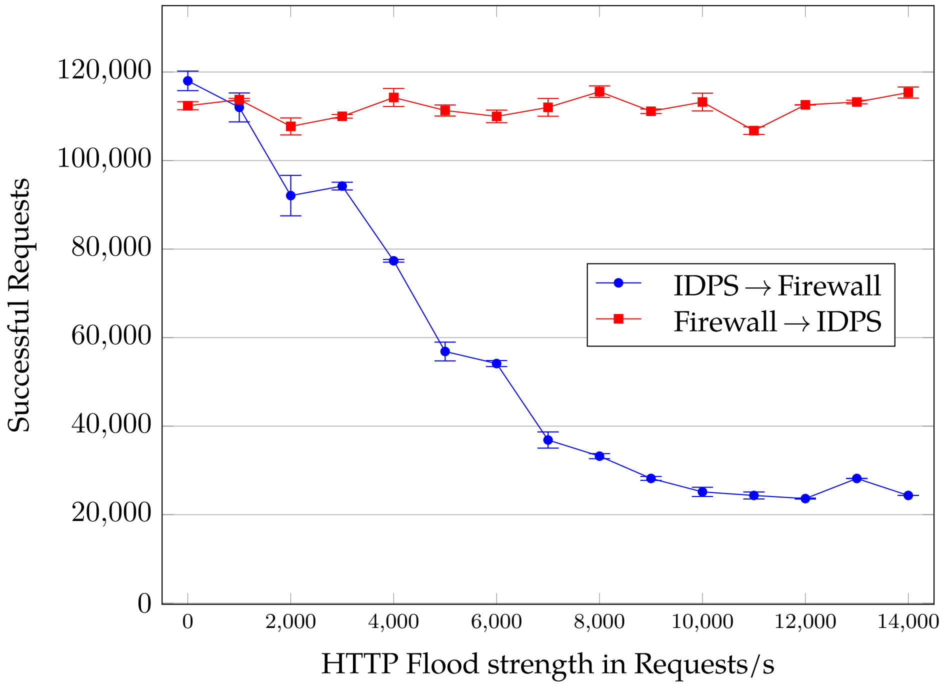


Figure 5.16: Successful requests during an HTTP flood attack with the firewall and the IDPS enabled in different SSFC orders.

Figure 5.16 presents the throughput results by visualizing the data from Table B.11. As for the UDP flood, both SSFC orders handle the benign workload and a small attack load of 1 000 requests per second well. At higher attack load levels, the number of successful benign requests drops for the IDPS → Firewall SSFC order. It continues to drop with higher load, and form 10 000 requests onwards settles above 20 000 successful requests. Meanwhile, the Firewall → IDPS chain is hardly affected by the attack load and remains close to the maximum attainable level and always stays significantly above the other order's level.

Chain	Average ICMP response	Packet loss
Firewall → IDPS	0.390ms	0%
IDPS → Firewall	0.403ms	0%

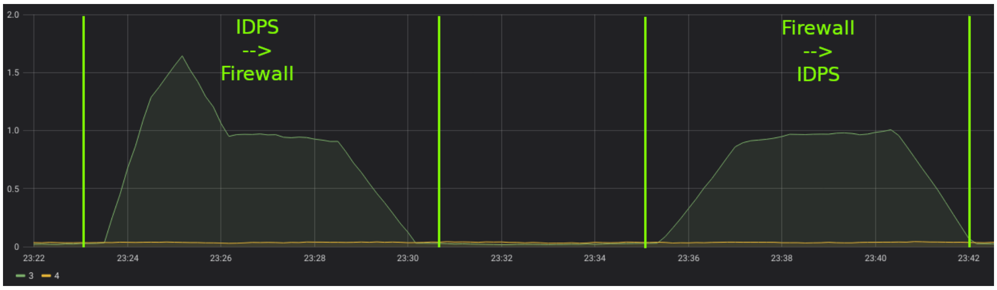
Table 5.9: Latency and packet loss during an HTTP flood with the firewall and the IDPS enabled in different SSFC orders.

Table 5.9 shows further metrics for the two orders. Unlike for the UDP flood, the IDPS → Firewall SSFC order does not generate a massive increase in latency or a packet loss. Thus, considering these values, both SSFC orders perform similarly.

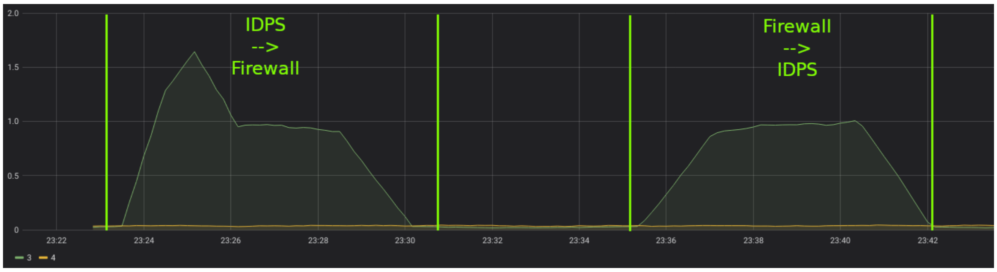
Figure 5.17 shows the different CPU load levels for both SSFC orders and both appliances. Again, the IDPS mainly creates `user` and `system` load. Here, both SSFC orders behave very similarly. In both cases, the IDPS reaches a load of 1.0 most of the time — thus, again suggesting it is the bottleneck. However, when putting the IDPS first, we observe a load peak at the beginning, suggesting an overload situation. The firewall creates a relatively small `softirq` load (below 0.8) at the beginning of the attack in both scenarios. For the other states and the rest of the time, it remains at shallow load levels. This behavior suggests massive reserves of the firewall for higher load levels.

Firewall ⟷ DPS

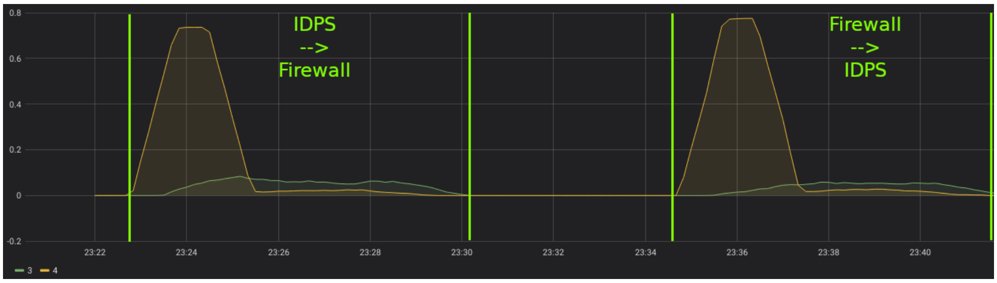
Within our setup, it was not possible to find a suitable HTTP flood generator for measuring the performance of the Firewall → DPS chains. HTTP Load Generator proved unsuitable for generating HTTP floods because it does not handle failing connections well. We used BoNeSi for the Firewall → IDPS measurements, but it caused unresolvable issues in combination with the DPS. Hence no measurements for theses chains are available.



(**a**) CPU load in user state.



(**b**) CPU load in system state.



(**c**) CPU load in softirq state.

Figure 5.17: CPU load on the IDPS (3) and firewall (4) during an HTTP flood with the IDPS and the firewall enabled in different SSFC orders.

5.2.3.3 SYN Flood

As a third benchmark, we perform a SYN Flood attack (for more information on this attack see Section 2.2.1.1). The DPS is the defending security function. For each run, we increase the SYN flood strength by 500 MBit/s, up to 6 500 MBit/s. We generate a load of 2000 benign HTTP requests per second for one minute to evaluate the successful requests during a SYN flood. We measure metrics other than the successful requests at 5 000 MBit/s attack load.

DPS ⟷ IDPS

We first compare the following two SSFC orders: (i) DPS → IDPS, and (ii) IDPS → DPS. We use the SDN rules from Table A.5 for both SSFC orders. Again, only a reconfiguration at `SW2` is necessary.

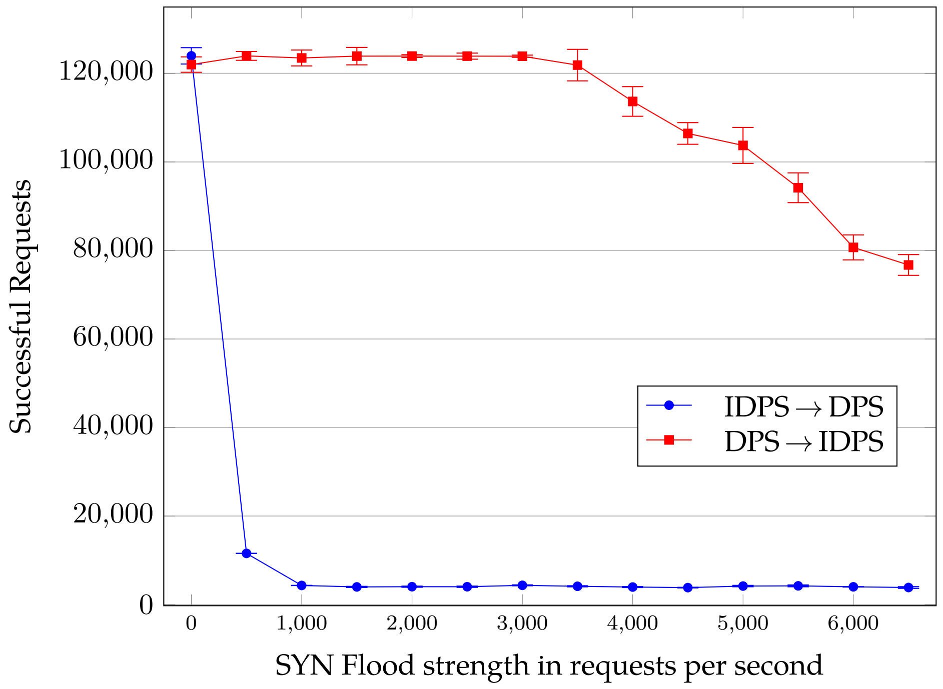


Figure 5.18: Successful requests during a SYN flood attack with the DPS and the IDPS enabled in different SSFC orders.

Figure 5.18 visualizes the successful benign requests (data from Table B.12). Without attack, both configurations work at peak performance. From 500 requests on, the results for the IDPS → DPS chain drop and reach their minimum of around 4 000 successful requests (a 97% drop) already at only 1 000 requests per second. For higher load levels, the results vary only slightly but do neither improve nor deteriorate. In contrast, the DPS → IDPS chain keeps up full throughput until 3 500 requests per second. From thereon, the results slowly drop almost linearly. At the maximum load in our experiment, this chain still serves 76 719 requests or 62%.

Chain	Average ICMP response	Packet loss	Average SYN response	SYN packet loss
DPS → IDPS	0.285ms	0%	0.9ms	0%
IDPS → DPS	223ms	26%	233ms	36%

Table 5.10: Latency and packet loss during a SYN flood with the DPS and the IDPS enabled in different SSFC orders.

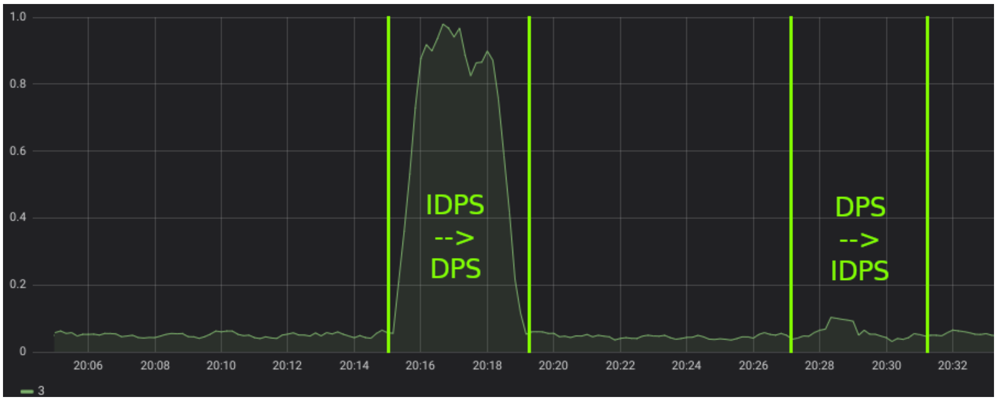
When we look at the other metrics, as presented in Table 5.10, vast differences become visible. The IDPS → DPS SSFC order increases both response times by three orders of magnitude. Also, it introduces significant packet losses at 26% (ICMP) and 36% (SYN). These losses do not appear for the DPS → IDPS SSFC order.

Figure 5.19 illustrates the CPU load for the IDPS during attacks for both orders. As stated before, the load for the DPS always remains at 1.0 due to busy-waiting. Therefore, the figure does not show it. The load in `user` state for the IDPS is close to 1.0 for the IDPS → DPS SSFC order but very small for the reversed SSFC order. Similarly, with the IDPS put first, the `system` load exceeds the IDPS's 1.0 limit and even the 8.0 limit of the used machine. This high load level indicates a massive overload situation. When reversing the SSFC order, no significant CPU load is present. For the IDPS-headed chain, there is a medium load up to around 0.25 in the `softirq` state. No such load is present for the reversed SSFC order. In general, the DPS → IDPS SSFC order almost eliminates all load on the IDPS.

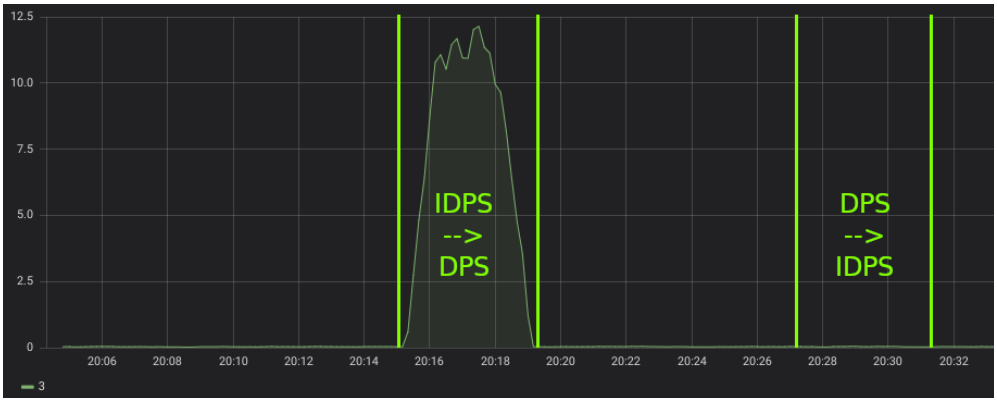
DPS ⟷ Firewall

For this attack, we benchmark two more SSFC orders: (i) DPS → Firewall, and (ii) Firewall → DPS. We use the SDN rules from Table A.6 for both orders. Once more, only a reconfiguration at `SW2` is necessary.

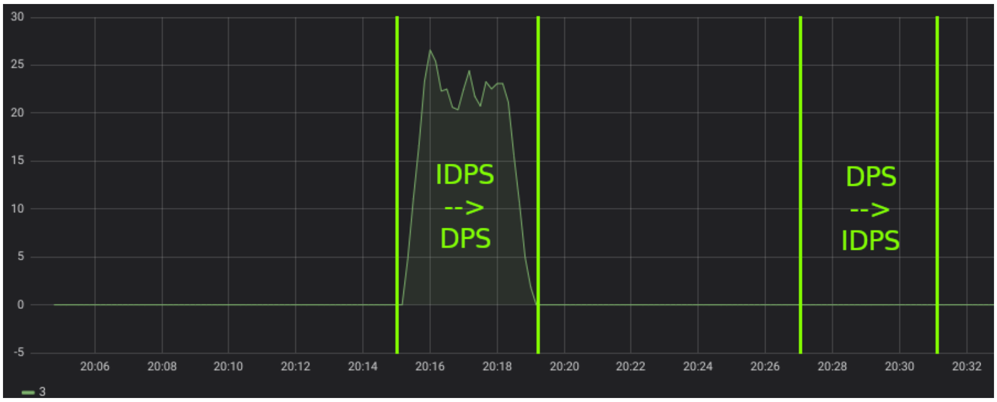
Figure 5.20 presents the number of successful requests for both SSFC orders based on the data from Table B.13. Both orders keep an optimal success rate



(**a**) CPU load in user state.



(**b**) CPU load in system state.



(**c**) CPU load in softirq state in percent.

Figure 5.19: CPU load on the IDPS during a SYN flood with the DPS and the IDPS enabled in different SSFC orders.

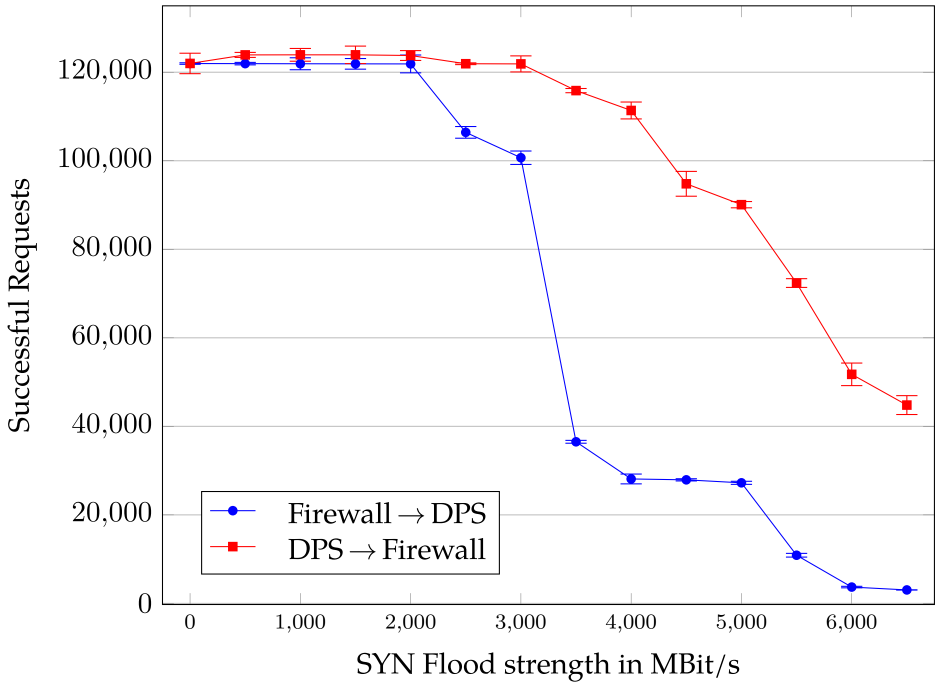


Figure 5.20: Successful requests during a SYN flood attack with the DPS and the firewall enabled in different orders.

until 2 000 MBit/s. After that, the firewall-headed SSFC order slowly drops to around 100 000 successful requests at 3 000 MBit/s and then suddenly at once drops bellow 40 000 successful requests at 3 500 MBit/s. Then again at 4 000 MBit/s slightly drops below 30 000 successful requests where it stays until 5 000 MBit/s. Then, it drops again to only 10 881 successful requests at 5 500 MBit/s. At 6 000 MBit/s, it further drops to 3 688 then stays at a similar level. The DPS → Firewall SSFC order continues at maximum performance until 3 000 MBit/s. Then, it continuously drops but stays significantly above the inverse chain's performance to 44 803 successful requests at 6 500 MBit/s (14.7 times the result for the inverse SSFC order).

Chain	Average ICMP response	Packet loss	Average SYN response	SYN packet loss
DPS → Firewall	0.288ms	0%	1.0ms	0%
Firewall → DPS	0.242ms	0%	4.5ms	0%

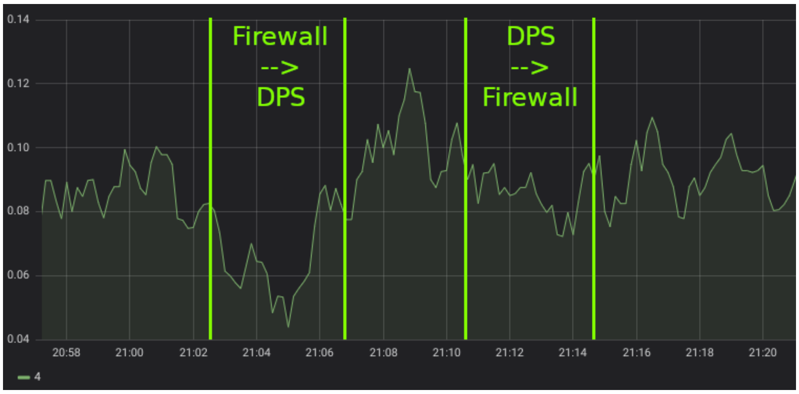
Table 5.11: Latency and packet loss during a SYN flood with the DPS and the firewall enabled in different SSFC orders.

Table 5.11 shows further metrics for both orders. The results are a little more diverse than for the previous attacks and combinations. The Firewall → DPS SSFC order offers a faster ICMP response while the DPS → Firewall SSFC order yields a faster SYN response. The relative difference for the SYN response is more substantial than for the ICMP response. Both configurations do not yield any packet losses.

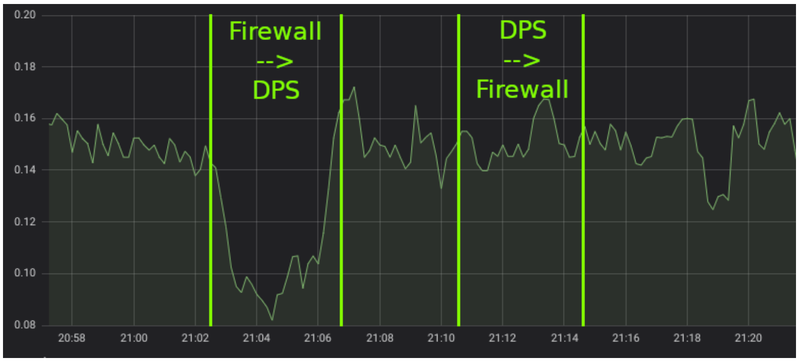
Figure 5.21 shows the CPU load for the firewall (the DPS once more not shown due to busy-waiting). The Firewall → DPS SSFC order impacts the load during the attack. The background load (e.g., OS operations or filesystem journaling) in the `user` and `system` states is forced out by the actual load by the firewall application. This application load appears in the `softirq` level, where the CPU spends 100% of its time. When reversing the SSFC order, no noticeable load shows in the `softirq` state, and the background loads remain in the `user` and `system` state. Thus, this SSFC order eliminates all load on the firewall.

5.2.3.4 Intrusion Flood

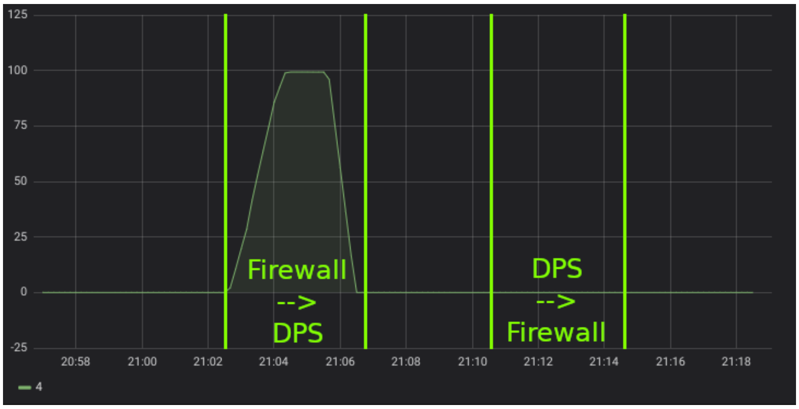
The fourth and last attack is an intrusion flood. We give additional information on this attack in Section 2.2.2. We use UDP packets containing a signature that



(a) CPU load in user state.



(b) CPU load in system state.



(c) CPU load in softirq state.

Figure 5.21: CPU load on the firewall during a SYN flood with the IDPS and the firewall enabled in different SSFC orders.

matches the IDPS rules from Appendix A.1 to create the flood. We perform the intrusion flood for up to 5 000 MBit/s scaling in steps of 500 MBit/s. We measure further metrics at an attacking load of 1 000 MBit/s.

Firewall $\longleftrightarrow$ IDPS

We compare two SSFC orders: (i) IDPS $\rightarrow$ Firewall, and (ii) Firewall $\rightarrow$ IDPS. We once more use the SDN rules from Table A.4 introduced for the UDP flood for diverging the traffic via the IDPS and firewall and vice-versa.

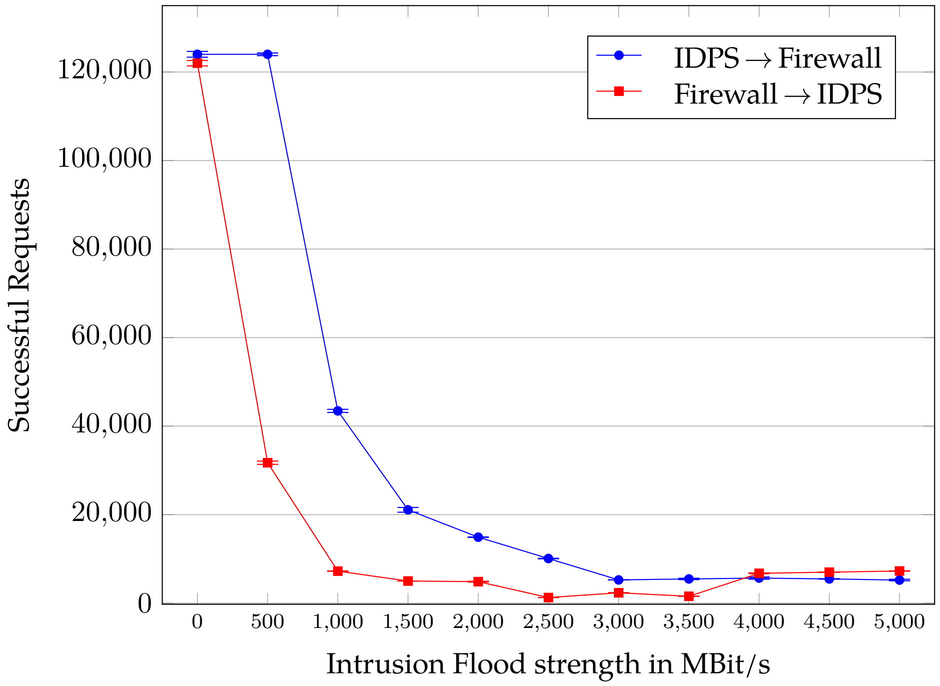


Figure 5.22: Successful requests during an intrusion flood attack with the firewall and the IDPS enabled in different SSFC orders.

Figure 5.22 shows (using the data from Table B.14) the number of successful requests for both chain orders. Both SSFC orders can fully satisfy with only the benign workload enabled. However, already at a flood strength of 500 MBit/s, the Firewall $\rightarrow$ IDPS chain drops to 31 732 successful requests. At 1 000 MBit/s flood strength, this chain further drops to 7 238 successful requests and from thereon stays at similar or lower levels. The reverse chain's performance drops later, starting at 1 000 MBit/s with a drop to 43 442 MBit/s. It then continues to fall slowly and finally aligns with the firewall-headed chains throughput at

4 000 MBit/s. Between the beginning of attacks at 500 MBit/s and the alignment, the IDPS → Firewall SSFC order outperforms the other chain.

Chain	Average ICMP response	Packet loss	Average SYN response	SYN packet loss
IDPS → Firewall	36ms	30%	37ms	34%
Firewall → IDPS	16ms	0%	17.9ms	0%

Table 5.12: Latency and packet loss during an IDS flood with the IDPS and the firewall enabled in different SSFC orders.

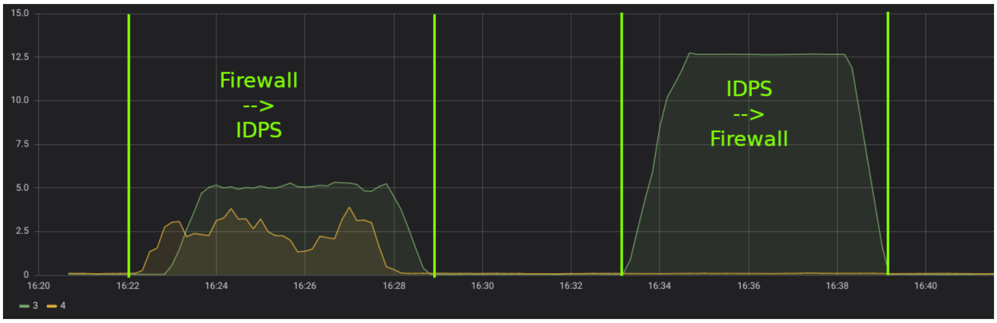
Looking at the further metrics in Table 5.12 with the previous impressions in mind yields an unexpected result. Notably, the IDPS → Firewall SSFC order doubles the response time compared to its counterpart. It also introduces a packet loss rate of about one-third of packets. This result surprises, since the higher throughput of the IDPS-headed chain, did not hint at this behavior. However, this shows that a way of getting higher throughput might lie in accepting packet losses.

Figure 5.23 shows the CPU usage during the experiments. Putting the firewall first creates `user` and `system` load for both systems. While the firewall is not in an extremely high load situation, the IDPS is in overload. Changing the SSFC order results in taking away the load from the firewall and the `system` load from the IDPS but heavily overloads the IDPS. The firewall spends most of its time the `softirq` state, when it is first in the chain. However, when the IDPS heads the chain, only a small peak appears at the beginning of the experiment.

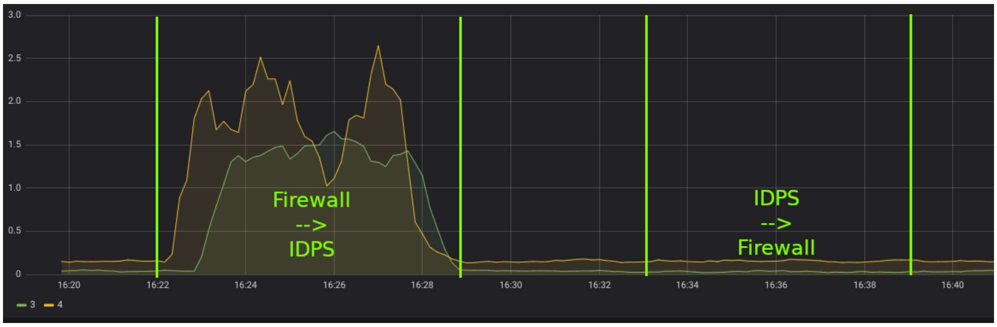
5.2.4 Discussion

Section 5.2.2.1 shows that even under benign workloads, the different security functions perform with significant differences. While the firewall can protect a service without reducing the throughput, the DPS and the IDPS reach their limits far before the protected service. Also, both systems (the IDPS more than than the DPS) show that their performance can further drop when the load increases after reaching their peak performance. The probable cause for the only significant impact on the latency between the client and the service is the number of necessary hops. Also, the DPS slightly increases the SYN response time.

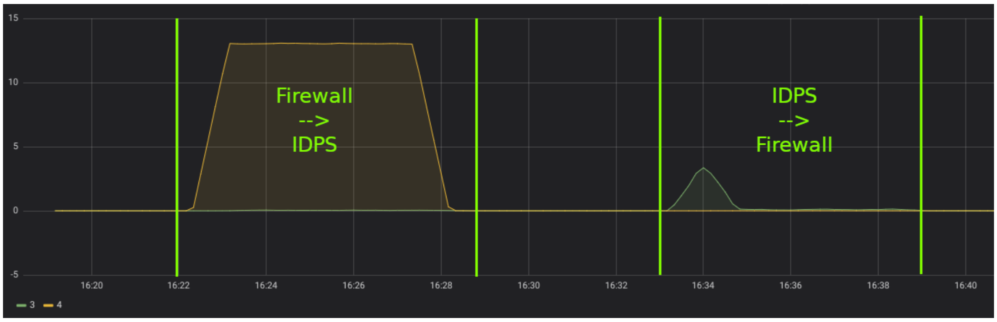
When adding the attacks in Section 5.2.2.2, other patterns emerge. Protecting the service against a UDP flood reduces the throughput while only yielding a



(**a**) CPU load in user state.



(**b**) CPU load in system state.



(**c**) CPU load in softirq state.

Figure 5.23: CPU load on the IDPS (3) and the firewall (4) during an intrusion flood with the IDPS and the firewall enabled in different orders.

small improvement in ICMP response time. This behavior shows that adding security functions that protect against vulnerabilities that are not present (the server does not listen to UDP packets) can harm performance. The same firewall can massively increase performance when protecting against the "right" attack — in this case, an HTTP flood with only a small cost in ICMP response time. A similar behavior presents itself when protecting against a SYN flood with the DPS. While the firewall can perform faster than our attack generator, the DPS extends the point where the throughput caves in by almost a factor of three. Unlike the DPS and the firewall that protect against DoS attacks, the IDPS protects against intrusion attacks. For this protection, DPI and, therefore, a massive amount of computing resources is necessary. Thus, as expected, the IDPS reduces the system's total performance.

The single function workloads show massive differences between the functions under benign as well as malicious loads. Thus, even without regarding the ordering, these differences require significant considerations when coming to scaling and placement.

Section 5.2.3 confirms our assumption that the order of security functions inside an SSFC has a significant impact on SSFC performance. When considering the throughput, we see different behaviors when comparing different attacks. Those behaviors share one commonality: placing the security function that defends against the attack first yields the most successful benign requests. In some cases, the SSFC order significantly prolongs the load level at which performance drops and slows the drop. Still, at some point, both SSFC orders converge to similar results. This convergence is the case for the UDP flood and the intrusion flood attacks. The results for the SYN flood converge slower than before and are still at high multitudes of the inverse SSFC orders performance when we reach the limit of our attack generator.

Additionally, the SYN flood also shows that the second function in the chain is relevant and yields better performance when using the firewall rather than the IDPS. The most significant difference is visible for the HTTP flood attack. Putting the firewall first consistently yields optimal performance, while the inverse order drops.

When analyzing further statistics, like ICMP and SYN response times and the respective packet loss, we observe that in many cases (UDP flood and SYN flood for IDPS and DPS), the order with the higher throughput yields the better metrics. For the HTTP flood and the SYN flood for firewall and DPS, only smaller differences occur. Last, for the intrusion flood, the order with the higher throughput is also the SSFC order with the worse other metrics.

The ordering also affects the CPU load on the security function servers. In general, the order with the throughput yields the lower load level – once more except for intrusion flood, where the IDPS load rises. For all scenarios, this change of order removes or reduces the load level on the non-defending security function.

Thus, we have shown that the SSFC order has a significant effect on the throughput, and also other metrics and the CPU load. For the selected attack combinations, we have also found that there is no optimal SSFC order for all attacks. While for HTTP and UDP floods, the firewall performs best before the IDPS, the reversed chain is superior during an intrusion flood. In general, putting the security function dedicated to protecting against the current attack first, yields the best results. Therefore, we require different SSFC orders depending on the current attack state of the system. This finding confirms our claim that dynamic SSFC reordering can improve the performance of SSFCs. We will follow the realization of this concept in the following sections.

5.3 Performance Modeling for Reordering Decision

In the previous parts of this chapter, we focused on the general idea of dynamic SSFC reordering and the impact of the SSFC order on the performance. Here, we will further focus on how to come to a precise model and a reasonable decision-making approach — especially for more complex systems. This model can later be an input to a dynamic SSFC reordering framework.

First, we take a look at how to model single security functions based on the incoming traffic. We then use this knowledge and combine multiple security function models into an SSFC model. Last, we discuss approaches to decision-making.

5.3.1 Modeling Single Security Functions

We model individual functions depending on the traffic they process. Therefore, we first need a traffic model that models benign and malicious traffic classes. **RQ 5.3a**

5.3.1.1 Modeling Traffic

As shown in Section 5.2, different security functions show a different behavior to different types of traffic. Thus, we need a model that, on the one hand, takes benign traffic into account and, on the other hand, considers the various attack types. To this end, our model of the arrival rate must consider the content (e.g., relevant for DPI used by an IDPS) and the composition of the traffic. Therefore, we model the traffic as different workload classes. For every workload class, we record the rate of packets and the bandwidth used by this class. For the security functions, we benchmarked in Section 5.2 this would constitute at least the following workload classes:

(1) Benign requests and unprotected attacks: In our setup, this represents the benign HTTP queries, the ICMP timing packets, and the TCP SYN requests from the client to the server. While this is sufficient for modeling the security function performance, modeling an entire system, including the security service functions and the compute services might require dividing this class into multiple sub-classes. Also, we summarize all malicious packets, we do not have a security function for, in this class, since there is no way of discerning them from the benign packets.

(2) UDP packets: Since we do not use benign UDP packets, they are an unwanted workload class. In a more complex scenario with UDP servers, there would be a separation by port.

(3) Malicious SYN requests: We consider all SYN requests that do not follow up with a complete connection establishment, as malicious. This simplification might lead to a few misclassifications, but these are a) to the safe side, and b) failed connection establishments that are not intended by the client still create load on the DPS or fill the servers buffer without the DPS.

(4) Malicious HTTP requests: This class contains all HTTP requests that do not target a valid port or Uniform Resource Identifier (URI).

(5) Intrusion packets: All packets that match the IDPS rules and would cause harm at the server if not filtered out belong to this class.

For our modeling, we, therefore, have five workload classes with two pieces of information per class. We model the traffic composition for the link from the external network to the first security function, every connection between security functions, and the link from the last security function to the protected system. Section 5.3.2 shows how to derive the values at each of those points.

5.3.1.2 Security Function Modeling

With a model for the traffic, it is possible to model the behavior of the single security functions. We propose to apply architectural performance models to model security functions. Architectural models capture the semantics, allowing for a plain view on the security functions, in contrast to low-level stochastic formalisms. We model each security function as a software component. However, we also offer simplified approaches to model security functions. It is necessary to model three aspects: (i) the effect of the security functions on the traffic composition, (ii) the performance behavior of the security function, and (iii) tertiary effects like packet-loss.

Modeling the Effect on Traffic Composition

Based on the distribution of the input traffic of a security appliance, the corresponding output traffic can be derived. We define the distribution of the input/output traffic as $P_{in/out}(t_i)$ with $i \in [1, n]$ for n different types of traffic.

Exemplary, a security function which drops all packets of the traffic type k the output traffic looks as follows:

$$P_{out}(t_i) = \begin{cases} P_{in}(t_i)/(1 - P_{in}(t_k)) & \text{for } i \neq k \\ 0 & \text{for } i = k \end{cases}. \tag{5.1}$$

Figure 5.24 shows the behavior of such a security function that drops a single class of packets – visualized regarding the bandwidth.

For our example with the functions from Section 5.2 and the traffic types described above, for the firewall $k = 2 \vee k = 4$, for DPS $k = 3$, and the IDPS $k = 5$. As seen, a security function can match more than one traffic type.

The current model assumes that a function eliminates all malicious traffic of one or more traffic types. This model does not yet take other behavior (e.g., false positives) into account. We discuss that, along with further tertiary factors.

Modeling the Performance of the Security Function

For the function, the most precise modeling solution would be to use a full-blown model of the software component to model the function's performance behavior. Such models usually base on the functions source code or are extracted by heavy black-box testing. However, often neither the source code is available nor the time and resources to perform extensive black-box testing. Still, even when such a model is not possible, we see a minimal model for a security function, as shown in Figure 5.25.

The depicted model shows the security function as a software component of type `SecurityFunction`. This component has a *behavior description* that contains a `BranchAction`, and this `BranchAction` contains a default behavior description and a description for at least one traffic class. Equation (5.1) is such a `BranchAction` for the behavior regarding the traffic.

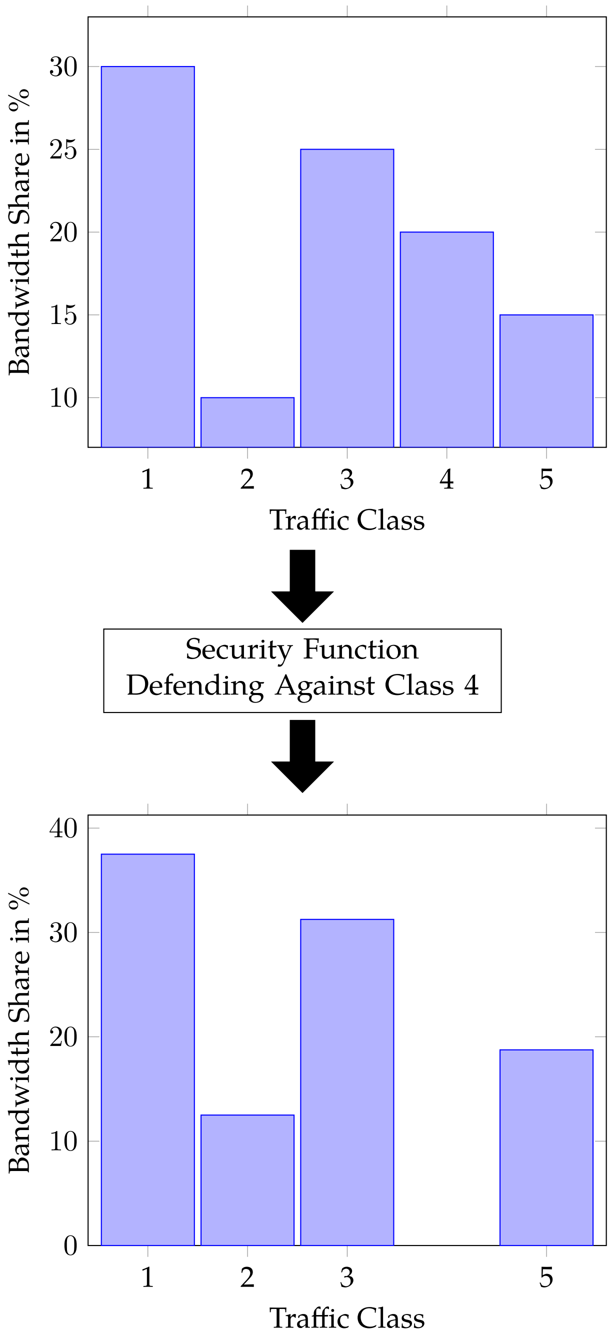


Figure 5.24: Sample model for a security function defending against malicious traffic of class four in a five class model.

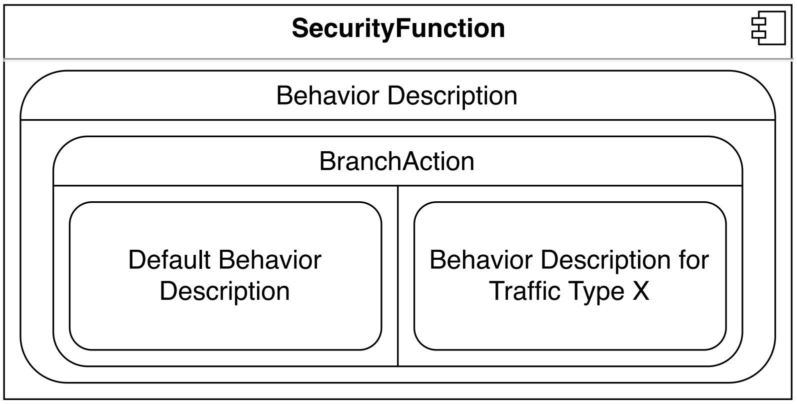

SecurityFunction
Behavior Description
BranchAction
Default Behavior Description
Behavior Description for Traffic Type X

Figure 5.25: Model for a single security function.

As expected, our security functions usually have different behavior for the traffic types against which they defend. However, they can also have diverging reactions to other classes. Furthermore, the behavior for the class that the functions protect against can have various subbranches.

Example: To illustrate, we describe the branch actions for the security functions evaluated and used in Section 5.2 regarding the five different traffic types before.

Firewall:

- The firewall analyzes the headers of all IP packets. This analysis creates a resource demand per packet. The resource demand is equal for all packets since the firewall only considers the fix-sized IP headers.
- The firewall discards UDP packets (Traffic Class 2). They do not create additional resource demand. The same is true for further non-TCP packets (e.g., the benign ICMP packets from Traffic Class 1).
- For all TCP packets a further distinction is necessary (here, our firewall is capable of working on OSI Layer 4 — the protocol layer):
  - For all TCP segments, the firewall inspects the segment's header. This inspection again creates additional resource demand per segment. If the header contains the wrong TCP destination port, the firewall drops the segment (Traffic Class 4). Still, this resource demand is constant per segment.

- For packets addressed to the correct port on the protected server again another distinction becomes necessary (thus, we have an OSI Layer 7 — application layer — firewall):
  * The firewall checks the HTTP request for blocked URIs. This check creates a resource demand to parse the resource passed in the HTTP request. This validation can take longer for longer URIs. However, this aspect is relatively small and, we, therefore, assume the resource demand as constant per HTTP request.

DPS:

- The DPS application listens to TCP segments. Therefore, we assume *no* resource demand for other traffic types (e.g., UDP packets from Traffic Class 2, and ICMP packets from Traffic Class 1).
- For TCP segments, the DPS makes a distinction based on enabled SYN and ACK flags. This distinction creates a constant resource demand per TCP segment.
  - If none or both of the flags are enabled, the packet does not concern the DPS, and it passes through and creates no additional resource demand (e.g., the first packets of established connections from Workload Class 1).
  - If only the SYN flag is enabled, the SYN handling computes the sequence number and replies with a SYN+ACK packet. These steps create a constant load per segment with the SYN flag.
  - If only the ACK flag is enabled, the DPS validates the sequence number. This step creates a constant resource demand.
    * If the sequence number is invalid, the DPS drops the segment creating no further resource demand.
    * If it matches, the DPS notifies the server, which in turn informs the SDN controller. Thus, this step creates a constant resource demand per segment at (i) the DPS itself, (ii) the server, and (iii) the SDN controller.

IDPS:

- The IDPS step-by-step checks incoming traffic against its rules. For every level, this creates per rule a constant resource demand per frame/packet/segment.

  - If a file matches all header conditions for at least one rule, the IDPS performs DPI on the packet's body. Thus, it creates a dynamic resource demand, depending on the packet size. The correlation between packet size depends on the pattern types. For simple patterns, a linear correlation is likely, while more generic patterns can even cause an exponential correlation.

For these devices, we have found two general types of resource demand generation. The first type is a constant demand created per unit (e.g., frame, packet, segment, request). The second type creates a demand correlating to the size of the unit with different correlation types. A specialty is with the DPS that creates additional resource demand on the server and the SDN controller. Since we only model the security functions, we ignore this effect for now.

With these types, it is possible to get a relatively accurate model consisting of a single function through running measurements with a simplified ANOVA approach, always modifying only one parameter until successfully deriving a complete model of the security function.

Modeling Tertiary Factors

There are additional factors that can increase the accuracy of the model in certain situations. These factors do not correlate directly with single packets or the traffic distribution but instead with the state of the security appliance.

Queuing Behavior: Some security appliances (e.g., the IDPS) have a characteristic queuing behavior. Thereby, this may delay the resource demand — on the current and following functions — and also traffic.

False Positives and False Negatives: Not all security functions (e.g., an IDPS based on anomaly detection) by default are save from having false positives and false negatives. An accurate model should account for these. False positives result in a reduction of traffic classes the system does not protect against, while false negatives have the opposite effect. Some packets of the traffic class — that the security function defends against — remain after the traffic leaves the function.

Drop Rate: Some security functions can — in certain situations — undesirably drop packets. Such a drop rate, as well as the resulting necessary retransmissions (at least for the benign packets), impact the system, and the model should consider them.

Overload Behavior: Under very high load levels, some security functions change their behavior. This effect especially impacts the three factors listed above and requires separate modeling.

Short Term CPU Frequency Scaling: For short terms using CPU frequency scaling can counter the issues caused by an overload. The modeling of the overload behavior should account for this functionality and implement a delay for the overload effect.

5.3.2 Modeling Security Service Function Chains

RQ 5.3b Based on the previously presented model for single security functions, it is possible to model the whole security function chain. Therefore, we model the chain by putting the functions one after the other and feeding the output of the previous function to the next.

5.3.2.1 Modeling Traffic Composition Throughout the Security Function Chain

When starting with the input traffic, the traffic results from putting it through one function after the other. Figure 5.26 shows such a development of the traffic for the function described above. For illustration, we use a simplified model without tertiary factors. The traffic starts with a distribution over all traffic types. At every security function, this function removes one or more traffic classes. Thereby, the share of the other classes increases. This process repeats itself at every security function until only the benign traffic remains.

This combination allows a full model of the chain when knowing the composition at the startup. However, most of the time, this composition is unknown. Still, it is possible to compute this composition using reports from the security function about occurring attacks. We present an implementation for this reporting in Chapter 6.

At the server, the traffic wholly consists of benign packets. The switch before the server (respectively in a more complex system, the switches before the servers) can report the number of packets and the bandwidth of benign traffic using, e.g., SNMP. With the reported attack composition from the last security the appliances in the chain, we add an equal amount of packets and bandwidth with the equivalent share in the traffic composition.

Again, this step repeats at every security function. After the inverse processing through the first security function, the result is the incoming traffic.

In some experiments, we realized, this approach yields a smaller total bandwidth than actually present at the input link. This effect is due to unmodeled

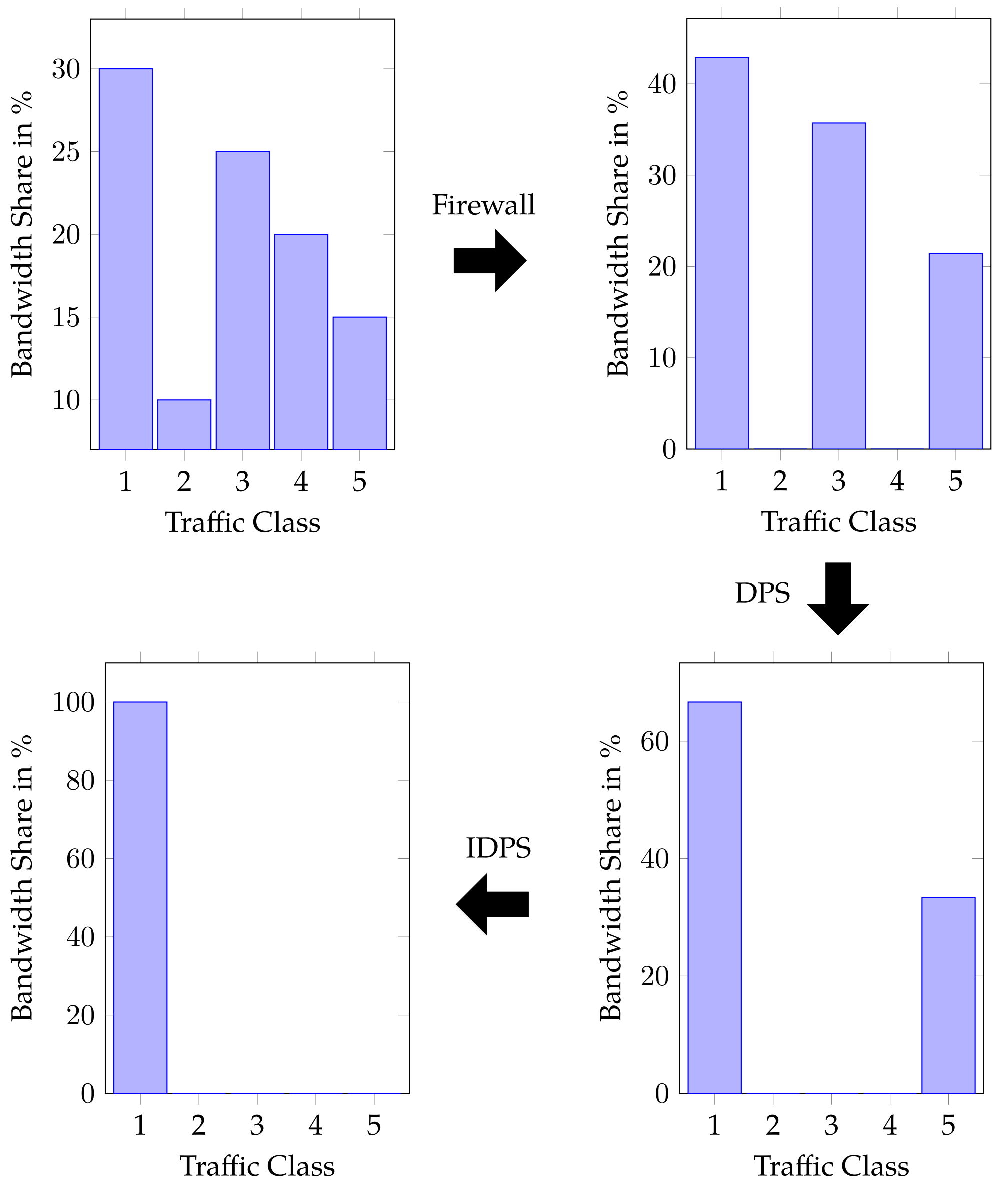


Figure 5.26: Exemyplary development of the traffic composition over the course of a security service function chain.

tertiary factors, as described before. A simple fix is to scale the traffic without changing its relative composition. We, therefore, propose to scale the total bandwidth and number of packets to the number reported by the SNMP switch, respectively, switches before the respective security function.

5.3.2.2 Calculating Total Resource Demand for the Security Service Function Chain

With the knowledge of the traffic composition at every step of the SSFC and the resource demand model for every single security function, we can compute the required resource demand.

We put the traffic composition at every stage through the resource demand model of its security function. This calculation yields the resource demand per function. The sum of the single resource demands is the total resource demand.

The unique resource demands help to decide upon necessary scaling (especially, since we have corrected for dropped packets using the data from the switches) and co-placement. Through the correction, resource demands more extensive than the provided resources can occur. This knowledge can facilitate scaling according to these demands. If the sum of demands for functions that follow in the SSFC are below the resources, it allows placing these functions on the same server to reduce the number of required virtual or physical instances and remove network latency due to additional hops (which Section 5.2.2 indicates as a significant factor).

5.3.3 Decision-making

RQ 5.3c Based on the introduced models, it is now possible to make decisions. During attacks, a dynamic SSFC reordering framework has to decide in which order to place the security functions inside the chain to achieve optimal performance. We present three approaches on how to compute the order for the security functions:

(i) complete calculation of all permutations of the chain,

(ii) front to back placement of the optimal security functions, and

(iii) security function swapping.

These approaches have different advantages and disadvantages and suit different use-cases depending on the complexity, the required accuracy, response time, and the available compute resources.

5.3.3.1 Complete Calculation of All Chain Permutations

The first approach is the most straightforward and accurate. The FCC computes every possible permutation of the security function chain. For each of these permutations, we feed the current traffic composition (gathered as shown before from the network switches and the security function reports) through a model of that permutation. For all these SSFC orders, the models yield the resource demand. The SSFC order with the lowest resource demand is the optimal SSFC order. Based on the resource demand, it is also possible to calculate how many instances of a security function are necessary.

Since we calculate all permutations, this approach is guaranteed to yield the SSFC order with the optimal (modeled) performance. However, this approach can be very compute-intensive. The number of permutations P of n different security functions results from the following Equation (5.2):

$$P(n) = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1 = n! \tag{5.2}$$

While for our examples of three security functions, only six permutations exist, four security functions already increase this to 24 permutations. The number of permutations grows faster than an exponential function. With this knowledge, it is possible to approximate the amount of computing steps C for n security function as follows in Equation (5.3):

$$C(n) = n\,(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1) = n! \cdot n \in \mathcal{O}(n!) \tag{5.3}$$

For a sophisticated security system with a multitude of security functions, the exploration space explodes. This growth leads to a massive need for computing resources and increases the possible reaction time to changes in the attack composition.

5.3.3.2 Front to Back Placement of the Optimal Functions

In contrast to the first approach, the second does not fully compute all permutations. Instead, the FCC builds the SSFC order from front to back. In the first step, the FCC applies the stand-alone model of every security function on the incoming traffic (as derived by the model). The security function with the smallest resource demand in this model then takes its place as the first security function in the chain. Next, we use the traffic after passing through this function and model it for every security function except for the already used function. The approach continues until having placed all functions in the chain.

This solution reduces the complexity of the calculation since it no longer calculates all permutations — in fact, it only computes the final order entirely. It is possible to approximate the number of computing steps C for n security function as follows in Equation (5.4):

$$C\left(n\right) = n + \left(n-1\right) + \left(n-2\right) + \ldots + 1 \in \mathcal{O}\left(n^2\right) \tag{5.4}$$

Thereby, the runtime shrinks from a more than exponential growth to quadratic growth. This shrink is a massive reduction and allows us to compute more complex SSFCs. However, still in some cases, a faster reaction is necessary, or computing resources for the FCC are rare. Also, the approach yields a good order but can not guarantee that a different order might be more efficient.

5.3.3.3 Security Function Swapping

Unlike the first two, the latter approach does not recompute the entire chain. Instead, it calculates the effect of swapping two functions. Therefore, it randomly selects two functions to swap and models the system with and without swapping these functions. If swapping reduces the resource demand, the FCC orders to swap them.

This solution further reduces the complexity of its invocation. It calculates only two full chains. The number of computing steps C for n security function per iterations results from the following Equation (5.5):

$$C\left(n\right) = 2n \in \mathcal{O}\left(n\right). \tag{5.5}$$

This reduction allows a constant and fast reconfiguration. However, the reconfigurations are only step-wise. A full reordering of the chain can take multiple steps.

Also, the approach behaves like a simple hill-climbing algorithm. Thus, it can reach a local performance maximum and stop. In contrast, still, a better global maximum might be available (e.g., when doing a single reordering, that decreases performance, this can open another reordering, with a more significant benefit). Further, the selection of the functions to be swapped is non-trivial. A random approach is inefficient, and the algorithm could benefit from taking factors like the position in the chain and the current resource demand into account. Here, a heuristic study could improve this approach.

5.3.3.4 Overview

We introduced three approaches to decision-making and reordering. They have different characteristics, as shown in Table 5.13. As expected, reducing the run

Approach	Optimal Order Guaranteed	Full Reordering Performed	Runtime
Complete Calculation	✓	✓	$\mathcal{O}(n!)$
Front-to-Back	✗	✓	$\mathcal{O}(n^2)$
Swapping	✗	✗	$\mathcal{O}(n)$

Table 5.13: Comparison of the suggested decision-making approaches.

time of the calculation also removes quality features like a guaranteed optimal order or a full reordering. Depending on the use-case, a system architect will have to weigh these factors and decide which best suits the desired use-case. Furthermore, the approaches could combine, e.g., the swapping approach could run continuously while performing the full calculation in longer intervals.

5.4 Summary and Evaluation of Research Questions

In this chapter, we introduced the concept of attack-aware dynamic Security Service Function Chain reordering. This concept incorporates changing the order of SSFCs to optimize them to most efficiently counter attacks.

RQ 5.1: What components and capabilities does a Security Service Function Chaining framework require?

At first, we described the general idea. The main component is the Function Chaining Controller (FCC). It gathers information to model the security systems state, uses this information to compute the desired configuration, and enforces the order.

RQ 5.2a: How do single security functions perform under attack load?

Next, we developed an evaluation environment for individual security functions and SSFCs. When benchmarking single security functions, we found different types of behavior. While for some attacks, the use of a security function eliminates all performance impacts from the attack, for other attacks, it merely reduces the effect. For the third type of attack, using the security function even reduces the system's performance compared to not using the function.

RQ 5.2b: What is the impact of the ordering when combining different security service functions?

For every tested combination, the order has a significant impact on the system's performance. In general, putting the function — that defends against the attack — first, yields better performance. The difference can make up two or more orders of magnitude. For different attacks (e.g., HTTP flood vs. intrusion flood), we found orders that contradict each other. Thus, there is no order that is optimal for every attack.

RQ 5.3a: How to model single security functions for the reordering decision?

We model the traffic categorizing it into traffic classes, where benign traffic and every attack type each forms a class. Every security function affects the traffic as a function depending on the traffic composition. Also, depending on the traffic composition is the resource demand of the security functions. We map the traffic to constant resource demands per traffic unit (frame, packet, or segment) as well as dynamic demands depending on the size of the traffic unit.

RQ 5.3b: How to model security function chains for the reordering decision?

The model for an SSFC consists of multiple security function models. Traffic that exits one function continues to the next. Thereby, it is possible to compute the total resource demand. We use this model backward to derive the incoming traffic composition from the traffic that enters the server and the attack reports from the security functions. Last, we introduced a correction to modeling errors using the SNMP data from the used switches.

RQ 5.3c: What strategies are suitable for determining a better order?

Last, we introduced three approaches to deciding for new security function orders: (i) a complete calculation of all chain permutations, (ii) front to back placement of the optimal functions, and (iii) security function swapping. We discussed the advantages and disadvantages of every solution and associated them with corresponding use-cases.

Chapter 6

A Framework for Attack-aware Security Service Function Chain Reordering

In the previous chapter, we first sketched out the general idea of attack-aware dynamic SSFC reordering. The next section confirmed our claim that the ordering of security functions is relevant to the performance. We also found contradicting orders for different attacks leading to the need for dynamic reordering. Afterward, we introduced modeling formalisms for single security function and SSFCs and how to use them for decision making. In this chapter, we present an architecture for an attack-aware dynamic SSFC reordering framework and provide a proof of concept implementation. We then evaluate this implementation and its capabilities and discuss the results and further challenges that arise from them.

Research Questions

In this chapter, we tackle several research questions. All of the following research questions are part of the meta-research question **MRQ 6:** *How to design a framework for dynamic SSFC reordering?*. The numbering of these research questions maps to the sections of this chapter. If a section deals with more than one research question, those questions have their number appended by ascending Latin letters.

RQ 6.1 How to structure a framework for dynamic function chain reordering?

RQ 6.3a What results does a prototype implementation provide?

RQ 6.3b Do new attack vectors, and other issues arise from dynamic function chain reordering – and if yes, how can these issues be addressed?

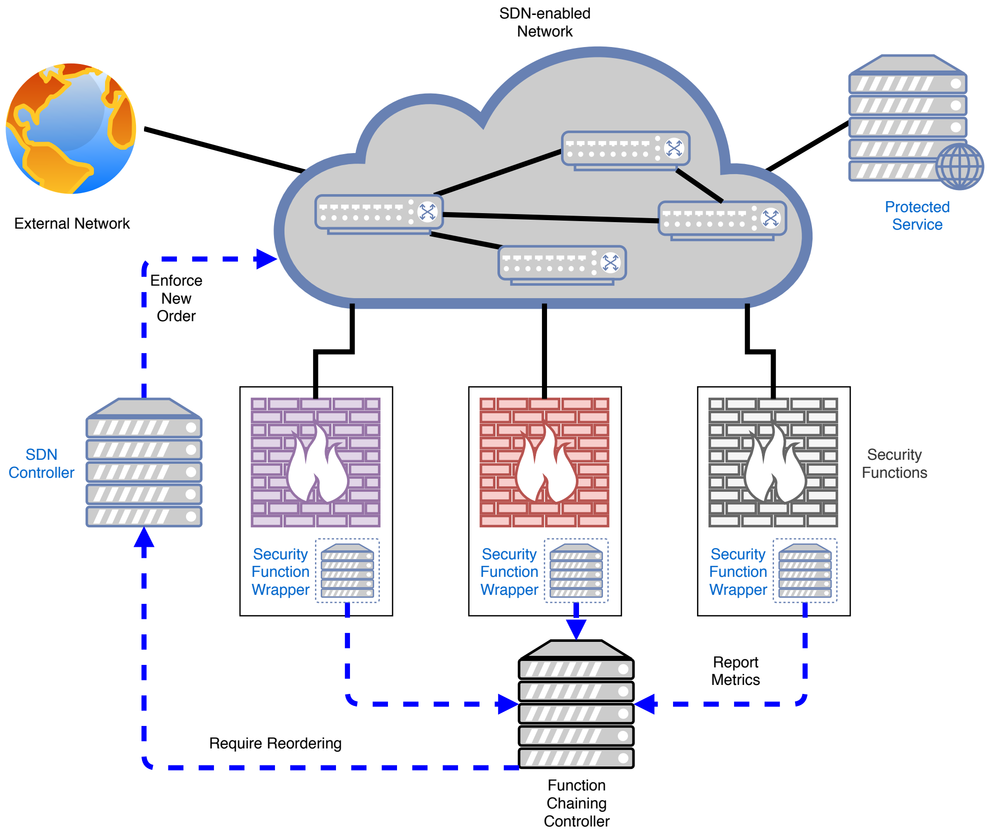


Figure 6.1: Components of the attack-aware SSFC reordering framework.

6.1 Architecture

RQ 6.1 The attack-aware SSFC reordering framework consists of multiple components. Figure 6.1 gives an overview of these components. As a backbone, a generic SDN-enabled network connects the external network and the service protected by our security system. Thus, all relevant security functions connect to that network as well. We deploy so-called security function wrappers alongside the security functions to gather metrics about them and their attack and report them to the FCC. The FCC collects data from the wrappers and optionally other sources. It forms a decision whether another ordering is better, based on the gathered security function data. If it deems that a reordering would improve performance, it sends the new ordering the SDN controller. The SDN controller then enforces the new order inside the SDN-enabled network. On the following pages, we give further detail on the security function wrapper, the FCC, and the requirements for their communication API.

6.1.1 Security Function Wrapper

The security function wrapper is a program running on the security function hosts and communicating with the FCC. It is responsible for registering the security function at the FCC, delete it on graceful shutdowns, keep a connection to the FCC to allow the management of security functions through the FCC, and finally offer an interface for the security function to report detected attacks to the FCC over the wrapper.

Figure 6.2 illustrates the communication between the security function wrapper and the FCC during start-up and shut down. At first, the security function wrapper validates and loads its configuration. If everything is loaded correctly, it sends a registration to the FCC with the group (the type of security function, e.g., firewall, IDPS, or DPS) and Media Access Control (MAC) address of the security function. The FCC then checks whether the security function is already registered. If this is not the case, the FCC adds the new function to its internal data structure and generates a unique token and ID and sends them back to the security function wrapper. If the data structure already contains the security function – e.g., when performing a hard reset on the security function server, the FCC retrieves the corresponding token and ID and sends it to the security function wrapper with a notification that it already registered before. If the token already expired, the FCC generates a new token instead of using the expired one. Next, the security function wrapper starts its keep-alive thread and enters its main loop (see below for both loops). If the wrapper catches a `SIGTERM` signal from the operating system, it shuts down the keep-alive thread.

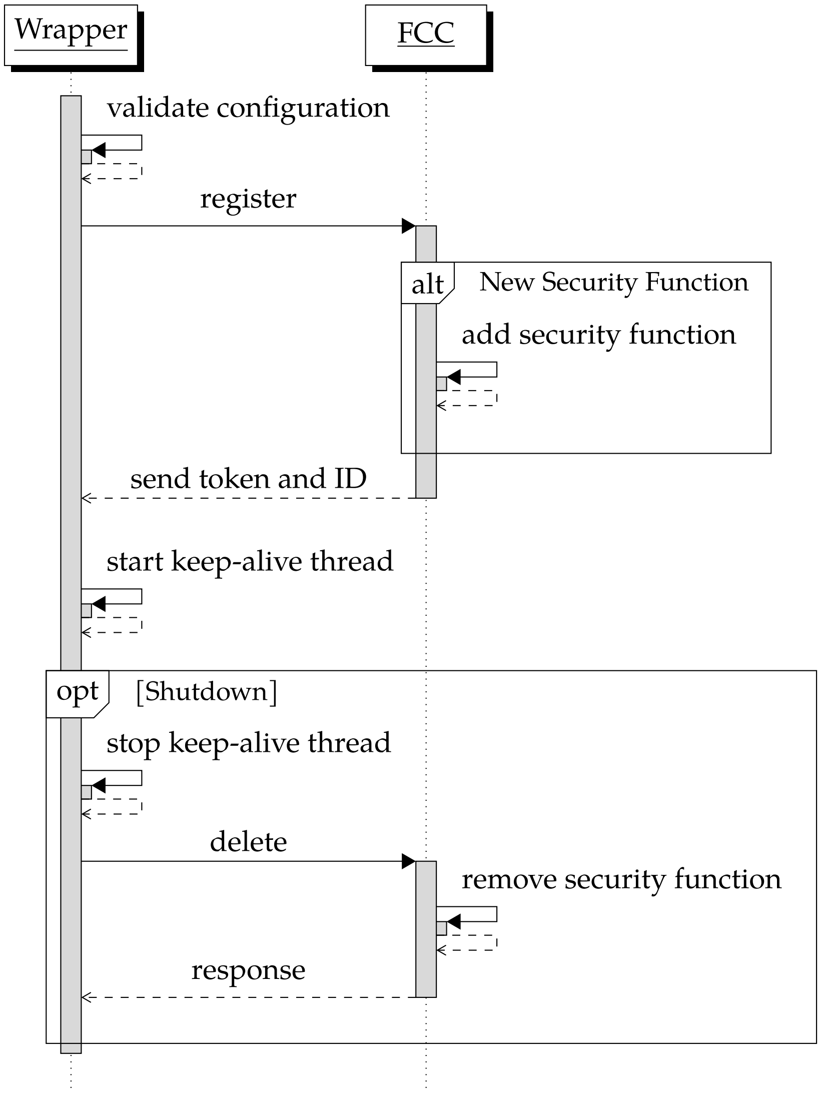


Figure 6.2: Start and Shutdown behavior of the security function wrapper and the communication with the FCC.

Then, it sends a delete message to the FCC, which removes the security function from its internal data structure and then confirms the deletion to the security function wrapper. After receiving this confirmation, the security function wrapper terminates.

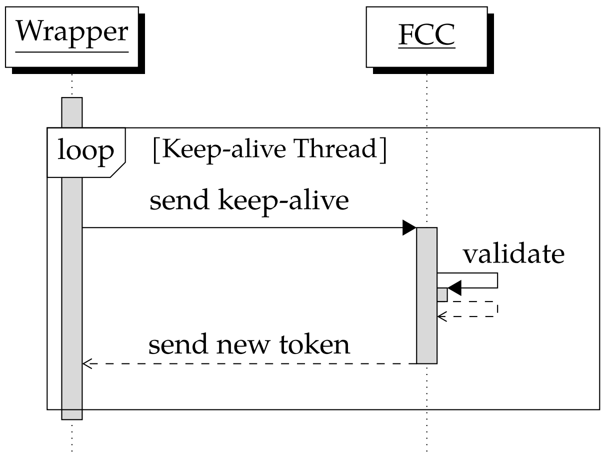


Figure 6.3: Keep-alive communication between the security function wrapper and the FCC.

In the keep-alive loop, as shown in Figure 6.3, the security function wrapper periodically sends a keep-alive message to the FCC. The FCC validates this message and sends an updated token back to the security appliance. The security function wrapper then uses the updated token for further reports.

In the main loop, as seen in Figure 6.4, the security function wrapper waits for attack reports from the security function. It then validates these reports (e.g., a machine on a 1GBit/s interface could not report a valid attack with a strength of 5 GBit/s). When this validation succeeds, the security function wrapper forwards it to the FCC. Depending on the reporting mode of the security appliance, it is also possible to collect attack reports, accumulate them, and then send the condensed data in a fixed interval. The FCC validates the report from the security function wrapper and updates its internal list of occurring attacks. It then acknowledges the alert to the security function wrapper.

6.1.2 Function Chaining Controller

The FCC runs in a central location reachable from all security functions. A webpage showing attack statistics, the current and standard configuration is part of the FCC. Additionally, the webpage contains a form to change the

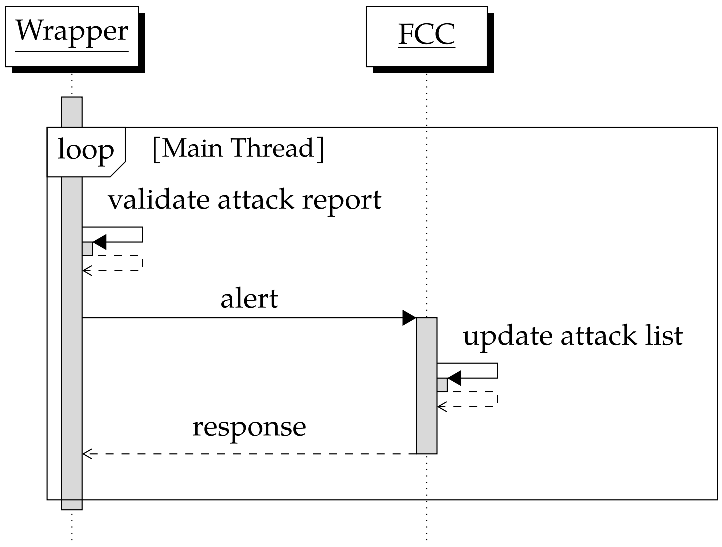


Figure 6.4: Communication between the security function wrapper and the FCC for attack reporting.

routing configuration manually based on the available groups of security functions. The controller needs to handle the requests from the wrapper instance, namely registration, delete requests, attack alerts, and keep-alive requests, as shown before. The FCC must keep a list of the security function groups and their respective attack rate to calculate the new optimal routing configuration reactively. After calculating the new routing configuration, the FCC sends it to the SDN controller, which then applies it to the switches and, therefore, the network.

Figure 6.5 visualizes the interaction between the FCC and the SDN controller as well as between the SDN controller and the switches comprising the SDN-enabled network. During the start of the controller, it validates its configuration file. This validation is important because the configuration lists all available groups of security functions and the standard routing configuration. After the validation of the configuration file, the routing function starts in a new thread. This thread periodically checks whether a new routing configuration can perform better based on a decision algorithm (for approaches to decision-making check Section 5.3). In this section, we use a simplified approach putting the security function group with the most attacks at the front. If a new routing configuration is necessary, the FCC sends it to the SDN controller. Next, the SDN controller creates the flow from the transmitted routing configuration.

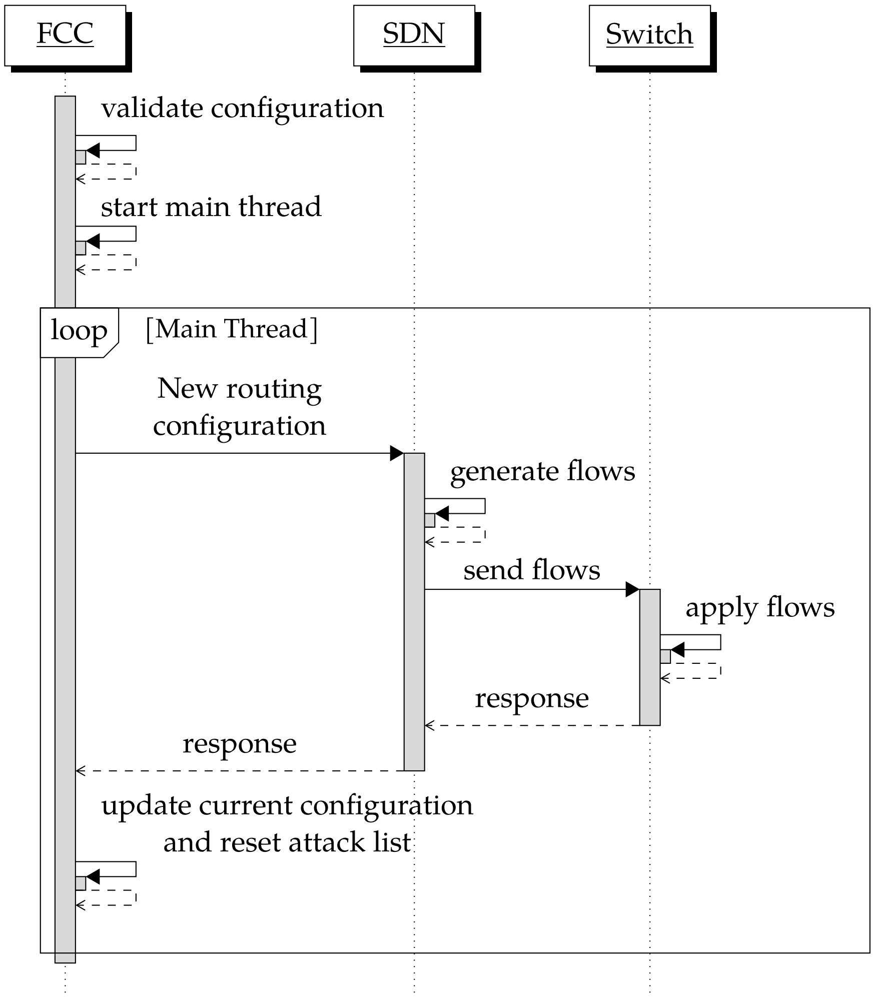


Figure 6.5: Interaction between the FCC, the SDN-controller and the switches for reordering.

The SDN controller sends the generated rules to the respective switches which apply them. If the switches applied the new routing configuration successfully, they send a response to the SDN controller, who, in turn, forwards it to the FCC. After successfully changing the routing configuration, the FCC changes the stored current configuration to the new routing configuration and resets the reported attacks.

6.1.3 API Requirements

As shown before, the security function wrapper and the FCC communicate and exchange data. This communication requires a predefined API. As is the standard for modern APIs, we encode all information using the JSON format. Each message uses a unique *type* of key for determining the message type. Our framework uses four types of messages for communication between the security function wrapper and the FCC, as described in the following.

REGISTER

Registration requests contain the following three additional keys:

group: the group of the security function (e.g., firewall, IDPS, DPS)

hw_addr: the MAC address of the interface connecting the security function to the switch

misc: placeholder for other usages (e.g., debug information) or future features

The response message for registration requests contains the token to validate future requests and an ID for the Security Appliance generated by the Function Chaining Controller.

KEEP-ALIVE

Keep-alive messages periodically trigger the FCC to renew the security function wrapper's token for the messages it sends to the FCC. They contain four additional keys:

name: the ID of the security function

group: the group of the security function

hw_addr: the MAC address of the interface connecting the security function to the switch

misc: placeholder for other usages (e.g., debug information) or future features

The ID is essential for the FCC to determine if the security function wrapper already registered with FCC by checking the list of designated security functions for the ID and verify that the group and MAC address match. The response message contains a renewed token.

ATTACK

The wrapper sends attack requests to the FCC containing an attack rate. The FCC determines from this rate possible new routing configurations. The attack requests include five additional keys:

name: the ID of the security function

group: the group of the security function

hw_addr: the MAC address of the interface connecting the security function to the switch

rate: the attack rate – depending on the attack in attacks per second or MBit/s

misc: placeholder for other usages (e.g., debug information) or future features

The security function determines the attack rate. All additional information is required to verify that the security function is authorized to send attack reports.

DELETE

When the wrapper terminates gracefully, it sends a delete message to the FCC. In addition to the message type, these delete messages contain two more keys:

name: the ID of the security function

misc: placeholder for other usages (e.g., debug information) or future features

With the previously described fields, the FCC receives all required information. We give further information about the API integration on the following pages.

6.2 Proof-of-concept Implementation

The complete framework uses Python 3. We chose Python because it is simple to learn, offers useful libraries for REST interfaces, and HTTP requests and does not require compilation allowing for unhindered modification. In this section, we first take a look at the used libraries and move on to the security function wrapper and FCC implementations. Then, we also introduce a small SDN controller, which we use for the evaluation.

We provide the source code for the security function wrapper at `https://github.com/bladewing/SecurityFunctionWrapper`, for the FCC at `https://github.com/bladewing/FunctionChainingController` and for the SDN controller at `https://github.com/bladewing/SDN-Controller`.

6.2.1 Libraries

We use four non-standard Python libraries: Flask [Ron20], PyJWT [Pad20], requests [Rei20], and netifaces [Hou20].

Flask is a lightweight Web Server Gateway Interface (WSGI) web application framework designed with simplicity as well as scalability for sophisticated applications in mind. It started as a simple wrapper and is now one of the most used Python web application frameworks. It uses the Jinja templating engine and the Werkzeug WSGI library.

Request gives itself the slogan "HTTP for humans" and describes itself as "A simple, yet elegant HTTP library." It allows sending HTTP/1.1 requests through it. Thus, it removes the need to configure the query strings to Uniform Resource Locators (URLs) manually.

PyJWT implements JSON Web Token (JWT). "JSON Web Token (JWT) is a compact, URL-safe means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC*) and/or encrypted." [JBS15]

The netifaces packet is a networking abstraction. It takes care of the underlying networking management tools and allows an independent standardized interface to access network information. We use this packet to get the IP and MAC addresses of the underlying machines.

6.2.2 Security Function Wrapper

The wrapper is a *Flask* application. The main tasks of the security wrapper are to forward security function attack reports to the FCC. After validating the wrapper configuration file, the security function wrapper sends a query for registration to the FCC, which answers it with a token. The FCC later uses this token to validate requests from the security function wrapper.

The wrapper implements the following URI:

`/attack`: The security function sends its attack reports to this URI, as described in the API requirements. The wrapper then forwards the incoming attack report to the FCC, as described in the API.

In the course of this thesis, we developed implementations for the IPTables firewall [Hem20] and the Snort IDPS [Fel20]. The THREADS DPS supports attack reporting by default.

Following the registration, the `keepalive` method starts in a separate thread. It contains a loop where every `TIMEOUT` minutes, where **TIMEOUT** is configurable in the configuration file, it sends a keep-alive message to the FCC. The wrapper will not start if the FCC is not available on the configured URL. Note that the `TIMEOUT` in the configuration file of the wrapper and the FCC should be the same. Otherwise, synchronization problems can appear. `PyJWT` supports the use of `leeway=X`, meaning that there is a margin of `X` seconds where tokens still count as valid if the expiration time is within the margin. This margin is essential as there may be connectivity issues between the wrapper and the FCC. When the wrapper catches a *SIGTERM* signal from the operating system, it shuts down the keep-alive thread and sends a delete message to the FCC.

6.2.3 Function Chaining Controller

The FCC is in its core a *Flask* application, and FCC has a log file where it logs everything to make debugging easier. The FCC provides the following URIs:

`/`: This URI is the controller's homepage. It is not responsible for handling API requests. It gives an overview over (i) the *current attack count*, (ii) the list of available security function groups, and (iii) their respective attack counter. Additionally, in the *Configurations* section, the homepage shows the configured standard ordering and the currently active configuration below it. This side-by-side presentation allows the user to see changes in the configuration, e.g., when the system has adapted to attacks. Finally, the *Change current configuration* section allows changing the current configuration manually. Therefore it is possible to select distinct groups and apply them using the *Change Configuration* button.

`/handle_data`: The event triggered when pressing the *Change Configuration* button in the homepage, sends the selected configuration to this URI. Here, the FCC validates it and sends it to the SDN Controller. If the request is successful, a page appears with the new configuration and the response code of the SDN Controller. On failure, the FCC loads a page showing the reason why applying the configuration was not possible, the configuration that was invalid, and the current configuration to help spot mistakes.

`/register`: This URI processes incoming registration requests of security function wrapper instances. It validates these requests via a JWT and, if valid, adds the security function to the security function manager. The response message to the security function wrapper contains a token for future communications and the ID of the newly added security function. If a security function is already present, e.g., after a restart of the security function's host machine, the FCC replies with the existing ID and token upon receiving a new register request for such a service. A shared secret guarantees that incoming register requests can only originate from the authentic wrappers and ensures securing access against attacks.

`/keep-alive`: The `/keep-alive` URI validates keep-alive messages from security function wrapper instances using JWT. After a successful request validation, the FCC checks if the security function already has registered. If meeting both criteria, the security function wrapper receives a new token as a response. Else, it gets an error message.

`/alert`: This URI is relevant for the FCC to find the optimum routing configuration. If a security function detects an attack, it sends the attack rate to the FCC via its security function wrapper. This URI then updates the attack counts of the respective security function.

`/delete`: When gracefully shutting down a security function wrapper instance, that instance sends a delete message to the FCC. If the request is valid, the FCC removes the security function from the security function manager.

`/secapps`: This URI prints the IDs of all registered security functions.

`/routing`: The FCC manages a list with all security function groups and their attack count, called `ATTACK_LIST`. The management function runs in a thread with a loop, checking for new configurations every `TIMEOUT` minutes. `TIMEOUT` is configurable in the configuration file of the FCC. The FCC sorts `ATTACK_LIST` by the attack count in descending order. Within the proof-of-concept implementation, the FCC sends this ordered list to the SDN controller, as described below.

Other methods with self-explanatory names handle some tasks of the URIs.

6.2.4 Software-defined Networking Controller

In general, this framework supports every SDN controller offering a REST API for flow modification. To ensure the absence of side effects from the SDN controller, we implemented a minimalistic SDN controller ourselves. To this end, for this proof-of-concept evaluation and the following evaluation, we limit our framework to the use with Open vSwitch. This SDN Controller consists of two *Flask* applications: the actual controller and a switch wrapper running on the **Open vSwitch** machines. The SDN Controller gets a list of security function groups, generates flows for the passed configuration, and forwards those flows to the switch wrappers, which execute the necessary `ovs-ofctl` commands, and, therefore, apply the new settings. Note that the process of generating flows is dependant on the implementation in the different SDN Controllers according to the documentation of their REST API. In the following, we describe how the SDN controller generates its flows.

Generating Flows

The simple SDN controller uses a template for flow creation with the following match fields and actions:

Match fields :

`hard_timeout=TIMEOUT`: The flow's hard timeout is `TIMEOUT` seconds. After this time, the flow expires, and the switch removes it from its flow table.

`priority=X`: This field sets the priority of the created flow. The priority of the flows used for reordering should exceed the priority of the standard configuration (simple MAC-based switching) as the matching rule with the highest priority applies to a packet.

***dl_type=ETH_TYPE*:** This field contains the *EtherType* of the packets that the flows should match. E.g., $0x0800$ for IP packets or $0x0806$ for Address Resolution Protocol (ARP) packets.

`in_port=X`: A flow only applies to packets coming from port `X`.

`dl_src=00:00:00:00:00:0X`: The flow only applies if the MAC address matches the one listed in this match field.

texttttnw_src=10.0.0.1: The flow only applies to packets that have the same IPv4 source address as listed in this match field.

`nw_dst=10.0.0.2`: The flow only applies to packets that have the same IPv4 destination address as listed in this match field.

Actions fields :

`mod_dl_src:00:00:00:00:0Y`: Modifies the MAC address of the packets before forwarding them.

`output:Y`: Forwards the packet to port Y.

With this template, the only fields changing are `in_port`, `dl_src`, `mod_dl_-src`, and `output`. For example, it allows routing the traffic from `EXT` to `INT` via a VNF on port 2 of the Open vSwitch in Listing 6.1.

Matching packets on the IPv4 source address and MAC address is sufficient to have distinct flows.

6.3 Evaluation

6.3.1 Testbed Environment

Figure 6.6 gives an overview of the used testbed. For the evaluation, we used four servers named *C39, C45, C48*, and *C49*. Each server has an Intel Xeon CPU E5-2640 v3 clocking at 2.60GHz with eight dedicated cores and 32GB RAM. Table 6.1 gives more detail on the server specifications and also shows that all servers and the spawned virtual machines use minimal Ubuntu 16.04 as an operating system. Minimal Ubuntu is a regular Ubuntu stripped of all command-line convenience features to reduce the image size, installation duration, and the number of required security updates (only installed software requires patching). However, if needed, all those features can be installed using Ubuntu's package management. Kernel Virtual Machine (KVM) is the hypervisor that spawns all needed VMs. **RQ 6.3a**

C39 and C48 are hosts for additional virtual machines. Every server and switch is on the `192.168.66.0/24` subnet (see Table 6.2). The security function wrapper runs on each virtual security function, and the FCC runs on `C45`

```
# Incoming from Sender, forward to VNF on port 2

hard_timeout=300,
priority=100,
dl_type=0x0800,
in_port=EXT,
dl_src=SENDER_MAC,
nw_src=SENDER,
nw_dst=RECEIVER,
actions=mod_dl_src:00:00:00:00:00:02, output:2

# Incoming from second port of VNF, forward to INT

hard_timeout=300,
priority=100,
dl_type=0x0800,
in_port=3,
dl_src=00:00:00:00:00:02,
nw_src=SENDER,
nw_dst=RECEIVER,
actions=mod_dl_src:RECEIVER_MAC, output:INT
```

Listing 6.1: Example flow configuration for the SDN controller.

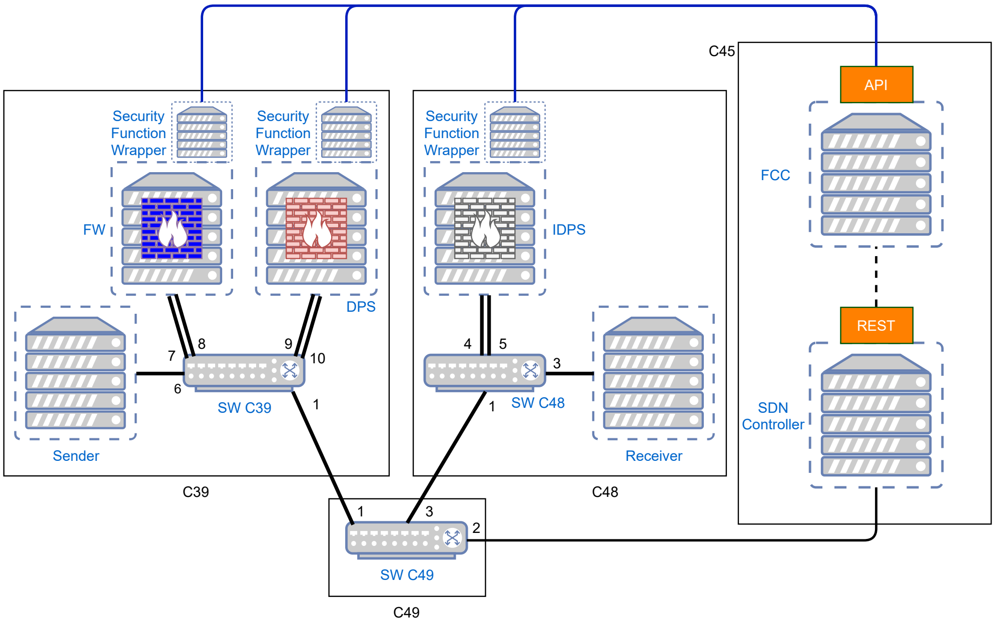


Figure 6.6: Testbed for the dynamic function reordering framework.

Unit	**Value**
Product	HP ProLiant DL360 Gen9
CPU	Intel Xeon E5-2640 v3
Default CPU frequency	2.60 GHz
Max CPU frequency	3.40 GHz
Min CPU frequency	1.20 GHz
Cores (Threads)	8 (16)
Cache (L1/L2/L3)	512 KB/2048 KB/20480 KB
Memory size	32GB (2 x 16 GB) DDR4 Dual Channel
Memory frequency	1.866 GHz
Memory Connection	Dual Channel
Storage Model	HP VK0800GEFJK 800 GB SSD
Storage Connection	SATA III (6GBit/s)
Operating System	minimal Ubuntu 16.04.2 LTS (x86-64)
Kernel	4.4.0-68

Table 6.1: Specification for all physical servers (C39, C45, C48, C49) in the testbed, as seen in Figure 6.6.

together with the SDN Controller. The VMs *Sender*, *FW*, and *DDoS* all run on `C39`, in combination with an Open vSwitch instance as a bridge between host and guests. Next, the VMs *IPS* and *Receiver* run on `C48` alongside an Open vSwitch instance as a bridge as well. The Open vSwitch on `C49` connects the Open vSwitches from `C39` and `C48`. It forwards every packet coming from `C39` to `C48` and back. Under this setup, incoming traffic from the sender to the receiver passes through all three switches. All three VMs *FW, DDoS*, and *IPS* have two NICs each, where one NIC handles incoming traffic which, after processing the packets, it leaves the security function via the other NIC.

Name	Host	IPv4	Ports	MAC
Sender	C39	`192.168.66.200`	6	`52:54:00:91:60:4d`
Receiver	C48	`192.168.66.201`	3	`52:54:00:93:cd:2d`
Firewall	C39	`192.168.66.100`	7, 8	`52:54:00:09:38:52`
DPS	C39	`192.168.66.101`	9, 10	`52:54:00:d3:db:f1`
IDPS	C48	`192.168.66.102`	4, 5	`52:54:00:e3:6f:ac`

Table 6.2: Network information of virtual machines in the testbed from Figure 6.6

6.3.2 Manual Reordering

Experiment Description

We test all six routing orders possible for the three security functions. Therefore, we created an automated to test these routing configurations in the network based on the standard configuration. The order of the permutations is not relevant. After starting the system and registering the Security Appliances, the *Sender* sends an ICMP echo request to the *Receiver* with the IP addresses `192.168.66.200` and `192.168.66.201`, respectively. Next, we started *tcpdump* with the following arguments on each security function:

```
timeout 65 tcpdump -i br0 -w DPS-exp1 icmp
```

Tcpdump must log the traffic for 60 seconds – the duration of the script. Thus, we added *timeout 65* to have a margin of 5 seconds. Next follows the *-i br0* argument. *Tcpdump* only listens on packets on the *br0* interface. The *-w DDOS-exp1* argument describes the output file for the dump. Here, the target file is

DPS-exp1. On the other security functions, we replace *DPS* with *IDPS* or *FW*. The last argument, *icmp*, allows us to filter the packets on the interface for ICMP messages and eliminate, e.g., controller traffic.

In summary, the command above logs ICMP traffic coming to the specified interface for a specified amount of time and writes it into a file. Additionally, the simple SDN controller saves the generated flows for each switch before sending them to the switches. The script iterates through all valid permutations of the standard configuration to show that the generated flows are correct. This automatic permutation allows reusing the script for SSFCs with more than three security functions.

Example of Flow Creation

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=6,
dl_src=52:54:00:91:60:4d,nw_src=192.168.66.200,
nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:09:38:52,
output:7
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=8,
dl_src=52:54:00:09:38:52,nw_src=192.168.66.200,
nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:d3:db:f1,
output:9
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=10,
dl_src=52:54:00:d3:db:f1,nw_src=192.168.66.200,
nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
output:1
```

Listing 6.2: Flows for the Open vSwitch running on `C39` for the first routing change.

Listing 6.2 shows the flows the SDN controller generated to change the routing configuration. As illustrated in Figure 6.6, the first rule matches the *in_port*, MAC address, and IPv4 source address of the *Sender*. The destination IPv4 address is the address of the *Receiver*. Every IP packet, coming from the *Sender* and having the *Receiver* as its goal, triggers a match from this flow. The *actions* ensure that the switch modifies the source MAC address of the packet to the firewall's MAC address and forwards all packets to port 7 (see Table 6.2). Looking at the second flow, every IP packet coming from port 8 with the source MAC address of the firewall triggers this flow forwarding it port 9. Both NICs of the firewall connect to the switch on ports 7 and 8, respectively. The first two flows forward incoming traffic from the *Sender* to the firewall. Before forwarding the packets to port 9, the switch again modifies the source MAC address of the packets to the MAC address of the *DPS* virtual machine. Looking at the last flow shows that it forwards every packet coming from port 10, and with the source MAC address of the *DPS* virtual machine proceeds to port 1 while also changing the source MAC address to the MAC address of the *IDPS* virtual machine. In summary, the switch on *C39* routes all packets first to the firewall VM, then to the DPS VM, and finally out on port 1 with the source MAC address of the IDPS VM. Next, they pass via the Open VSwitch on `C49` to `C48`.

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
output:4
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=5,
dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
output:3
```

Listing 6.3: Flows for the Open vSwitch running on `C48` for the first routing change.

Listing 6.3 illustrates the continuation of flows from before coming from `C39` on `C48`. The first flow matches every IP packet coming from *in_port* 1 with the source MAC address of the *IDPS* VM and forwards it to the respective port. The first flow has a little overhead due to once more rewriting the source MAC address to the MAC address of the *IDPS* VM, which could be optimized. Lastly, the second flow matches every packet coming from the *IDPS* VM port and MAC address and forwards them to their destination: the *Receiver*. It is important to note that in its current state, we did not yet optimize the creation of the flows to remove redundant flows, offering room for improvement in the future.

Results

Figure 6.7 visualizes all changes in the ordering configuration during the automated experiment. The graph visualizes the recorded packets from the

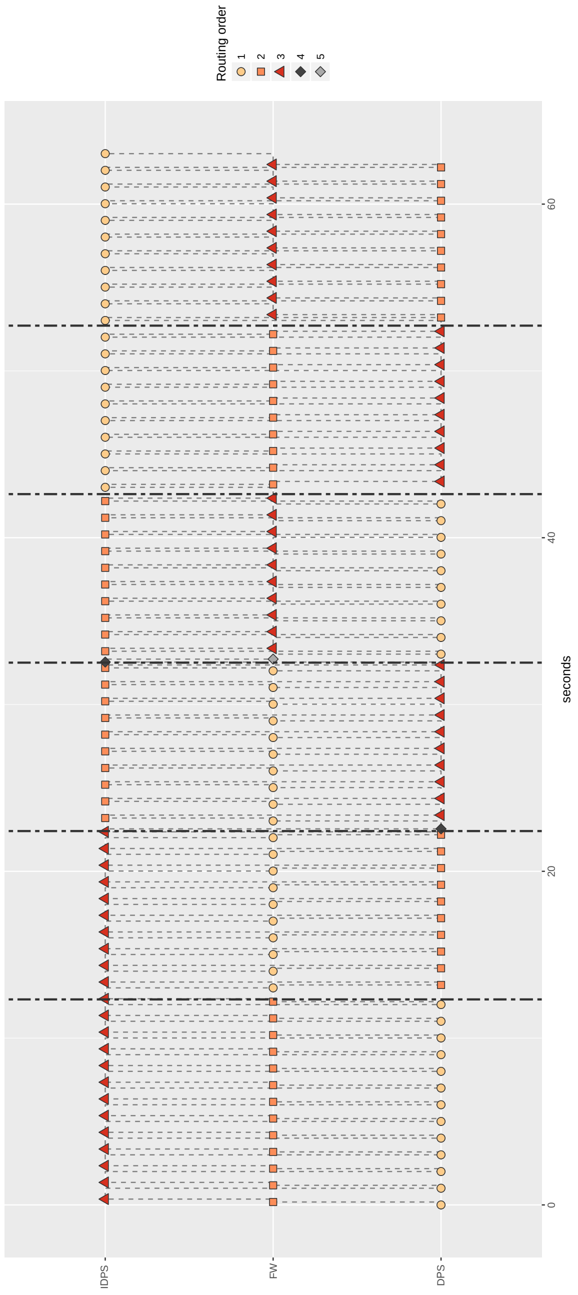


Figure 6.7: Landscape graph of a *TCPDUMP* of ICMP requests sent from the *Sender* to the *Receiver*

tcpdumps showing the order of security function each packet. On the x-axis (beware of the graph's 90-degree rotation, you can find the time of the experiment, and on thy y-axis, we depict the security functions. Circles (first security function, the packet passes through)squares (second security function), and triangles (third security function) mark the routing order. For the case that a packet traverses more than three functions, the charts indicate this with colored diamonds. Black vertical lines show the time when the script applied a new permutation of the default routing configuration. The first section (before the first vertical line) shows that the packets traverse through the *DPS* VM, next through the *firewall* VM and lastly through the *IDPS* VM. This order is the default configuration of the network. After applying the first new routing configuration, the packets now traverse through the *firewall* VM first, then through the *DPS* VM, and lastly through *IDPS* VM. The first new routing configuration matches the configuration from Listing 6.2 and Listing 6.3, showing that the packets traverse through the security function in the desired order.

Moreover, Figure 6.7 shows that our framework applies every permutation of the default configuration correctly. As seen, it is possible to traverse through more security functions than there are in the network. This issue is a result of changing the routing configuration while traffic passes through the system. The following packets go through the desired function chain. The gray and black diamonds in the graph show the packets that traversed through more security functions than configured because a new routing configuration went into effect during the ICMP requests. Although packet loss is theoretically possible, it does not occur, and even then, new routing configurations are applied instantly. Also, while theoretically possible, no attack completely skipped a security function.

6.3.3 Reaction to Simulated Attacks

Experiment Description

This experiment analyzes the FCC's attack-awareness. Thus, it validates whether the FCC correctly changes the routing configuration based on the attacks reported by the security functions. The previous section showed that the framework created all generated flows correctly and applied them successfully to change the routing configuration of the network. As described in Section 5.1, the main idea that led to the development of the framework is to change the routing configuration *dynamically*. As described in Section 6.1 and 6.2, the security functions report detected attacks to the co-located security function wrapper instance, which then reports it to the FCC. We simulate attacks on each virtual machine, to show that the attack reporting works and the routing configuration changes depending on the attack reports. The simulated security functions send attack reports with a changing probability of $2 \cdot 10^{-6}$, $4 \cdot 10^{-6}$, and $6 \cdot 10^{-6}$ every millisecond. We configured a threshold of 100 in the FCC. Only if the attack count exceeds this threshold, the FCC calculates and – if necessary – applies a new routing configuration. Additionally, we define an *imminent threshold*, which is three times as large as the regular threshold. The FCC checks for new routing configurations every five minutes, but every ten seconds, the FCC checks if the reported attacks of a security function group exceed the *imminent threshold*. In that case, it immediately recomputes the routing configuration and applies it if it has changed. We selected these values to prove our proof-of-concept implementation.

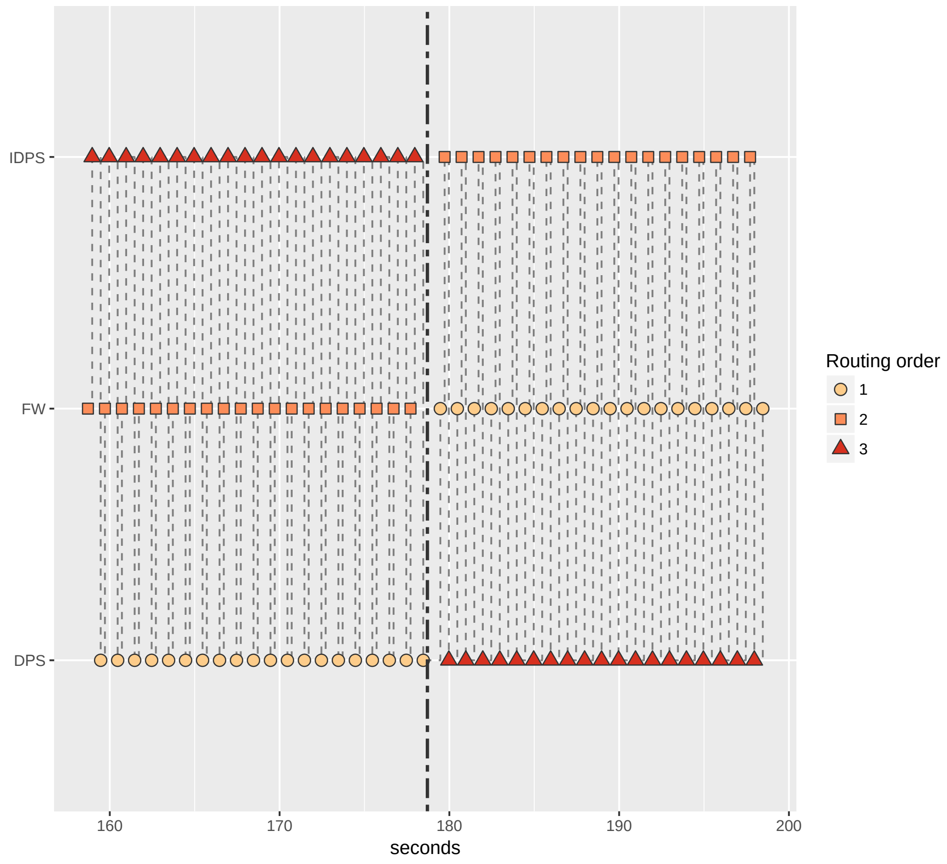


Figure 6.8: Change of routing configuration from DPS-Firewall-IDPS to Firewall-IDPS-DPS

Experiment Results

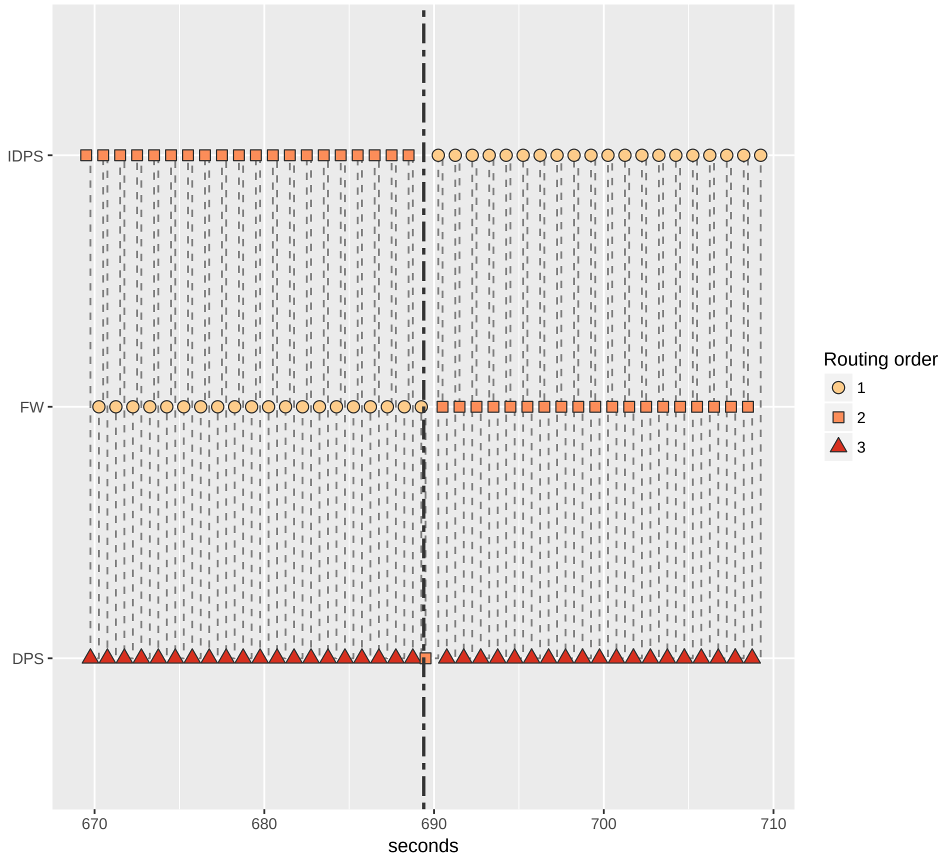


Figure 6.9: Change of routing configuration from Firewall-IDPS-DPS to IDPS-Firewall-DPS

We selected four instances during this experiment, where the routing configuration changes. Visualizing the complete experiment in one graph yields unreadable results. Therefore, we show only parts of the graph in the timeframe of 40 seconds around changes of the routing configuration to illustrate the functionality tested in the experiment.

Figure 6.8 illustrates the *imminent attack* functionality. The first modification of the routing configuration occurs at almost 3 minutes into the experiment. The routing order changes from *DPS-Firewall-IDPS* to *Firewall-IDPS-DPS*. This change could not originate from the regular check as it occurs before the five-minute mark. Listing C.1 and Listing C.2 list the corresponding flows for the Open VSwitch instances on `C39` and `C48`.

Figure 6.9 illustrates the regular functionality of the FCC. The routing configuration takes effect approximately 12 minutes after the start of the experiment. The routing order changes from *Firewall-IDPS-DPS* to *IDPS-Firewall-DPS*. Listing C.3 and Listing C.4 list the corresponding flows for the Open VSwitch instances on `C39` and `C48`. Here, it becomes visible that for one ping, the packets did not traverse every security function, posing a potential security risk. We provide possible solutions to this issue in the discussion.

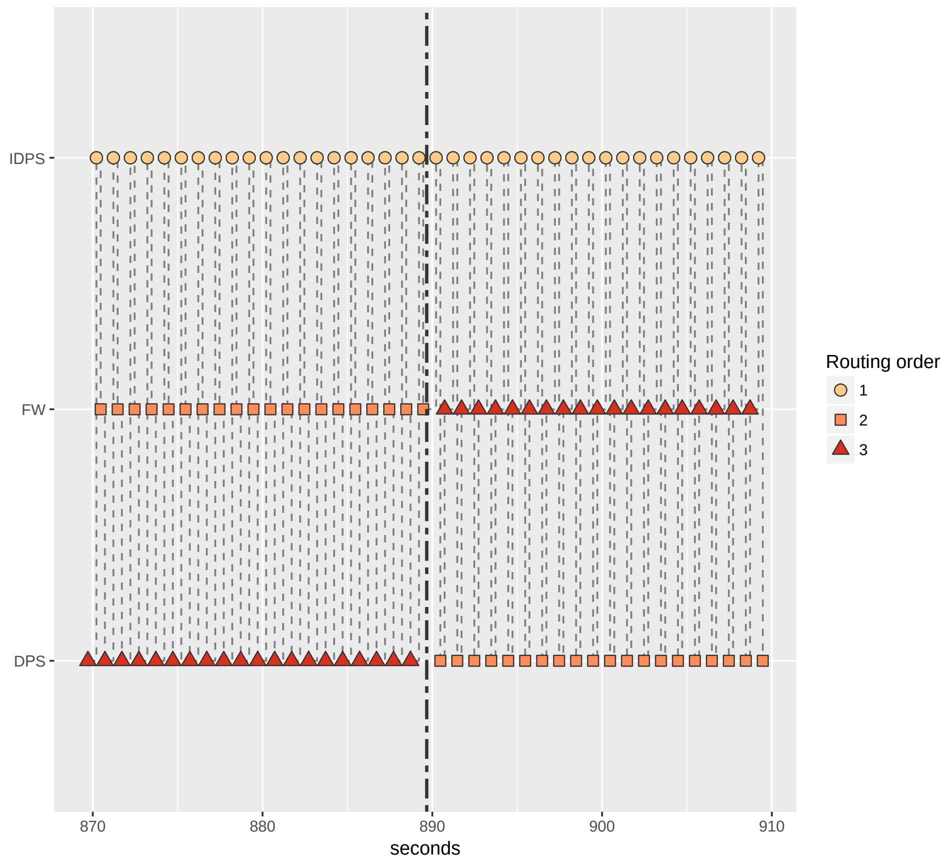


Figure 6.10: Change of routing configuration from IDPS-Firewall-DPS to IDPS-DPS-Firewall

Figure 6.10 shows that the framework applied the new routing configuration approximately 15 minutes after the start of the experiment due to an imminent attack on the IPS. The routing changes from *IDPS-Firewall-DPS* to *IDPS-DPS-Firewall*. Although IDPS was already the first instance the traffic traverses through, the DPS reported more attacks than the firewall and thus moved ahead of the firewall. Listing C.5 and Listing C.6 list the corresponding flows for the Open VSwitch instances on `C39` and `C48`.

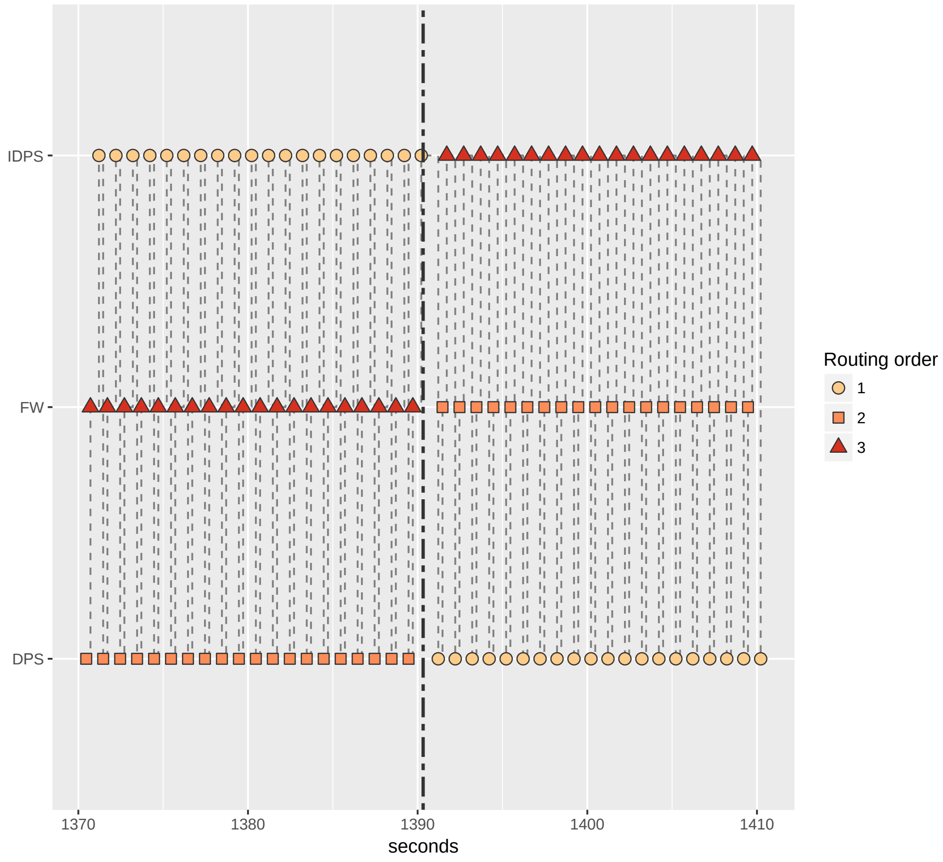


Figure 6.11: Change of routing configuration from IDPS-DPS-Firewall to DPS-Firewall-IDPS

Figure 6.11 shows a routing configuration applied approximately 23 minutes after the start of the experiment. The routing changes from *IDPS-DPS-Firewall* to *DPS-Firewall-IDPS*. *DDoS* and *FW* reported more attacks than the *IPS*, resulting in the corresponding configuration. Listing C.7 and Listing C.8 list the corresponding flows for the Open VSwitch instances on `C39` and `C48`.

The Function Chaining Controller resets the configuration to the standard configuration if reported attacks of all security functions are below the configured threshold. This reset to default allows users to select a default configuration that best fits the average attack on the system.

6.3.4 Discussion

Functionality

In summary, the developed framework is working as expected. Small issues like packet loss may occur during the application of new routing configurations. The generation of routes and their application works as desired. We also showed that the framework is indeed attack-aware and successfully changes the routing configuration of the network based on the reported attacks from the security functions. After attacks fade out, the framework switches back to the default configuration.

Security Issues During Reconfiguration

As shown before, three undesired scenarios can occur during reconfiguration: **RQ 6.3b**

(i) Packets get dropped because the framework has not installed the required flow yet. This packet loss requires retransmission.

(ii) Packets traverse one or more security functions more than once. E.g., in a chain with three functions, if a packet currently resides in the third function and a reconfiguration puts the third function first, the packet again has to traverse the former first and second function.

(iii) Packets do not traverse through all required security functions. E.g., when using a chain with three functions, a packet residing in the first function continues directly to the receiver, if a reordering puts the former first function last. Thus, the packet skips the former second and third functions.

The first and second issues only pose QoS and QoE concerns. Retransmissions (if supported by the protocol) take additional time. Similarly, passing through more security functions increases the time the packets spend in the SSFC. However, the third issue is relevant to security. If packets can skip security functions, single malicious packets can reach the receiver. This issue is of little concern for flood attacks, but for intrusions, a single packet might be enough to trigger a vulnerability and cause a severe security breach.

To avoid this issue, we propose several solutions:

- A straightforward solution would be to have a second set of security functions. Reconfigurations would then use this second set for the SSFC. Once all packets clear the security functions in the first chain, those functions become the functions, the next reconfiguration can use. Many security systems have hot spares to ensure the availability component of security. Therefore, for these systems, this solution is simple to implement and has no overhead. As a beneficial side-effect, this option also fixes the first and second issues.

- A second option is to model the stay of packets inside the security function by adding short-lived flows with artificial delays that ensure that no packets are inside the functions when executing the reordering. However, this requires detailed knowledge of all security functions inside the chain, especially regarding their queuing behavior. While precise modeling is possible using existing performance engineering tools (e.g., [SWK16]), when the source code is available, black-box models are more complicated to attain and less precise. This approach does not fix the first and second issues.

- The third concept is to force the security functions to drop all packets before executing the reordering. This solution fixes the second and third problems but moves the affected packets and others to the first issue.

- The fourth option is to use the options field in the IP header. We create a counter field in the options. For every reconfiguration, the SDN controller increments this counter. The inbound switch has a rule that modifies incoming packet headers to contain the current counter value. The flows match against this header field and the current counter. Older flows expire after some time (either by removing them manually with a clean-up routine inside the SDN controller or using soft timeouts. The main limitation of this approach is that many hardware switches do not support matching against this field. However, newer hardware switches and software switches in general support matching against this field.

Depending on the use-case, the security system architecture, the employed switches, and the used security functions, there are different optimal solutions. We have not yet implemented these solutions but will do so in the future. For the proof-of-concept, the presented implementation is sufficient.

6.4 Summary and Evaluation of Research Questions

In this chapter, we introduced a framework for attack-aware dynamic Security Service Function Chain reordering.

RQ 6.1: How to structure a framework for dynamic function chain reordering?

All security functions reside inside an SDN-enabled network. A *security function wrapper* co-located with every security function reports attacks at these functions via a separate management network to the FCC. The FCC computes the desired order for the security functions and submits it to the SDN controller, which enforces the order by creating the necessary flows on the SDN switches.

RQ 6.3a: What results does a proof-of-concept implementation provide?

We developed a proof-of-concept implementation using a simplified decision algorithm for ordering decisions and a minimal SDN controller tailored to our infrastructure. The framework shows that it can enforce all possible orders. We also put the framework through simulated attack patterns. The framework successfully adapted to all attacks and — after the attacks ceased — successfully restored the default configuration afterward. Thus this proved the desired functionality.

RQ 6.3b: Do new attack vectors, and other issues arise from dynamic function chain reordering – and if yes, how can these issues be addressed?

An issue occurs that during reordering, packets can drop or pass through a function twice. These effects are undesirable but not a security issue. More significant is the third issue: during reordering, a packet can skip functions. We proposed four options to combat this issue: (i) using a spare set of security functions, (ii) modeling the stay of packets inside the security functions, (iii) force the functions to drop enqueued packets, and (iv) using the options field in the IP header. The optimal solution depends on the existing infrastructure and use-case.

Chapter 7

Heat-aware and CPU Boost-oriented Server Load Rotation

Modern processors can exceed their designed clock rate for short time frames. Starting with Intel's Turbo Boost technology in 2008, these capabilities have evolved, and competing CPU manufacturers have adopted these technologies. At first, the idea was to increase only the clock rate of a single CPU core. A few years later, Intel extended Turbo Boost by additional variation levels. With many CPUs having more than two cores, it was sensible to create intermediary steps between boosting one core to maximum clock rate and having all cores running at the base clock rate. In addition to continually boosting a single core or few cores, Intel added the capability to exceed a CPU's thermal budget temporarily. Thus, it is possible to exceed the base clock rate for all cores for a limited amount of time. The deployed cooling solution determines this amount. Once exceeding a certain temperature threshold, the dynamic boost is disabled. This boost can be between around 20% (for servers) and over 100% (for ultra low power CPUs, e.g., Intel's Core Y-Series). Section 2.7 gives more information about Turbo Boost, its competitors, and thermal management.

Many works dealing with software performance see the turbo boost (similar to HyperThreading) as an unwelcome and unpredictable interference to their performance models and disable this feature. While this is a sensible choice to validate the applicability of performance models, it limits their real-life applicability since most servers in their default configuration come with Turbo Boost enabled in the Basic Input/Output System (BIOS), and only a few hosting providers allow to change that.

Instead of seeing the CPU boost as a nuisance, in this chapter, we work on harvesting its potential. Imagine an infrastructure with several homogenous compute nodes. On each of these nodes, an application is running, consuming around 60% of that nodes compute resources. We assume optimal consolidation for each of these applications. Now we want to add a single application that would require 50% of the resources of one of our hosts. Within the existing possibilities, we would have to boot up another host machine (or violate an SLA on purpose).

As we have learned about how CPU boosting technologies allow for a server to have around 20% higher clock rate than usual. Thus, on a boosted server, we could deploy the new application as long as the boost stays active. Unfortunately, the boost does not stay up indefinitely. This limitation brings us to the idea of heat-aware load balancing. When a host drops out of boost (or even better, when predicted to drop out of boost), we migrate the application to the host expected to stay boosted for the longest possible time. We continue this process as long as more resources are required than available.

We expect this approach to provide a constant boost in large environments where there is always a boostable server available. For smaller environments, this approach could help to quickly deploy the application and then later migrate it to a newly booted additional host. The contributions of our work are (i) an approach to heat-aware load balancing, (ii) a prototype implementation using software-defined networking, and (iii) the evaluation of the prototype's functionality and utility.

Research Questions

In this chapter, we will tackle several research questions. All of the following research questions are part of the meta-research question **MRQ 7:** *Can a CPU boost-oriented heat-aware server load rotation improve server performance?*. The numbering of these research questions maps to the sections of this chapter. If a section deals with more than one research question, those questions have their number appended by ascending Latin letters.

RQ7.1a Can SDN leverage potential in short-term CPU frequency boost technologies to increase the computing performance of a system?

RQ7.1b How to design an SDN-based load-balancing system with such capabilities?

RQ7.2 What existing solutions are suitable to implement this approach?

RQ7.3a To what extent do different workloads impact the approach (e.g., low load and high load)?

RQ7.3b Is it possible to extend this effect for more prolonged periods?

RQ7.3c How is the effect of this solution on power consumption?

Structure

In the remainder of this chapter, we first describe our approach in Section 7.1. Next, we detail our proof-of-concept implementation of said approach describing the components and the implementation of the algorithm in Section 7.2. We evaluate this implementation in Section 7.3 including a description of the evaluation environment, the validation of the desired functionality, the resulting behavior on real hardware, and a comparison to a system without heat-aware load balancing regarding performance and power consumption. Last, we summarize the chapter and answer the previously stated research questions.

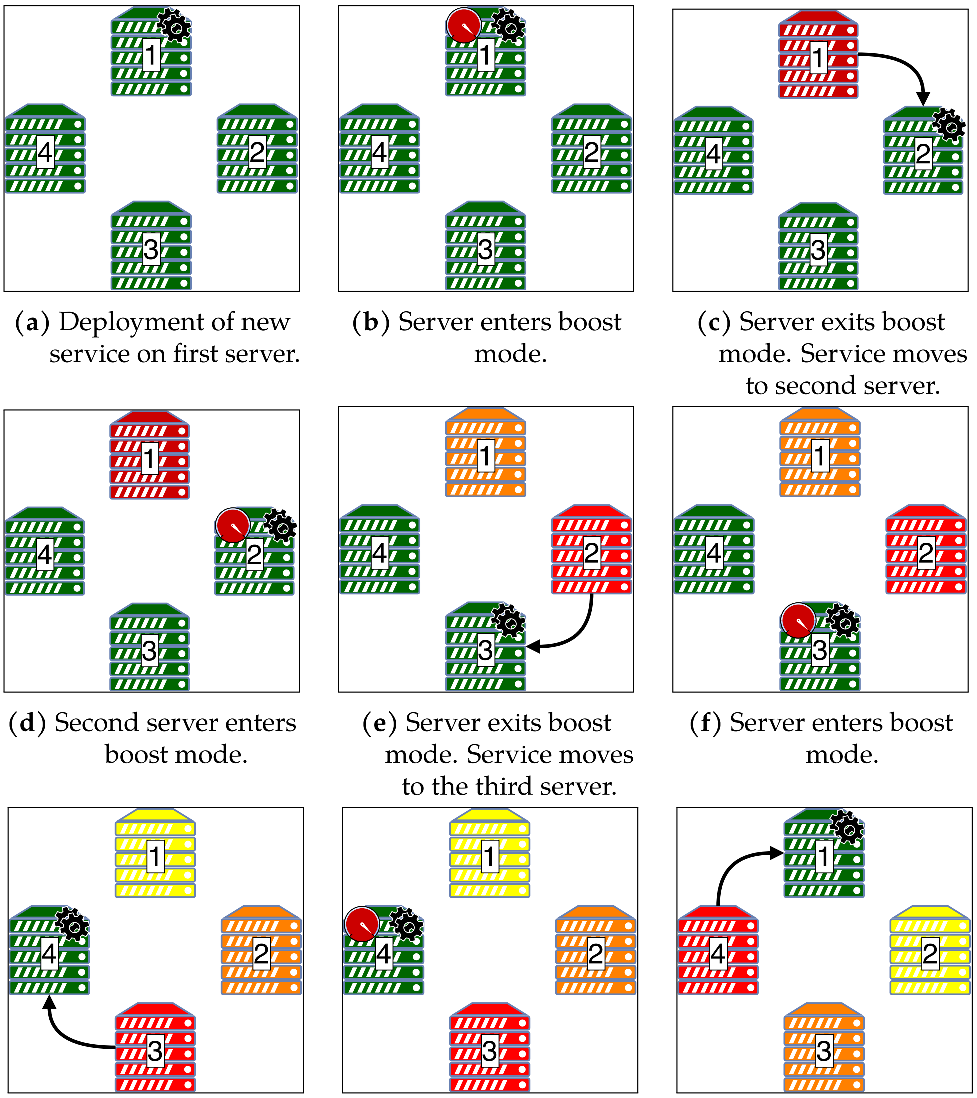
7.1 Approach

RQ 7.1a CPU boosting technologies allow for a short-time overclocking. For servers, this headway is around 20% of the CPU's base clock. In this section, we first present the concept. Next, we discuss how to realize this approach using SDN. Last, we take a peek at a simplified model for decision making.

7.1.1 Concept

Our approach follows the idea of having an application, always running on a boosted server. Figure 7.1 shows an exemplary behavior for a set of four servers. We assume that either the application alone creates enough load to trigger the server going into boost mode or that the server already has a high enough load that the new application can only fit when boosting the CPU frequency.

We deploy a new application on the first host in Figure 7.1a. This activity leads to the host's CPU switching to boost mode in Figure 7.1b. The boost mode results in a higher creation of heat than the cooling solution can transport away. Thus, after some time, the thermal budget of the CPU is exhausted (shown by the server turning red), and the CPU leaves its boost mode in Figure 7.1c. We simultaneously move the application to the next server to keep it on a machine at an elevated clock rate. Thus, this target server now enters its boosted state (Figure 7.1d). Again, at some point, the server exits its boosted state, and we move the application to the next server (Figure 7.1e). During the time the application spent on the second server, the first server cools down a little (represented by the server turning orange). The process repeats with boosting (Figure 7.1f), moving the fourth and last server, and the first and second server again cooling down (Figure 7.1g), and again with boosting it (Figure 7.1h). As shown by the heat icon in the figures, the accumulated heat level on the hosts



(**a**) Deployment of new service on first server.

(**b**) Server enters boost mode.

(**c**) Server exits boost mode. Service moves to second server.

(**d**) Second server enters boost mode.

(**e**) Server exits boost mode. Service moves to the third server.

(**f**) Server enters boost mode.

(**g**) Server exits boost mode. Service moves to the fourth server.

(**h**) Server enters boost mode.

(**i**) Server exits boost mode. Service moves to the first server.

Figure 7.1: Overview of a full rotation using heat-aware load balancing.

diminishes after exiting boost state and relinquishing the application to another host. At some point, the first host again is capable of going into boost state (represented by it turning green again). When the last host leaves its boost state, we assume that the first host is again ready to boost (Figure 7.1i). Thus, the application again moves to the first host. From thereon, it again continues through the rotation.

7.1.2 Realization Using Software-defined Networking

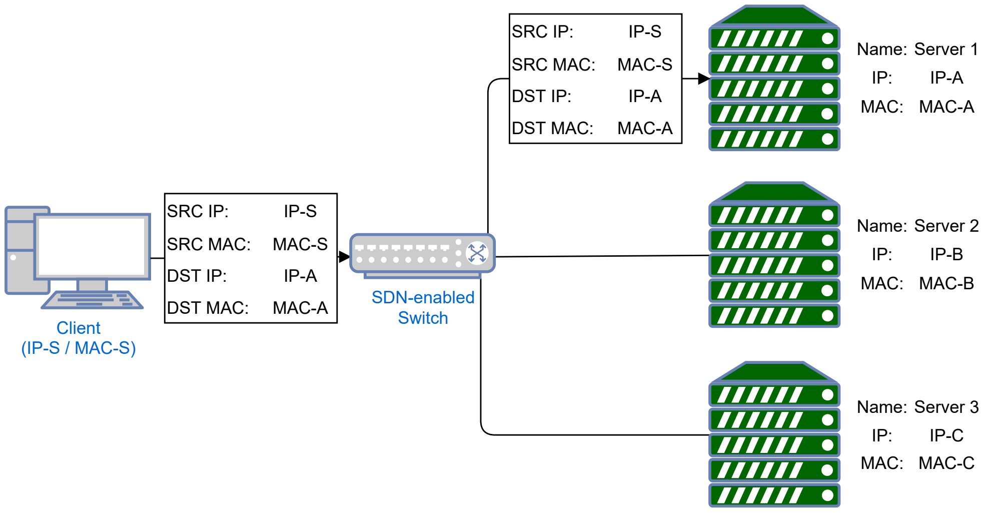
RQ 7.1b For the implementation of the heat-aware load balancing, we propose to use SDN. We introduced SDN in Section 2.4. SDN allows for the dynamic modification of network packets. Here, we use SDN to change the destination of requests to a service based on the machine's temperature and ability to go into or stay in boosted mode.

Therefore, we make a list of computing machines known as workers. Our application runs on each of these machines and listens to the active port. The application does not require a significant amount of resources when running without receiving queries.

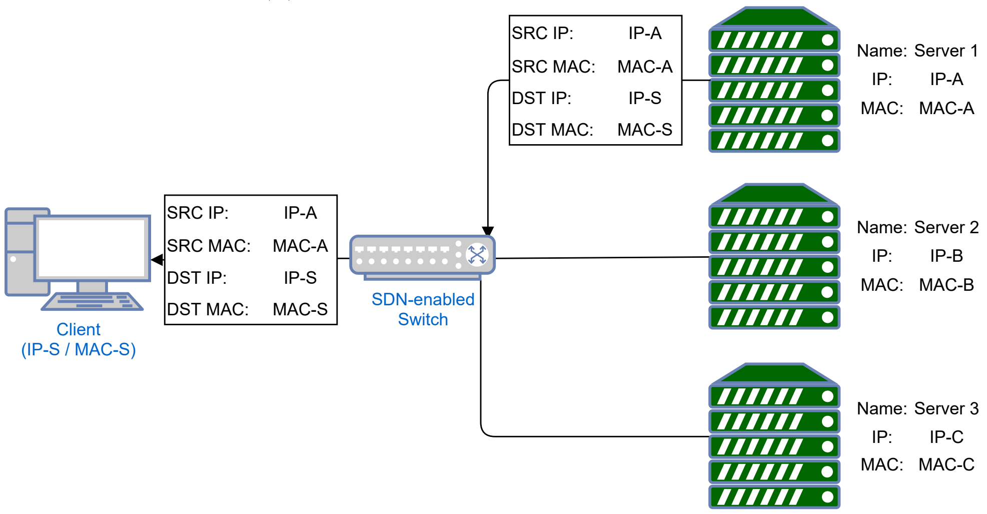
We demonstrate the algorithm for our approach on an example of one client and multiple servers. Figures 7.2 to 7.3 show the infrastructure for this example.

We define a default worker (here *Server 1*). The client always addresses this worker. When sending a request to the worker, this request reaches the SDN-enabled switch, as seen in Figure 7.2a. This switch requests a rule to create an SDN flow from the SDN controller. If, since the system start, the default worker has not gone into a state, where it is no longer able to boost, the new rules forward the packet to the default worker. Thus, no modification to the packet is necessary. The worker services the request and sends a reply to the client. Figure 7.2a shows the route of this reply. As seen, The reply already uses the correct addresses. Thus, the switch just forwards it to the client. As long as the server is not too hot to boost, this pattern continues. So far, the process does not differ from how the system would behave without an SDN switch.

Figure 7.3 shows the behavior, once the default worker has reached its thermal capacity limit and is no longer able to maintain boost mode. Still, the client addresses requests to the main machine. However, now, the switch receives a rule to forward the packet to the coolest server (here *Server 2*) from the SDN controller. Without modification, the new server would decline the packet, since its destination MAC and IP addresses do not match the server's addresses. Therefore, the received rule also contains instructions for the switch to rewrite the packet's header, so that it contains the new server's IP and MAC addresses, as shown in fig. 7.3a. Now, the server accepts the request, services it, and sends

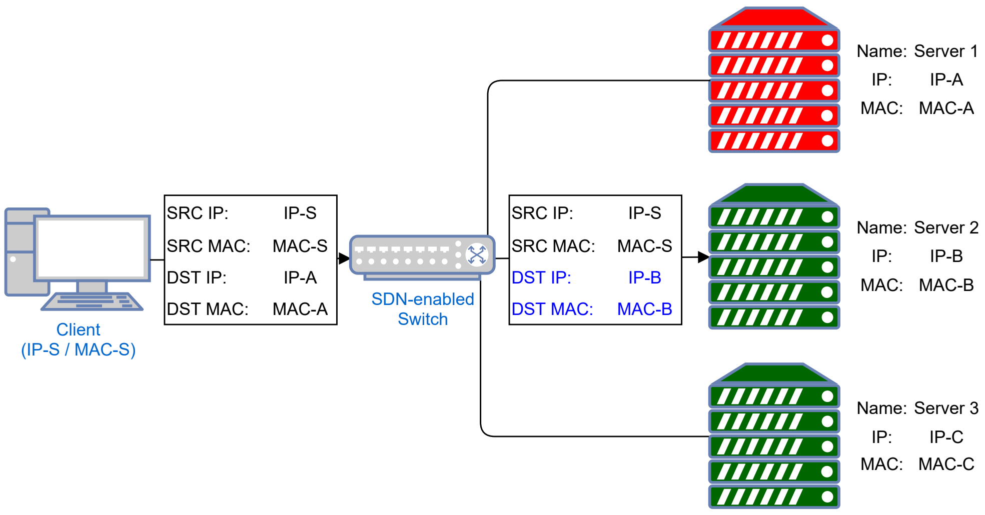


(a) Packet behavior from client to server.

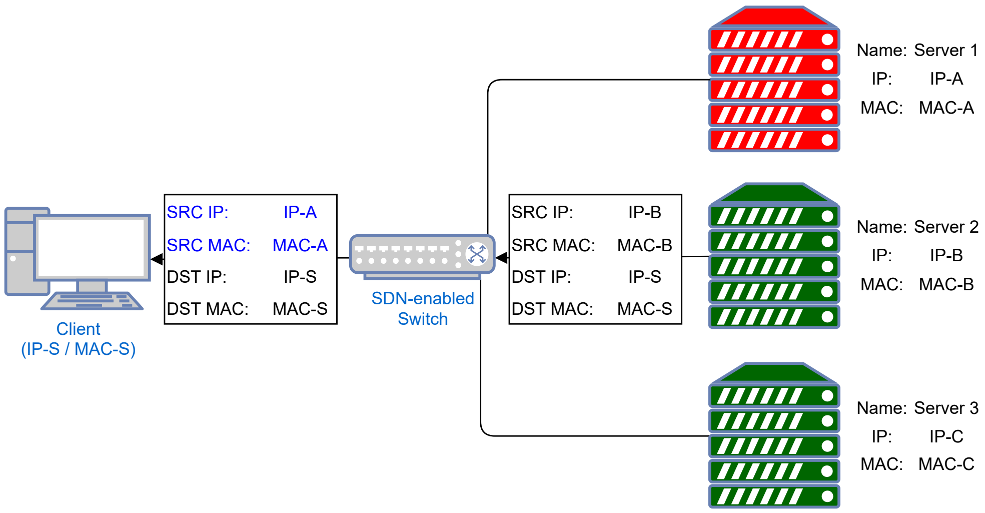


(b) Packet behavior from server to client.

Figure 7.2: Behavior of the SDN-enabled network without hot servers.



(a) Packet behavior from client to server.



(b) Packet behavior from server to client.

Figure 7.3: Behavior of the SDN-enabled network after the first server is to hot to go into boosted mode.

a reply. As seen in Figure 7.1d, the switch forwards this reply to the client. Again, without the modification, the client would discard the reply, since it comes from a server that it never contacted. So, again, the switch rewrites the packet to contain the source IP and MAC addresses from the main worker. Thus, the client accepts the reply. Further replies continue to the second server until it falls out of boost mode. Then, a new rule forwards the packet to the new coolest server.

To realize this approach, we use three algorithms: (i) an initialization script creating the static rules, (ii) a periodic monitoring script updating the desired target server, and (iii) an event-triggered script creating the dynamic rules.

Algorithm 7.1: Creating server to client rules on system initialization.

```
Data: List of workers, mainWorker, client
1  foreach worker ∈ workers do
2      create new SDN flow on the switch with
3          matching:
4              matching:
5              inPort = worker.connectedSwitchPort,
6              ethernetType = 0x800,                          /* TCP */
7              sourceIP = worker.IP
8              sourceMAC = worker.MAC ;
9          actions:
10             setFields:
11                 destinationIP ← client.IP
12                 destinationMAC ← client.MAC
13                 sourceIP ← mainWorker.IP
14                 sourceMAC ← mainWorker.MAC;
15             output:
16                 port = client.connectedSwitchPort;
```

Algorithm 7.1 presents the initialization script. For each server, it creates a rule that forwards TCP packets from the server to the client. Additionally, the rule replaces the MAC and IP addresses of any worker with the ones of the default worker. Thus, a client would perceive all replies by the other servers as replies from the default worker. These rules constitute the reverse paths only used by the servers' replies and not the answers to the client. Thus, these rules can be static. Static rules also have the advantage that they allow for a server to finish active queued requests still when it is no longer the active server for the load balancer. Therefore, the initial script can create the rules, and no dynamic resource-consuming and latency-generating solutions is necessary.

Algorithm 7.2: Monitoring loop.

```
Data: List of workers, mainWorker
Result: activeWorker
1  boostReady ← true;
2  counter ← 5;
3  while true do
4      while boostReady and counter > 0 do
5          if current server isBoosted then
6              boostReady ← false;
7          counter = counter - 1;
8          sleep for 1 second;
9      while current server is boosted do
10         sleep for 1 second;
11     activeWorker ← coolestWorker;
```

The monitoring algorithm, shown in Algorithm 7.2 runs permanently. It starts with the currently active worker. If this worker is not boosting when first entering the loop, it sleeps for five seconds or until the server is boosted. If, after that time, the server is still not boosted, the system checks if a cooler server is available. Suppose this is the case, the active worker switches to this server, and the loop restarts. If the server is boosting, the loop waits until the server exits this state. Once this happens, the coolest server becomes the active worker. Thus, once a server enters boost mode, the algorithm maxes out the possible performance from this mode until the server's thermal budget is exhausted. When no server is in boost mode, the algorithm selects the coolest server for new requests to ensure minimized cooling requirements.

The final algorithm is the PacketIn handler, as seen in Algorithm 7.3. The SDN controller calls this handler every time the switch receives the packet and has no existing rule that matches it. Because the switch has permanent rules for all packets from the servers, only packets from the client can trigger this function. Now, the SDN controller creates a new rule for the switch. This rule matches all packets from the client (regarding the incoming port on the switch, and the IP and MAC addresses) and rewrites the packets, so that their destination IP and MAC addresses match the target server selected by

Algorithm 7.3: PacketIn handler.

```
Data: currentWorker, client
create new SDN flow on the switch with
    timeout ← ruleTimeout ;                          /* from config */
    matching:
        matching:
        inPort = client.connectedSwitchPort,
        ethernetType = 0x800,                        /* TCP */
        sourceIP = client.IP
        sourceMAC = client.MAC ;
    actions:
        setFields:
            destinationIP ← currentWorker.IP
            destinationMAC ← currentWorker.MAC
        output:
            port = currentWorker.connectedSwitchPort;
```

the monitoring component. Last, the rule outputs the packet to the port, to which the server connects. This new rule has a configurable timeout. Packets from the client continue to the selected server for this preset amount of time. After the timeout expires, the algorithm reruns, allowing for an update of the target server. An alternative would be not to create a rule but decide every time a packet from the client arrives. However, this creates overhead at the SDN controller and latency for every packet. Therefore, a rule with a timeout allows for a configurable compromise between update frequency (and the time between the end of boost mode and the system's reaction) and performance metrics like latency.

7.1.3 Simplified Temperature Model

We previously used the term *coolest server*. We will evaluate our proof-of-concept implementation in Section 7.3 using homogenous servers. Therefore, their behavior when heating up as well as cooling down is similar, and just taking the coolest server is sufficient. However, for a more complex heterogeneous infrastructure, this solution might not be enough, since the coolest server might not be the server, that can maintain the boost mode for the longest time.

While state-of-the-art literature provides powerful cooling models from the level of chips up to complete racks or data centers [Bre+20] they require an excessive amount of information from the heat conductivity of the CPU cooler over the temperature of the inlets to the temperatures in neighboring servers or around the data center. Acquiring this information is often not feasible (e.g., consumers often just book a few height units in a data center and do not know about the neighboring servers) or very complex. Thus, we present a very simplified model for heterogeneous environments.

We describe the point at which the CPU goes out of boost mode as E_{boost_max} regarding energy and as T_{boost_max} regarding temperature. Since, as described before, energy models are very complex, we focus on the temperature. The point T_{base} marks the baseline to which the temperature can drop when the load-balanced service does not run on that specific server. The current temperature T, therefore, varies between the base temperature and the maximum temperature plus a tolerance ($\Delta T_{tolerance}$), since the migration of the service to the next server might have a delay. Equation (7.1) shows this relation.

$$T_{base} \leq T \leq T_{boost\_max} + \Delta T_{tolerance} \tag{7.1}$$

For simplification, assume a linear behavior when cooling and boosting. It possible to derive increase in temperature per second when boosting ($\Delta T_{boosting}$) and the decrease when cooling ($\Delta T_{cooling}$) by performing some simple experiments on the used servers by timing the time the boosted system takes to go from T_{base} to T_{boost_max}. Since the boost is active, when the server is at maximum load, a differentiation for multiple load levels is not required. This simplification leads to Equation (7.2) showing a formula to extrapolate the temperature if the system would go into boost mode or be left cooling.

$$T(t_2) = \begin{cases} T(t_1) + \Delta T_{boosting} \cdot (t_2 - t_1) & \text{when boosting} \\ T(t_1) + \Delta T_{cooling} \cdot (t_2 - t_1) & \text{when cooling} \end{cases} \tag{7.2}$$

In a heterogeneous environment, servers have different values for T_{base}, T_{boost_max}, $\Delta T_{boosting}$ and $\Delta T_{cooling}$. With these values known, an algorithm can now calculate which server can maintain a boost for the most extended amount of time. Furthermore, a more complex sophisticated could even predict the number of switches between servers for different orders. This prediction could lead to unintuitive decisions, where the server that could stay in a boosted state the longest does not yet become active to cool down further.

7.2 Implementation

We provide a proof of concept implementation. Additionally, we provide ansible scripts to deploy the application. **RQ 7.2**

Components

Our application has multiple components that interact: (i) a central monitoring component, (ii) distributed worker-side monitoring components, (iii) an SDN controller, and (iv) an SDN-enabled switch.

While the worker-side components must run on the worker machines, the other components can run on a single machine or spread over multiple machines.

Central Monitoring

The central monitoring component collects data from the worker machines. For this task, our choice fell to InfluxDB [Inf20a]. Chronograph visualizes the recorded data. We provide pre-defined dashboards for the visualization of the relevant data for our approach.

Worker-side Monitoring

To collect information regarding CPU frequency and temperature from the worker machines, we use Telegraf [Inf20b].

SDN Controller

We use Ryu as an SDN controller. Ryu is lightweight, supports basic switching and REST per default, and provides for a simplified extension using simple Python scripts. We realized the rule-settings algorithms (Algorithm 7.1 and Algorithm 7.3) as a Ryu module using OF. Algorithm 7.2 runs in parallel outside of Ryu since it is not directly involved when setting SDN rules.

SDN Switch

In general, any OF-1.3 compatible switch should suffice for our approach. However, many vendors do not provide the level of support for OF features that they claim. To ensure full OF compliance, we use Open vSwitch.

7.3 Evaluation

In this section, we evaluate the previously introduced proof-of-concept implementation. We first introduce the evaluation environment in its virtual and physical variants. This environment description contains information about the service and client software used for load generation. Next we validate the general functionality of our approach using the virtual evaluation environment. We then take a look at the behavior of the solution with varying workloads and load levels. Last, we assess the impact, heat-aware load balancing has on energy and performance of the load-balanced service.

7.3.1 Evaluation Environment

We use two different evaluation environments. Therefore, we first introduce the common aspects an then the specifics of both testbeds.

Composition

We evaluate our proof-of-concept implementation in an environment consisting of five servers and a switch. The servers act as (i) a client, (ii) an SDN controller (that also takes over the role of an experiment controller), and (iii) three service hosts. They interconnect with the switch as shown in Figure 7.4.

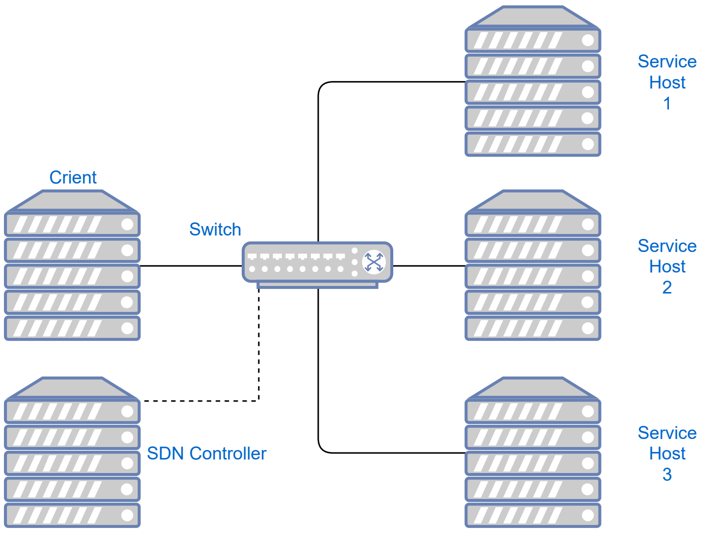


Figure 7.4: Connection scheme for both the virtual and physical testbed.

Load Driver

We use the *LU Workload* — a workload that creates heavy CPU load from matrix computations. It computes the lower- and upper-triangular forms of a matrix A, which means $A_L A_U = A$. The LU factorization's runtime for a Matrix A of size $n \times n$ is $\mathcal{O}\left(n^3\right)$ and its space requirement for data is $\mathcal{O}\left(n^2\right)$.

LU's mechanism is similar to Gaussian elimination. One variable is eliminated each step by subtracting lines from each other. This process makes the LU workload a CPU intensive procedure. In addition to the regular entry-by-entry computation, there is also a "blocked" version computing blocks to increase performance [CGS07]. Since we intend to create a load and not reduce the load per computation, we use the regular version.

We deploy the LU workload in the version from SPEC SERT [LT19] reimplemented in BUNGEE [Her+15]. Therefore, we deploy one instance of the BUNGEE LU worklet on every service host.

HTTP Load Generator [KDK18] generates requests to the worklets. The director runs on the experiment controller, and the actual load generator runs on the client. We use HTTP Load Generator's configuration file to configure the size of the matrix for the LU worklet and the frequency of the requests.

Switch

We use OpenVSwitch for both testbeds. The main reason is OpenVSwitch's excellent compliance with the OF standard. OpenVSwitch runs on ananother server (or VM in the case of the virtual testbed).

7.3.1.1 Virtual Testbed

To assert the basic functionality, we use a virtual testbed spawned by Vagrant [Has20]. Virtual machines replace all servers shown in Figure 7.4, and the switch and run on the host described in Table 7.1.

An issue with our approach in the virtual environment is, that — depending on the hypervisor — VMs do either not have data from temperature sensors or all show the host's temperature. Therefore, we introduce a small script that writes "fake" temperature values into our InfluxDB. It integrates with the load-balancing algorithm. When a service host is the active worker, its temperature increases by a configured value per second, and otherwise, it decreases by another configured value per second without going below a minimum value.

Unit	**Value**
Product	HP ProLiant DL360 Gen9
CPU	Intel Xeon E5-2640 v3
Default CPU frequency	2.60 GHz
Max CPU frequency	3.40 GHz
Min CPU frequency	1.20 GHz
Cores (Threads)	8 (16)
Cache (L1/L2/L3)	512 KB/2048 KB/20480 KB
Memory size	32GB (2 x 16 GB) DDR4 Dual Channel
Memory frequency	1.866 GHz
Memory Connection	Dual Channel
Storage Model	HP VK0800GEFJK 800 GB SSD
Storage Connection	SATA III (6GBit/s)
Operating System	Ubuntu 18.04.3 LTS (x86-64)
Kernel	4.4.0-72

Table 7.1: Specifications for the VM host.

7.3.1.2 Physical Testbed

For the evaluation of the impact and the real-wold behavior, we require a physical testbed. This testbed realizes the network shown in Figure 7.4 The connections between the client, the service hosts, and the switch are 10 GBit/s links while the connection between the switch and the SDN controller is a 1 GBit/s link.

We use standard DELL servers for the client, SDN controller, and the worker machines, as specified in Table 7.2. There are no modifications to the operating system and kernel. It is noteworthy that the second service host resides directly between the first and second service hosts in the rack.

For the switch, we use another DELL server with the specifications from Table 7.3. The default OpenVSwitch implementation is sufficient since compute power is the limiting resource and not network throughput.

The workers' power supply runs via a Yokogawa power meter to measure the power consumed by the workers. The idle power for all three machines together is 98.19 W or on average 32.73 W per machine.

Unit	**Value**
Product	Dell PowerEdge R210 II
CPU	Intel Xeon E3-1230 v2
Default CPU frequency	3.30 GHz
Max CPU frequency	3.70 GHz
Min CPU frequency	1.60 GHz
Cores (Threads)	4 (8)
Cache (L1/L2/L3)	64 KB/256 KB/8192 KB
Memory size	16GB (2 x 8 GB) DDR4
Memory frequency	1.600 GHz
Memory Connection	Dual Channel
Storage Model	HGST Ultrastar A7 K2000 500 GB@7200 rpm
Storage Connection	SATA II (3GBit/s)
1st NIC (Controller & Backend)	2 Port Broadcom Limited NetXtreme II BCM5716 Gigabit Ethernet
2nd NIC (Experiments)	Intel X520 10-Gigabit SFI/SFP+ Network Connection
Operating System	Ubuntu 18.04.3 LTS (x86-64)
Kernel	4.4.0-72

Table 7.2: Specifications for the client, the SDN controller, and the workers.

7.3.1.3 Application Scenarios

We use three different workload-frequency combinations for the LU worklet and the HTTP load generator.

We measured the the duration, a request to the LU workload takes on the workers in the physical testbed with turbo boost disabled in the BIOS. These results allow to establish a baseline for what the servers can handle. Thereby, we found the results shown in Table 7.4.

In the following we present the results for three scenarios — using a simplified ANOVA approach. We modify the frequency as well as the matrix size. Table 7.5 shows the selected scenarios. In any case, we set a ten second timeout for the SDN flows.

Unit	Value
Product	Dell PowerVault NX400
CPU	Intel Xeon E5-2420 v2
Default CPU frequency	1.90 GHz
Max CPU frequency	1.90 GHz
Min CPU frequency	1.20 GHz
Cores (Threads)	6 (12)
Cache (L1/L2/L3)	64 KB/256 KB/12288 KB
Memory size	16GB (2 x 8 GB) DDR4
Memory frequency	1.600 GHz
Memory Connection	Dual Channel
Storage Model	HGST Ultrastar A7 K2000 500 GB@7200 rpm
RAID	RAID 5 with four drives
Storage Connection	SATA II (3GBit/s)
1st NIC (Controller & Backend)	2 Port Broadcom Limited NetXtreme II BCM5716 Gigabit Ethernet
2nd NIC (Experiments)	Intel X710 10-Gigabit SFI/SFP+ Network Connection
3rd NIC (Experiments)	Intel X520 10-Gigabit SFI/SFP+ Network Connection
Operating System	Ubuntu 18.04.3 LTS (x86-64)
Kernel	4.4.0-72

Table 7.3: Specifications for the switch.

7.3.2 Functionality

Regarding the functionality of the load-balancing implementation, it is necessary, to assert, that (i) a change of the active worker occurs when desired, (ii) all packets reach their correct host, (iii) the switch carries out all packet header modifications correctly, (iv) the servers process all packets correctly, (v) no retransmissions occur, and (vi) the client reviews the replies to all requests and views them as successful.

We execute multiple requests in the virtual environment to assure these aspects of functionality and log their results. Also, we capture all packets on the interfaces of the client, the service hosts, and the switch. Last, we create

Matrix Size	Duration [s]
800	1
1200	5
1400	10
1950	30

Table 7.4: Duration per LU request for a given matrix size on the physical testbed.

Scenario	Matrix Size	Requests per Second
A	1 200	2
B	1 200	1
C	800	2

Table 7.5: Scenarios used for evaluation.

verbose output at the SDN controller.

Change of Current Worker

When the "fake" temperature of the current worker exceeds the value from which the algorithm no longer considers it as boosted, after the expiration of the current flow rule, the SDN controller's debug output shows that the controller correctly creates a new flow to the now coolest server.

Correct Switching

The packet captures show that after the change to a new service host occurs, the packets correctly reach this service host. Also, no new packets reach the previous worker. Still, as desired, the replies from the previous worker(s) to requests unprocessed before the switch correctly reach the client.

Valid Modification of Packet Headers

The dumps show correctly modified headers when the packets pass through the switch. Notably, they contain the correct destination IP and MAC addresses for packets from the client to the service hosts and the correct source addresses for packets from the service hosts to the client. Furthermore, all checksums are valid.

Correct Processing of Requests

Since the target addresses of packets reaching the service hosts match their addresses, the hosts accept the requests. They process all requests and send the correct reply.

Absence of Retransmissions

The dumps show that no request retransmission occurs. Since HTTP Load Generator uses TCP, this means, that, also, no packet losses occurred.

Reply Acceptance by the Client

The client receives the correct replies to all sent requests. However, the replies are not always in the same order as the requests. This behavior is typical for load-balancing approaches and not an issue.

Overall Functionality

The experiments show that the proof-of-concept implementation meets all desired functionality requirements. Thus, it meets its functionality goals.

7.3.3 Scenario-dependant Behavior

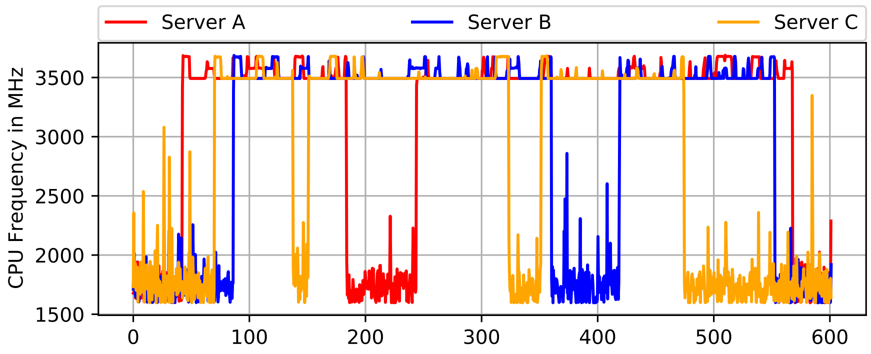
On the physical testbed, we enable our algorithm on the SDN controller. We then execute the workloads described in Section 7.3.1.3. We measure the temperature and the maximum CPU frequency over all cores per server at a time. The maximum CPU frequency over all cores F_{core_max} results from the following Equation (7.3):

$$F_{core\_max}(t) = \max\left(F_{core\_0}(t), F_{core\_1}(t), \ldots, F_{core\_7}(t)\right) \tag{7.3}$$

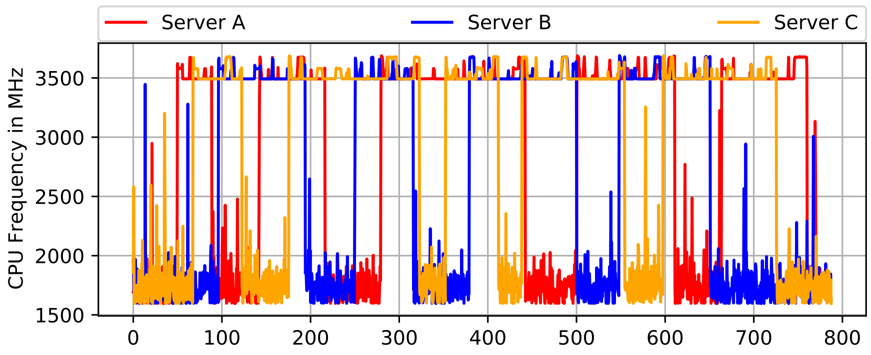
Scenario	Requests[%]		
	successful	failed	remaining
A	45.7	6.3	48.0
B	88.9	0.5	9.6
C	100	0.0	0.0

Table 7.6: Result of the requests after experiment termination.

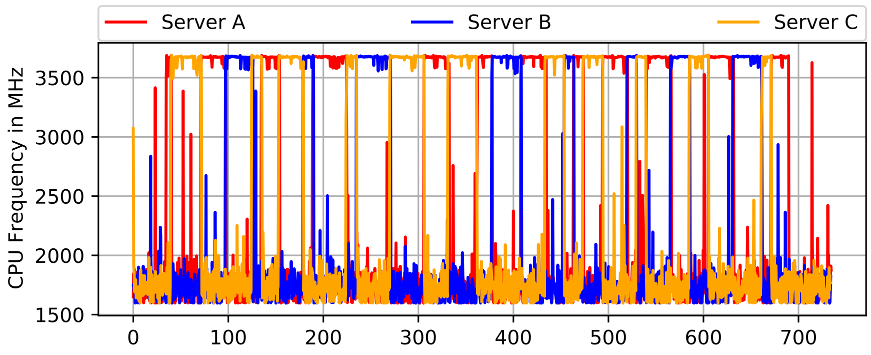
RQs
7.3a
7.3b



(**a**) Scenario A

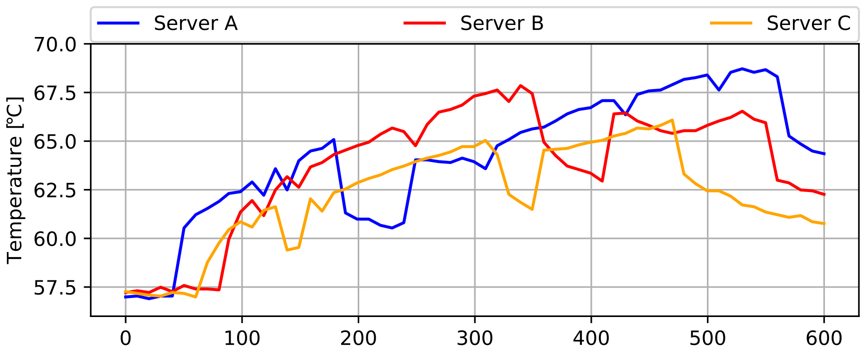


(**b**) Scenario B

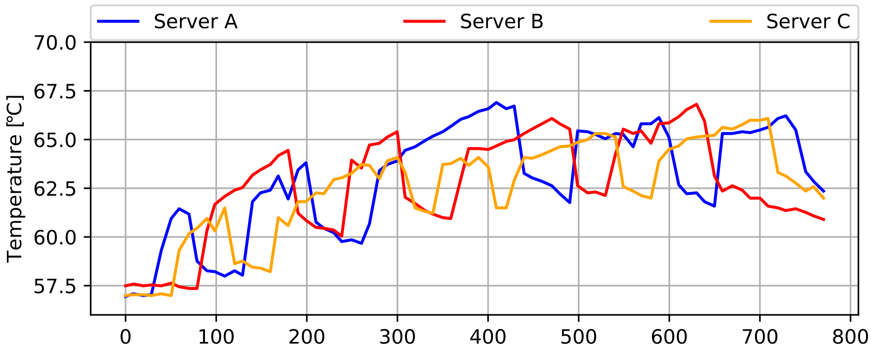


(**c**) Scenario C

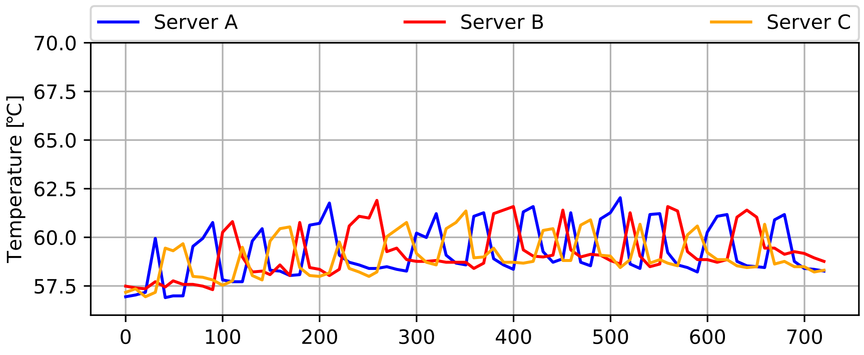
Figure 7.5: Highest core frequency during the experiment.



(**a**) Scenario A



(**b**) Scenario B



(**c**) Scenario C

Figure 7.6: Temperature during the experiment.

For each scenario, we measure and present these values for the time of the experiment. Figures 7.5 to 7.6 visualize these results. The visualizations also show a short time before and after the experiments. Thus, they include the ramp-up and the ramp-down phases, including the processing of remaining queued packets. At the end of the experiment, we inspect the log from HTTP Load Generator. From there, we take the number of total requests, successful requests, failed requests (e.g., the unexpected closure of an active TCP connection), and requests not processed until the end of the experiment. Table 7.6 shows the normalized results for all three scenarios.

7.3.3.1 Scenario A: Five Second Workload Every Half Second

In Scenario A, we create requests, which usually would take five seconds to process with turbo boost disabled, every half second. This load results in an overbooking of one server by a factor of ten and for the available three servers of factor 3.3. Thus, the workload significantly exceeds the server's capacity.

As seen in Figure 7.5a, most of the time, at least two servers are in an active state. Thus, our original idea of only one active service host does not apply in this scenario. Since the servers stay in an active state for a longer time, they only maintain a boosted state of above 3 500 MHz for a short time and spend most of their active time at their maximum non-boosted frequency.

Figure 7.6a shows the temperature over all servers. This temperature rises significantly throughout the experiment with only short dips that, with few exceptions, always end up higher than the previous ones. The servers reach temperatures of over 68 degrees.

The first row in Table 7.6 shows that at the end of the experiment, multiple requests remain unanswered. Thus, they still reside in the servers' queues. Still, while the servers should have been only able to process around 30% of the requests, they accomplished 45.7%, which suggests an impact of the heat-aware load-balancing, which we will discuss in Section 7.3.4.

7.3.3.2 Scenario B: Five Second Workload Every Second

Section 7.3.3.2 showed, that the idea of cooling servers back down only works to a limited extent when heavily overbooking said servers. Thus, in Scenario B, we half the request frequency to one per second. The requests themselves stay as before. So, the number of requests would overbook a single non-boosted server by a factor of five and three servers by a factor of 1.7.

Figure 7.5b shows that the servers more frequently alternate between active and passive states. Most of the time, at least one server and sometimes even two servers are in passive mode. However, this scenario still does not realize the idea of having exactly one active server. Also, the time spent by the server in boost-mode when active significantly increases compared to Scenario A.

Figure 7.6b explains this fact by an overall lower temperature by over two degrees. The servers seem to reach a temperature limit that they do not exceed afterward. However, the temperature still significantly increases relative to the idle state.

The second row in Table 7.6 shows that a much higher number of requests received successful replies. It is of little surprise that the shares of successful requests nearly doubles when halving their frequency. Also, the failed share massively decreases, but still, almost ten percent of requests remain in the queue. Since the servers together without boost would only process 60% of the requests, these results again suggest a positive impact on performance.

7.3.3.3 Scenario C: One Second Workload Every Half Second

In Sections 7.3.3.1 to 7.3.3.2, we analyzed scenarios that heavily overbook the cluster. In this scenario, we analyze a workload that does not overbook the whole infrastructure. We send a request that without boost would take one second every half second. So, while this overbooks a single server by a factor of two, it only uses two-thirds of the capacity from the three servers put together.

Figure 7.5b shows the servers almost perfectly alternating between their active states. Most of the time, one server is active, and the other two are in passive mode. Furthermore, the active servers spend all of their time in a boosted state. While the servers can not maintain the highest boost-level of 3 700 MHz all the time, during the remainder, intermediary boost levels are active. Thus, this scenario accurately characterizes our initial idea of one active server for the application. It is noteworthy that Server B, located between Server A and Server C, has a little less active time. A possible reason is the heat from the neighboring servers.

In Figure 7.6b, the temperature of all machines remains at a low level and never exceeds 62 degrees. The graphs show not only a lower temperature but also that the temperature behaves similarly to the CPU frequencies and the hosts alternate regarding the highest and lowest points.

The last row in Table 7.6 shows that all requests completed. This result is the expected behavior since the cluster is not overbooked. Thus, for this scenario, throughput is not a very relevant metric, and a basic load-balancing algorithm would have achieved the same result.

7.3.4 Energy and Performance Impact

In the previous section, we analyzed the behavior of the algorithm. When taking a look at the number of successful requests during these experiments, they suggest a positive impact of heat-aware load balancing on the performance — here throughput. In this section, we take a closer look at performance metrics and also the energy consumption of the machines during the experiments. We repeat the experiments from Scenario B and Scenario C. We perform them once without load balancing on a single server and once with load balancing enabled. We measure (i) the response time, (ii) the number of sent, received, and lost requests, (iii) the maximum, and average temperature, (iv) the average maximum and the average CPU frequency, and (v) the average combined power consumption of all servers. The term *average maximum CPU frequency* is not entirely self-explanatory. It results from (i) measuring the CPU frequency of all servers over time as three vectors, (ii) for every point in time taking the highest frequency, one of the active servers has and storing them into a single vector, (iii) computing the average of this vector. Since in our experiments, always at least one server is under load, this value indicates the average boost-frequency. For the power consumption and the frequency, we also provide the standard deviation. Here, this is no metric for the accuracy of our measurement but shows to what extent the metrics vary throughout the experiment. For example, the frequent use of turbo boost leads to a higher standard deviation. **RQ 7.3c**

7.3.4.1 High Load Scenario

Table 7.7 shows the measured metrics for the high load scenario. The heat-aware load balancing reduces the average response time by 80.2% — a significant reduction. While a tripling of the resources leads the assumption of a reduction by two thirds, the heat-aware approach yielded an even more massive reduction. The relative share of successful requests more than doubled (+136.8%) — as expected from the added resources — and no requests fail anymore.

The maximum temperatures of all three hosts are significantly lower (between 4.41 K and 4.89 K) than the one of the single machine. This effect increases when looking at the average temperature, which is between 6.03 K and 6.42 K below the single-server setup. These reduced temperatures contribute to extending the servers' live.

Since temperature correlates to the CPU frequency, it is of little surprise, that also the average CPU frequency is lower for the load-balanced setup. However, the average maximum CPU frequency for the load-balanced setup exceeds the one from the single-machine setup.

Metric	Heat-aware Load Balancing				
	disabled			enabled	
Server	A	B	C	A	
∅ response time[s]	4.62			22.25	↓
requests sent/received/lost	630/599/0			630/253/32	*o*/ ↑ / ↓
Max temperature[°C]	65.40	65.12	64.90	69.81	↓
∅ temperature[°C]	62.83	62.56	62.54	68.86	↓
∅ CPU freq[MHz]	3007	2980	3057	3504	↓
∅ max CPU freq[MHz]	3594			3524	↑
σ max CPU freq	178.37			58.92	*o*
∅ power consumption[W]	191.23			151.39	↓
σ Power consumption[W]	24.17			3.73	*o*
Δ power consumption[W]	+93.04			+53,20	↓
Δ power consumption[%]	+94.76			+54,18	↓

↓: lower values are better; ↑ higher values are better.

Table 7.7: Comparison switched and non-switched with five second workload.

This metric indicates that a longer time (almost half of the time), a servers is in boost-mode. When using three servers to perform a workload that heavily overbooks a single machine, it yields a higher power consumption than for one machine. Nevertheless, the increase in power compared to the idle state is less than double the increase from using only one server, yielding a better ratio between additional successful requests and extra energy consumption.

Since the heat-aware load balancing relies on the usage of turbo boost, the standard deviation of CPU frequency and power consumption is significantly higher. The differences between the servers likely depend on their placement in the rack.

In summary, the heat-aware load balancing increases performance and reduces the average and maximum temperatures of the employed servers. At the same time, the power consumption grows slower than the number of used additional compute resources would suggest.

7.3.4.2 Low Load Scenario

This scenario diverges from the previous one, as seen by the number of successful requests in Table 7.8 for the single-machine setup. The single machine

Metric	Heat-aware Load Balancing				
	disabled			enabled	
Server	A	B	C	A	
∅ response time[s]	0.85			1.83	↓
requests sent/received/lost	1310/1310/0			1310/1304/6	*o*/ ↑ / ↓
Max temperature[°C]	61.99	61.13	61.35	69.76	↓
∅ temperature[°C]	59.29	58.81	59.13	65.96	↓
∅ CPU freq[MHz]	2460	2148	2334	3518.44	↓
∅ max CPU freq[MHz]	3670			3580	↑
σ max CPU freq	23.47			70.97	*o*
∅ power consumption[W]	135.54			147.98	↓
σ Power consumption[W]	5.91			6.59	*o*
Δ power consumption[W]	+37.35			+49.79	↓
Δ power consumption[%]	+38.03			+50.71	↓

↓: lower values are better; ↑ higher values are better.

Table 7.8: Comparison switched and non-switched with one second workload.

is almost capable of servicing all requests limiting the expectable performance increase from load-balancing.

However, the load-balancing still cuts the response time by 54.6% and eliminates the remaining failed requests, so that all requests terminate successfully. The reduction of the maximum temperature lies between 6.83 K and 7.15 K and is more significant than in the previous scenario. Also, the average temperature drops by between 6.83 K and 7.15 K (precisely the same level as the maximum temperature).

The average CPU frequency decreases linked to the temperature, for the load-balanced scenario. On the other hand, the average maximum CPU frequency reaches 3 670 MHz for the load-balanced setup. Thus, it is 90 MHz higher than for the single machine. This result confirms the observation from Figure 7.5c that at least one CPU is always in a boosted state but sometimes not at the highest boost level.

Astounding is the result of the power consumption. The Turbo boost is known to increase power consumption significantly. Nevertheless, the average power consumption decreases. Thus, using three servers creates less extra power consumption relative to active standby than using one server. While this result is not intuitive, we still found a possible explanation: The reduced temperature leads to a reduced fan activity. We measured standard server fans and found them to draw around 10 W at the maximum level. Therefore, we assume that, while the boost when a server consumes more power, the reduction in required fan-power in the pauses between active phases more than outweighs this increase.

In total, the heat-aware load balancing leads to better performance metrics, lower temperatures, and reduced power consumption. Thus, the application for low-load scenarios has no drawbacks, as long as the other servers are already available, e.g., for use as hot spares.

7.4 Summary and Evaluation of Research Questions

In this chapter, we presented a solution for heat-aware load balancing. This solution allows us to maximize the time active CPUs spend in the state of a short-term frequency boost.

RQ7.1a Can SDN leverage potential in short-term CPU frequency boost technologies to increase the computing performance of a system?

A heat-aware load balancer must detect the moment, a server falls out of its boost mode and then move the service to the next server. This load balancing allows to increase the amount of time, the service spends on boosted servers. Thereby, the performance of the service can increase.

RQ7.1b How to design an SDN-based load-balancing system with such capabilities?

The solution consists of two main components inside an SDN-enabled network. A monitoring component watches the states of all workers and collects information on their capability to go into boost mode. The SDN controller then uses this information to decide on which server to position the service. Therefore, the SDN-enabled network carries flow rules that modify packets so that to a client, multiple servers appear as one server by modifying IP and MAC addresses in the packet headers.

RQ7.2 What existing solutions are suitable to implement this approach?

The InfluxDB database can provide significant parts of the monitoring components. The Ryu SDN controller is extendable to take over the part of the decision-making controller. Then, a single script can link both components together.

RQ7.3a To what extent do different workloads impact the approach (e.g., low load and high load)?

When the load is higher, more systems at the same time are active. This correlation leads to an increase in average and maximum temperature. Hot systems can not enter a boosted state. Thus, the higher the load, the less time the systems in total spend in a boosted state.

RQ7.3b Is it possible to extend this effect for more prolonged periods?

For higher loads, the temperature increases over time since the cool-down periods are too short. Thus, at some time, the systems are no longer able to enter the boosted state. It is possible to extend the time, where the boost works, by either increasing the cooling or adding more service hosts. However, for lower loads, the cool-down periods are sufficient, and the approach would work indefinitely.

RQ7.3c How is the effect of this solution on power consumption?

In a high-load scenario, the heat-aware load balancing significantly increases the system's power consumption. Nevertheless, the relative increase is lower than the relative performance gain. Surprising results came from the low load scenario, where heat-aware load balancing not only improves the performance metrics and decreases the temperature levels but also reduces the total power consumption. We assume that this reduction originates from reduced fan speed, outweighing the increase from the use of turbo boost.

Chapter 8

Signature-based Database Ransomware Detection

In today's era of digital transformation, data has become more critical than ever before. The amount of data we produce daily is astonishing — every day, hundreds of millions of people are taking photos, make videos, and exchange messages. Furthermore, data is not only an asset for users nowadays, However it has also become the global key component of digitization and transformation of today's businesses globally. Enterprises collect data on consumer preferences, purchases, and trends and use them to optimize their business models and strategies. Given such trends, the importance of database security is hard to overestimate — the rapid growth of the data volume stored in the databases of service providers, in cloud environments and enterprise data centers, as well as their increasing importance, make them attractive attack targets.

Traditionally, attacks on data have aimed to undermine confidentiality and authenticity. More recently, however, attacks against the availability of data, services, and users have become common as well — modern attackers deploy ransomware, malicious software that encrypts data and holds the decryption key until the victim pays a ransom.

They still claim the ransom pretending to have encrypted the data. The financial loss from ransomware is significant — it reached 5 billion USD in 2017, and predictions see it hit 11.5 billion by 2019 [Mor17].

While the first ransomware attacks targeted client platforms (information stored in users' files), recently, such attacks leaped to server-side databases that store, accumulate, and process (big) data. In January 2017, an attack called MongoDB Apocalypse [Cim17c; Cim17d] hit tens of thousands of MongoDB servers, followed by a second attack wave targeting MySQL servers [Ziv17]. Since then, server-side ransomware attacks spread to a wide range of server technologies, including ElasticSearch [Cim17e], Cassandra [Cim17a], Hadoop and CouchDB [Cim17b].

The typical attack scenario of server-side ransomware observed so far is as follows: First, an attacker gains remote privileged access to the database through the exploitation of configuration vulnerabilities such as the usage of default passwords [1]. Once connected, they execute commands for data enumeration (e.g., to learn names of databases and tables hosted), then drop (delete) data and insert the ransom message with instructions on how to pay the ransom. Remarkably, in contrast to client-side ransomware, the new attack form wipes the data without making any plaintext or encrypted copy, e.g., acting as a *wiper*. This strategy has, on the one hand, more dramatic implications for the victim, since the data is unrecoverable even when paying the ransom. On the other hand, the attack is stealthier without requiring intensive and easily detectable operations. Such operations include bulk encryption or massive data copying. Furthermore, the attack requires no backchannel to the attacker needed (e.g., for delivering the decryption key or recovered data) that could allow to trace them back.

While server-side ransomware is more recent, and to this day, less widespread than client-side ransomware, there are reasons why the situation might change quite soon. First, enterprises can afford to pay higher ransoms than private users. As a comparison, the typical ransom amount for regular users lies in the range of a few hundred dollars. However, businesses can pay much more – for instance, in a recent attack, a Los Angeles Hospital paid USD 17 000 ransom to attackers [CS16].

Second, in recent years, researchers and antivirus companies developed countermeasures against client-side ransomware. However, to date, no solutions exist against ransomware targeting database servers. This lack of protection makes databases easy attack targets.

Note that there is evidence that even though server-side ransomware is a wiper, some desperate victims paid the ransom, nonetheless. We identified that two known ransomware addresses involved in MySQL attacks [Ziv17] received 0.6 BTC (equivalent to 3 payments). For the attacks against MongoDB, we identified a total of 160 ransom payments to the addresses collected in [Cim17c], totaling in 26.35 BTC. Moreover, the survey reveals that even production systems lack sufficient protection by strong passwords and sensible backup strategy: Among 123 surveyed ransom victims, only 11% had recent backups, and 8% paid the ransom.

[1]Note that default passwords and other configuration errors are prevalent real-world problems. For instance, Mirai botnet [Ant+17] used similar vulnerabilities to take over more than 600,000 IoT devices arond the globe.

The survey also reveals that even production systems lack protection by strong passwords and sensible backup strategies.

Existing anti-ransomware solutions aim at the detection of client-side ransomware only. They follow two dominant strategies: Signature-based detection of malicious binaries and runtime monitoring and behavioral analysis for anomaly detection. The first one builds upon the detection of malicious binaries and is typically used by antivirus vendors, while the second strategy originates from research papers [Con+16; Con+17; Kol+17b; Sca+16]. It relies on runtime monitoring of file accesses and the detection of malicious activity based on heuristics, such as access to multiple files, their modification, and renaming. Unfortunately, both strategies do not apply to the Since in the server-side ransomware attack scenario, an attacker connects to the database remotely, there is no malicious binary on the platform that could be detected. Furthermore, monitoring at the file system level for abnormal activity is not adequate either since there is no direct correlation between an attacker's activity and file access patterns.

Password-based authentication suffers from problems such as weak user-chosen passwords, password reuse, and phishing. Nonetheless, despite over 30 years of research on better authentication methods [Atc87], passwords prevail due to usability reasons. Most database systems (e.g., MySQL, MongoDB, ElasticSearch, Cassandra, Hadoop, CouchDB) use password-based authentication. Hence, we consider secure user authentication as an orthogonal problem and aim to design a solution that preserves compatibility to systems in use. Moreover, even given more robust authentication methods, attackers still have other attack techniques in hand, which enable them to gain privileged access, such as exploitation of privilege escalation vulnerabilities in MySQL statements [Gol17].

Unfortunately, such solutions do not make transparent if they are limited to the analysis of individual queries and their expressions or do also consider attacks using query sequences. Note that the detection of server-side ransomware would require an analysis of query sequences. For instance, it would be necessary to detect the combination of the individual queries to list and drop a table, as well as the insertion of a ransom message (i.e., giving instructions on how to pay the ransom). Using only rule-based methods to detect ransom messages will produce a high number of false positives, as content might include trigger words such as BTC or a Bitcoin address. Moreover, these solutions concentrate on the analysis of queries within user sessions in order to detect behavioral anomalies, so malicious queries spread over multiple connections might evade detection.

In this chapter, we aim to improve the security of database systems and propose Dynamic Identification of Malicious Query Sequences (DIMAQS), a signature-based intrusion detection tool that can detect sequences of malicious queries. Generally, the tool is not limited to ransomware detection. It can potentially be applied to the detection of other attack classes as long as they rely on malicious sequences of queries (e.g., advanced SQL injections aiming at removing code execution [Dzu09]). However, motivated by the rise of server-side ransomware, we apply it to this problem. We make the following contributions:

We provide the design and implementation of DIMAQS, a framework that can detect sequences of malicious queries. To keep track of queries and to perform detection, our solution leverages CPNs to model the series of events used in attacks and to match them to known malicious patterns. Our system design exhibits several novel techniques (dynamic creation of colors, merging of tokens, and token expiration) to reduce the complexity of the system representation and achieve better performance. Our framework performs system-wide monitoring and, as such, can detect malicious sequences injected through several user sessions and interleaved with benign queries. This quite exciting feature eliminates the most apparent evasion strategies. Our implementation targets MySQL, one of the most popular database management systems, and imposes only a very moderate performance overhead under 5%. We realize our solution in the form of a MySQL plugin that is easily installable on existing MySQL servers, thus preserving compatibility with legacy software.

We apply DIMAQS to the challenging problem of server-side ransomware. To make detection of such attacks possible, we analyze previously observed attacks and extract their distinctive properties that provide a basis for attack detection. We then evaluate the effectiveness and practicality of our solution using three data sets: Malicious data set recorded by us, and benign query sets from a publication management system and a MediaWiki server. The results demonstrate the high efficiency of our approach with no false negatives or false positives.

Upon attack detection, our tool makes a temporary backup of information erased by an attacker and notifies a database administrator. If true positive, an administrator can restore all the data without paying a ransom to an attacker. If false positives, backup is simply deleted after some pre-defined amount of time. We evaluate the effectiveness of our tool and report promising results.

Research Questions

In this chapter, we tackle several research questions. All of the following research questions are part of the meta-research question **MRQ 8:** *How to apply signature-based intrusion detection to multi-query database ransomware attacks?* The numbering of these research questions maps to the sections of this chapter. If a section deals with more than one research question, those questions have their number appended by ascending Latin letters.

RQ 8.1a How to model multi-query database ransomware attacks?

RQ 8.1b Which components does a multi-query database IDPS require and how do they interact?

RQ 8.2 How to integrate a prototype multi-query database IDPS into a MySQL server?

RQ 8.3 How does the multi-query database IDPS perform in terms of security and performance?

Chapter Structure

The remainder of this chapter is structured as follows. In Section 8.1, we present the system design of DIMAQS. In Section 8.2, we reveal the details of our prototype implementation. Next, Section 8.3 discusses prototype evaluation results. In this section, we discuss the evaluation results. Section 8.4 summarizes the section and answers the previously stated research questions.

8.1 Approach

DIMAQS is the first system that aims at the detection of multi-query ransomware attacks in databases. In a nutshell, it represents an intrusion detection system that leverages knowledge about the attack pattern (or signature) and performs real-time system monitoring and pattern matching to detect intrusion attempts. For pattern matching, we leverage a CPN to encode the system states and their transitions inside the color information to detect when the system transitions to the state associated with the attack description.

The usage of PNs and CPNs is a known technique for pattern matching, and their application to intrusion detection problems was the topic in previous works [HP03; KS94]. However, typical application scenarios of CPN-based intrusion detection systems target other environments, e.g., networks [VH02] and operating systems [Axe00].

The application of PNs for intrusion detection in databases was only considered by Hu et al. [HP03]. They aimed at the detection of anomalies of any sort, not specific to ransomware. However, they use *uncolored* PNs and leverage them to model benign states of a database system rather than attack states. Hence, their solution requires a training phase to gain knowledge about the underlying data structure as well as about benign data update patterns. In contrast, our system does not require similar training. Moreover, their work is theoretical. Hence, they did not provide any implementation or evaluation results with which to compare.

In our work, we aim to fill the gap and address the problem of ransomware attacks targeting databases. As such, we investigate the applicability of CPNs for ransomware attack detection in databases. We observe that databases are complex systems and modeling their state regarding dependency relationships and update patterns, as, e.g., done in [HP03], may lead to overly complex system representations (for large and complex databases) and non-trivial overhead. Hence, we tackle the problem differently and choose to model malicious query sequences — an approach which results in a much simpler system representation, and independence from the structure of the underlying data and update patterns.

Our approach is system-centric and allows for the detection of attacks that span over multiple sessions or multiple user accounts. We also develop several novel techniques that even further to simplify the system representation, namely (i) dynamic color creation (creating an infinite color space), (ii) token merging and duplication, and (iii) token expiration making the use of CPN practical.

The remaining part of this section is structured as follows: We first describe a typical ransomware attack scenario (Section 8.1.1). Next, we present our adversary model (Section 8.1.2), followed by the system architecture description (Section 8.1.3). Finally, we show the interaction of the system components when handling incoming queries (Section 8.1.4).

8.1.1 Attack Scenario

Our attack scenario originates from an analysis of a large-scale ransomware attack targeting MySQL servers that took place in February 2017 [Ziv17]. The attacker performs the attack remotely by connecting to the database using a TCP connection.

Once connected, an attacker gains root access through, e.g., brute-forcing the 'root' password of the database. Next, they enumerate the data in the database through the retrieval of the list of the databases present. After that, the attacker creates a new table with an arbitrary name (e.g., the table with the name 'WARNING'), either in a new database (e.g., named 'PLEASE_READ') or in an already existing database. This table includes a ransom message containing a contact e-mail address as well as payment instructions to a bitcoin address. Finally, the attacker deletes (drops) the databases on the server and disconnects from it.

The scenario above describes the attack steps recorded in real-world attacks. Additionally, we accept that attack steps can deviate from this scenario: For instance, an attacker could first perform the database deletion and only after that insert the ransom message. Also, attackers may use arbitrary names for databases and tables and arbitrary patterns for the ransom message. We, however, assume that the attacker demands payments in cryptocurrency (such as Bitcoin or Ethereum) since they provide at least some level of anonymity in contrast to more traditional payment methods that involve banks[2]. We also assume that an attacker continues to wipe data and does not aim to keep any data copies, since this would slow down the attack significantly. Also, it would require storage on the attacker's side and a communication channel between the victim and the attacker, which demands additional resources and increases the chances of exposure. We also assume an attacker does not perform on-site database encryption since we did not identify any standard SQL commands that allow him to do so.

8.1.2 Adversary Model

We make the following assumptions about the goal and the capabilities of the attacker. The attacker's goal is to destroy the available data and claim the ransom. We assume the remote attacker who is accessing the server over the Internet has no physical access to it. The software running on the server is trusted, i.e., the attacker has no malicious software installed on the system. However, the attacker has full access to the network and can communicate with the Database Management System (DBMS) without any restrictions. Furthermore, we assume an attacker with administrator-level privileges to the DBMS. This assumption often becomes true in practice since the problem of weak or reused passwords [IWS04] is well known and not satisfactory solved for over decades.

For instance, findings show that most of the MySQL servers had no root password set due to using an insecure default configuration [Ora18]. Alterna-

[2]Since banks are obliged to follow "know your customer" policy.

tively, an attacker might exploit a security vulnerability like [Gol17] to gain administrator privileges for the database.

We, however, do not assume administrator privileges of the attacker to the operating system. Also, we leave DoS attacks are out of our attacker model since an attacker with administrator privileges to DBMS can always cause a denial of service, e.g., through the creation of fake databases or tables and exhausting the DBMS's memory. The attacker wants to perform a hit-and-run attack without considering other services and ways of communication.

8.1.3 System Architecture

Figure 8.1 shows the DIMAQS system architecture. DIMAQS comprises six components: (i) monitoring, (ii) classifier, (iii) security policy, (iv) incident resolution, (v) notifier, (vi) query rewriter, and (vii) controller. The monitoring and query rewriter components use the query parser embedded in the database server. Hence, the figure shows them as belonging to both, DIMAQS plugin and the database server. In the following, we describe the role of every component in more detail. **RQ 8.1b**

8.1.3.1 Monitoring

The monitoring component monitors all incoming queries for potentially malicious query sequences. Note that this module monitors all queries arriving through different connections, not specific to user sessions. Notifications on the occurrence of incoming queries result from the database server's audit functionality.

8.1.3.2 Classifier

The classifier component processes the incoming queries and produces a verdict whether a query is benign or malicious. For the classification, DIMAQS uses a CPN, to which we added multiple extensions. The token colors allow us to attach runtime information to the tokens, such as timestamps, table names, and modified cell values. Since such token colors are dynamic and unbounded, conventional PNs would be unable to represent all the possible states. This information also provides additional information to the DIMAQS administrator in the case of an incident[3].

[3]Note that DIMAQS administrator and database administrator are different entities

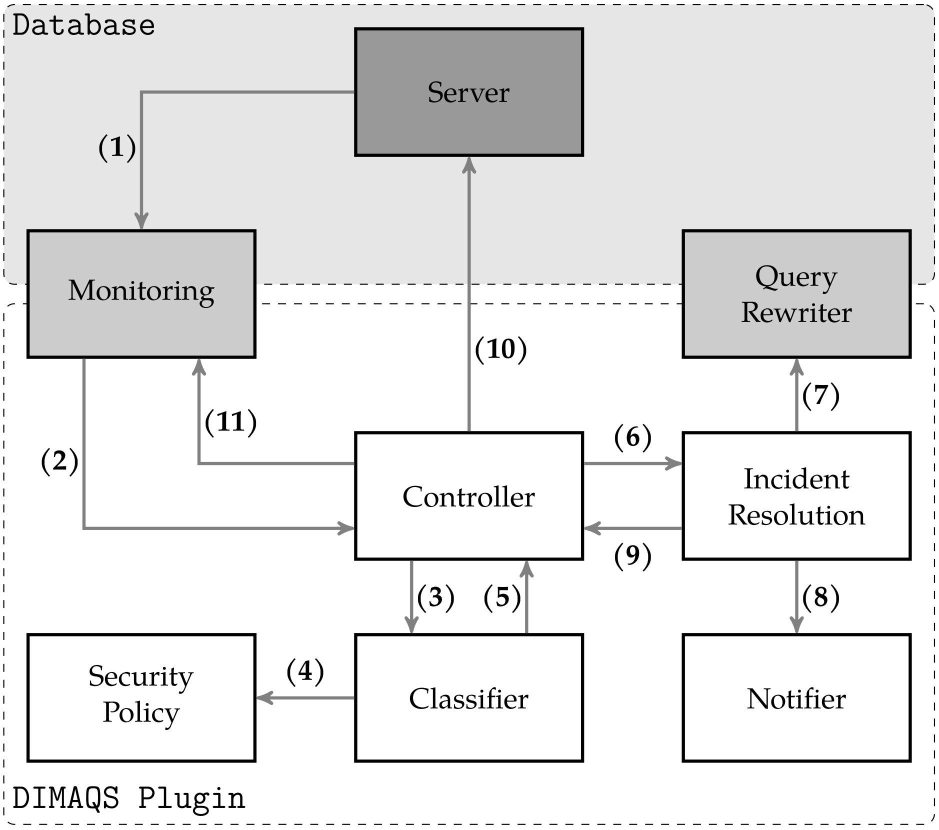


Figure 8.1: System architecture of DIMAQS. Dark grey boxes are components provided by the database, light grey boxes are components that interface between DIMAQS and the database, and white boxes belong to DIMAQS itself.

Extensions to CPNs

For our purposes, we extend CPNs with three new features:

The first is the dynamic creation of colors for storing information inside the tokens. This extension extends the space of possible token colors to infinity and allows us to encode any serialized information inside of a token.

The second extension is the ability to merge tokens that are identical except for their timestamps. This extension improves performance and does not impede classification accuracy.

The third extension allows for token expiration. Since each place in the

CPN can have timeout information, this feature allows us to limit the time window of analyzed query sequences. It is highly unlikely that a malicious query sequence spawns over a long period (e.g., multiple days), since this increases the risk of detection and complicates the attack (the database can change considerably over time). Abundant or absent timeouts can additionally result in a higher false-positive rate since, eventually, all transitions might be triggered by unrelated queries. The timeout threshold is, therefore, a security parameter, which enables a trade-off between effectiveness and false alerts. In real-world attacks observed so far, attackers did not stretch malicious query sequences over long periods. Hence, even short timeouts (1-2 minutes) would work well against them. Attackers might increase the attack time window to avoid detection. However, the longer they stay connected, the higher the burden for them (since the attacks are not generally automated), and the higher the risk of being uncovered.

8.1.3.3 Security Policy

RQ 8.1a

The security policy component holds information about patterns of malicious query sequences (or attack signatures). The CPN configuration represents it in our system — it describes the CPN's places, place actions, transitions, transition actions, transition conditions, and arcs.

All places and transitions are named, and the arcs are each weighted with a value of one token. Each place can be assigned several place actions executed upon CPN transitions to the corresponding place. Transitions are used to check for the execution of a (next) step in a malicious query sequence. They become active when every source place contains at least one token. Each transition is assigned one transition action, representing conditions for incoming queries. For instance, they may specify the query type (e.g., query that lists tables) and the actual content of the query (such as a table name or a typical ransom message).

A transition may also have an arbitrary number of transition conditions that are used to evaluate the token data from the source place against the query values. Our policy includes only one transition condition, ensuring ransom message insertion into a previously created or modified table.

We depict the CPN that was tailored to the observed attacks configured according to our security policy in Figure 8.2. Table 8.1 shows the place actions executed after putting a token on the place.

Transitions fire when an action occurs that is specified as malicious by the security policy component. Note, that no single action alone is enough to transit the CPN to the "attack detected" state. Typically, the sequence of actions would

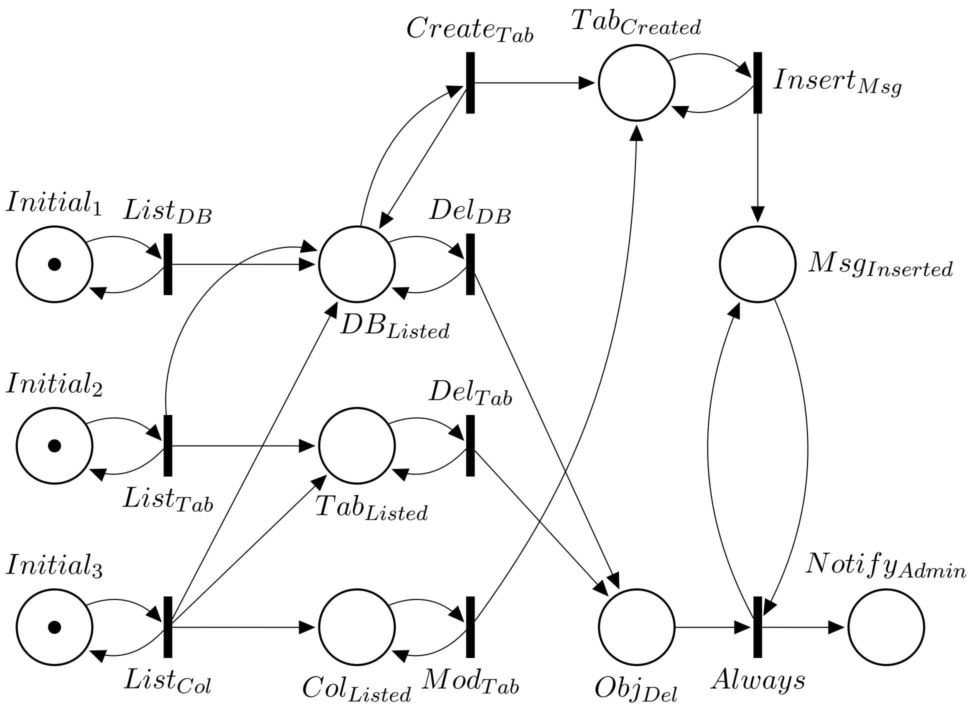


Figure 8.2: The CPN used to classify database transactions. All arcs are weighted with a value of one token.
States: $Initial_x$: initial states; $List_x$: objects listed, $Tab_{Created}$: table created; Obj_{Del}: object (database or table) deleted; $MSG_{Inserted}$: ransom message inserted; $Notify_Admin$: notification sent
Transitions: $List_{DB}$: list databases; $List_{Tab}$: list tables; $List_{Col}$ list columns; $Create_{Table}$: create table; $Drop_{Table}$: drop table; $Modify_{Table}$: modify table; $Insert_{Msg}$: insert ransom message

Place	Description
DB_{Listed}	Rewriting
Tab_{Listed}	Rewriting
Col_{Listed}	Rewriting
$Tab_{Created}$	Trigger creation
Obj_{Del}	Create backup
$Notify_{Admin}$	Create notification

Table 8.1: Configured actions for the places inside the CPN in Figure 8.2. When a token reaches a place, the specified action can be executed.

be required, and their execution requires a specific order (defined by the CPN configuration) to reach the state that corresponds to attack detection.

The policy is easily adaptable to include new attack signatures by modifying the CPN. While reconfiguration is a manual process, it is not cumbersome and can be accomplished in a reasonable amount of time[4].

8.1.3.4 Incident Resolution

When an event in the classifier component issues an action, an action must be carried out by the incident resolution module. Possible actions are "create backup," "rewriting," and "create notification." The incident resolution performs the rewriting of malicious queries as well as creates backups.

Create Backup Action

Whenever the system detects a potential attack, the incident component will move the database, or the table dropped by an attacker to a safe place instead of deleting it. The backup copy is invisible to users (and, hence, from the attacker). Thus, an attacker cannot drop it again or even identify that such a backup exists. The incident resolution uses a "rewriting" action to hide backed-up tables and databases from users. While performing such a move, the incident resolution renames the protected tables to avoid name collisions.

Rewriting Action

Rewriting actions rewrite queries to exclude tables and databases created by DIMAQS. The query rewriter component performs these actions.

Notification action

Notification actions are used by the incident resolution component whenever there is a need to notify an administrator about a detected attack. The Notifier component performs this notification, as described below.

8.1.3.5 Notifier

The notifier component informs about security incidents by sending an email to the DIMAQS administrator. The administrator information (e.g., email address) is defined in a configuration file. The gathered information relevant to

[4]Our estimate is 30 min.

the incident is attached to the notification so that the administrator can evaluate the incident and respond accordingly (e.g., restore the deleted table).

8.1.3.6 Query Rewriter

The query rewriter component rewrites queries to exclude tables and databases created by DIMAQS from query results. For a 'rewriting' action, the query rewriter receives the name of the table and, if applicable, the name of the database from the incident resolution component. If the queries are nested, the query rewriter extracts them into sub-queries, rewriting each sub-query separately. For instance, a query dropping a table will be rewritten to move the table to safe storage space. This operation happens without any indication to the attacker. Additionally, it rewrites statements that list tables and databases to exclude the hidden information from query results.

8.1.3.7 Controller

The controller component connects all other DIMAQS system segments. It is the central element that orchestrates the processing of incoming queries by other components, e.g., through the invocation of the classifier component to classify the query as malicious or benign, or the incident resolution component to initiate incident resolution upon attack detection.

8.1.4 Component Interaction

Figure 8.1 depicts the interaction between the components during query processing. The database server first receives the query and then notifies the monitoring module (1). If the monitoring module raises an alert for a potentially malicious query type, the controller is notified (2). The controller then forwards the suspicious query to the classifier (3) for evaluation. The classifier is configured using the security policy from the security policy (4) and returns the classification result to the controller (5). There are two possible outcomes: the query's classification is either benign or malicious. In the former case, the controller terminates its actions, and the server executes the query as-is (10). In the latter case, the query is considered malicious, and the controller calls the incident resolution (6), which in turn backs up dropped tables and rewrites the malicious query using the query rewriter (7). It then invokes the notifier to inform the administrator about an incident (8) and presents the supplied information. The controller then receives the rewritten "disarmed" query from the incident resolution (9). The database server then executes the query (10).

The controller informs the monitoring component when additional objects need observation (11), e.g., when a query creates new tables.

8.2 Implementation

DIMAQS's design is generic and applies to different database technologies. For the sake of illustration, we have chosen to prototype it for MySQL servers — our implementation is realized as a MySQL plugin compatible with MySQL server versions 5.7.x. To function, DIMAQS requires our own PN net implementation library libPetri as well as the mysqlservices library provided by the MySQL server. We chose the C++11 language for DIMAQS since it is the default language for MySQL plugins. DIMAQS consists of 4 908 Lines of Code (LoC), while libPetri results in 1 008 LoC.

8.2.1 Plugin Integration

The plugin is loaded during MySQL server startup and registers itself as an auditing plugin. The MySQL server plugin interface provides multiple notifications [Ora18] for the following useful events: **RQ 8.2**

- `MYSQL_AUDIT_CONNECTION_CLASS`,
- `MYSQL_AUDIT_CONNECTION_CONNECT`,
- `MYSQL_AUDIT_CONNECTION_DISCONNECT`,
- `MYSQL_AUDIT_PARSE_CLASS`, ''
- `MYSQL_AUDIT_PARSE_POSTPARSE`.

Notifications of the `MYSQL_AUDIT_PARSE_CLASS class` provide an event of a single to-be-executed query. Queries, however, could also be nested.

Per default, the MySQL server does not provide any event that returns the atomic values of database elements affected by `INSERT`, `UPDATE`, and `DELETE` queries. These queries are typical for the use in attacks like mimicry, e.g., for the insertion of ransom messages. To allow us to access the atomic values, we create triggers. We generate "`before INSERT/UPDATE`" triggers for every table. In these triggers, we execute a user-defined function. This function forwards the values affected by the queries to the controller for evaluation.

As detailed in the MySQL trigger syntax [Ora18], a trigger becomes associated with a table named `tbl_name`. This name must refer to a permanent table, which means that a trigger does not apply to a temporary table or a view. This limitation does not affect our solution since it is unlikely that an attacker would attack data stored in temporary tables.

8.2.2 Component Integration

In the following, we detail the implementation of DIMAQS' modules.

Monitoring

Additional triggers are required to access information that is not transparent to the DIMAQS plugin when using MySQL's audit features. The trigger creation occurs when loading the plugin, and existing triggers are recreated after server startup since the database structure might have changed. Trigger creation within so-called "stored procedures" or "stored functions," the conventional concepts supported by the MySQL server is not possible. Due to this limitation, the creation must be within the plugin code. The function `dimaqs_plugin_init()` performs the creation of the additional triggers and runs directly after the initialization of the server and before entering the listening state. `dimaqs_plugin_init()` creates a trigger for every non-virtual database. Virtual databases are databases that contain read-only views rather than base tables and have no database files associated with them. Hence, the protection of virtual databases is not necessary.

The `INSERT` and `UPDATE` triggers call `eval_value()`. Several values are passed to that function, namely (i) schema name, (ii) table name, and (iii) new column values. Using this structure, we can identify inserted/updated values.

Classifier

The classifier is implemented using our library libPetri. libPetri is a C++ library implementing the functionality of colored Petri nets. It includes dynamic coloring, token timeout, and token merging features mentioned in Section 8.1.3.2. Since libPetri has been developed explicitly for DIMAQS, it carries no additional feature overhead. Thus, libPetri contains all necessary functionality within 1 008 of LoC.

libPetri keeps track of all active transitions. Since all our arcs in the classifier are weighted with the value one as seen in Figure 8.2, active transitions have tokens on all input places. If the to-be-classified query matches the action attributed to an active transition that transition fires. When transferring a token to a place with an associated action, that action executes with the corresponding parameters. Until completion of these actions, the classifier does not accept additional queries.

Security Policy

The security policy is a database that contains tables holding the information about the actions that can fire transitions (e.g., the regular expression for detecting the ransom message) and the places with their associated actions. The classifier processes this information on startup and during classification.

Incident Resolution

The incident resolution backs up dropped databases and deleted values. The renaming of databases is not trivial due to MySQL limitations. MySQL added a command to carry out a database renaming called '`RENAME DATABASE <database_name>`.' However, this command was only active through a few minor releases before its discontinuation. The simplest way to rename a database is to move its tables to another database. Each moved table requires the recreation of the affected triggers. Table renaming follows the following schema: "`<storagespace>.<object prefix>_<dbname>_-<tablename>_<timestamp>`" where `storagespace` being a preconfigured variable of DIMAQS. The function `renameTable()` performs this renaming. If a database drop occurs, `renameDatabase()` calls the `renameTable()` for every table.

For backup actions, a 'DROP DATABASE <db_name>' does not require rewriting. However, before executing, *renameTable* or *renameDatabase* is executed to back up the database tables.

Notifier

The notifier sends an email with all transmitted information about the suspected attack to the administrator. The administrator's address can be configured inside the database or in a configuration file.

Query Rewriter

The rewriter component rewrites a query by adding a WHERE/AND condition to hide sensitive information or rewrites it entirely, e.g., for backup operations. Table 8.2 lists the commands that require rewriting.

Command	WHERE | AND
SHOW PLUGINS	no rewriting possible
SHOW DATABASES	‘Database‘ NOT LIKE ’dimaqs%’
SHOW TABLES	rewriting not needed
SHOW TRIGGERS	‘Trigger‘ NOT LIKE ’dimaqs%’
SHOW COLUMNS	rewriting not needed
SHOW VARIABLES	‘Variable_name‘ NOT LIKE ’dimaqs%’
SELECT FROM information_schema.columns	SCHEMA_NAME NOT LIKE ’dimaqs%’
SELECT FROM information_schema.files	FILE_NAME NOT LIKE ’./dimaqs%’
SELECT FROM information_schema.key_column_usage	TABLE_SCHEMA NOT LIKE ’dimaqs%’
SELECT FROM information_schema.partitions	TABLE_SCHEMA NOT LIKE ’dimaqs%’
SELECT FROM information_schema.schemata	SCHEMA_NAME NOT LIKE ’dimaqs%’
SELECT FROM information_schema.tables	SCHEMA_NAME NOT LIKE ’dimaqs%’
SELECT FROM mysql.db	Db NOT LIKE ’dimaqs%’
SELECT FROM performance_schema.file_instances	FILE_NAME NOT LIKE ’%/dimaqs%/’
SELECT FROM performance_schema.objects_summary_global_by_type	OBJECT_SCHEMA NOT LIKE ’dimaqs%’
SELECT FROM performance_schema.table_handles	OBJECT_SCHEMA NOT LIKE ’dimaqs%’
SELECT FROM performance_schema.table_io_waits_summary_by_index_usage	OBJECT_SCHEMA NOT LIKE ’dimaqs%’
SELECT FROM performance_schema.table_io_waits_summary_by_table	OBJECT_SCHEMA NOT LIKE ’dimaqs%’
SELECT FROM performance_schema.table_lock_waits_summary_by_table	OBJECT_SCHEMA NOT LIKE ’dimaqs%’

Table 8.2: Rewriting WHERE/AND performed by DIMAQS.

Controller

The controller is implemented using the *visitor* design pattern. This visitor extracts the nested statements from inside to outside. It then forwards each extracted query to the classifier.

8.3 Evaluation

In this section, we describe our test setup and evaluate our implementation with regards to effectiveness and performance. We conclude by discussing security considerations. **RQ 8.3**

8.3.1 Test Setup

For the evaluation, we first define the used testbed and data sets.

8.3.1.1 Testbed

Unit	Value
Product	HP ProLiant DL360 Gen9
CPU	Intel Xeon E5-2640 v3
Default CPU frequency	2.60 GHz
Max CPU frequency	3.40 GHz
Min CPU frequency	1.20 GHz
Cores (Threads)	8 (16)
Cache (L1/L2/L3)	512 KB/2048 KB/20480 KB
Memory size	32GB (2 x 16 GB) DDR4 Dual Channel
Memory frequency	1.866 GHz
Memory Connection	Dual Channel
Storage Model	HP VK0800GEFJK 800 GB SSD
Storage Connection	SATA III (6GBit/s)
Operating System	Ubuntu 16.04.4 LTS (x86-64)
Kernel	4.4.0-121

Table 8.3: The server used for evaluating DIMAQS.

To execute performance and security tests, we use the following setup. For the database server, we use an HPE ProLiant DL360 Gen9 server. The server is equipped with a single 8-core Haswell generation Xeon E5-2640 CPU with a base clock of 2.60 GHz and a turbo clock of 3,40 GHz and packaged with a total of 20 MB of cache. Simultaneous multithreading is enabled, allowing the execution of 16 threads in parallel. Table 8.3 gives additional information. To provide a DBMS to evaluate against, we install and run MySQL server 5.7.22 on this server.

All tests are executed directly on this server. Thus, the network is not a limiting factor for the benchmarks. Due to the performance of the server, we expect the resources consumed by the client running in parallel to the server to be negligible. Their performance influence is, therefore, not evaluated in this work.

8.3.1.2 Data Sets

We employ three data sets during our evaluation. The first set (malicious set) includes malicious query sequences, which we generated ourselves using information about real-world attacks collected at [Ziv17]. Our resulting query set contains query sequence permutations with an expected malicious classification, as well as their possible permutations (since an attacker may execute them in arbitrary order). The full test set contains 13 485 tests. Each test contains nine queries. The first five queries of each test are to set up two databases and a table at the beginning of the experiment and remove them at the end. Relevant to the detection are four queries: (i) listing all databases, (ii) creating a table, (iii) inserting a ransom message into this table, and (iv) dropping a table or database. Therefore, the set performs 53 940 queries in total.

The second set (*Bibspace set*) is from the publication management system *Bibspace* [Ryg18], which was gathered over 40 days from the 13th of April 2018 to the 22nd of May 2018 and contained a total of 52 085 queries. Among them, 24 430 are `CREATE_TABLE_IF_NOT_EXISTS` queries, 8 357 `INSERT` queries, and 38 `DROP_TABLE_IF_EXISTS` queries.

The third query set (*MediaWiki set*) is from a locally run *MediaWiki* [Med18] with the *Semantic MediaWiki* [sem18] plugin enabled, collected for 50 days from the 3rd of April 2018 to the 22nd of May 2018. Containing 2 514 764 queries, it includes 69 261 `INSERT` statements, 29 830 `CREATE_TEMPORARY_TABLE` statements, and 29 797 `DROP_TEMPORARY_TABLE` statements.

8.3.2 Effectiveness

In the following, we evaluate the precision of the classifier module. Thus, we evaluate whether a wrongful classification of benign queries as malicious (false positives) or malicious query sequences as benign (false negatives) occurs.

8.3.2.1 Security Policy

The execution policy for the classifier is as described in Section 8.1.3.2. Our policy is quite generic in the sense that we do not look for specific table or database names, but instead detect the removal or renaming of any table or database. However, we are looking for a specific pattern of the ransom message. We search for the occurrence of a BTC or Bitcoin string inside the inserted message since attackers until now requested ransom in Bitcoins[5]. We used the regular expression `'(\d*[.]){0,1}\d+\s*(BTC|Bitcoin)'` (case insensitive). The matching expressions are, e.g., 5 BTC|Bitcoin, .5 BTC|Bitcoin, 20.1 btc|Bitcoin.

8.3.2.2 False Negatives

To test for false negatives, we used the *attack set* described in Section 8.3.1.2. After processing all the queries from the data set by our CPN, we achieved a 100% attack detection rate and received no false-negative result. This result confirms that our CPN correctly models each attack from our malicious data set.

8.3.2.3 False Positives

To test for false positives, we choose to use the *Bibspace set* and the *MediaWiki set*. The sets contain a total of 2 566 849 benign queries. The classifier performs the classification of every set. Afterward, the classifier state shows, if DIMAQS wrongfully detected attacks and how many false detections occurred. If tokens reach place N in the classifier, their number represents raised alerts. For this evaluation, we disable the token timeout to increase the potential for false positives.

Table 8.4 shows the population of the CPN after running all the queries from the *Bibspace set* through the classifier. No token has reached the state N, which would have triggered an alert to the administrator. Next, the classifier processed the queries of the *MediaWiki set*. Table 8.4 also shows the state of the CPN after classification. Again, no token has reached the state N, and no ransom attack was detected, which is a favorable result.

[5]Our policy can be trivially extended to detect ransom messages requesting payments in other cryptocurrencies.

Position	Query Set	
	Bibspace	MediaWiki
$Initial_1$	1	1
$Initial_2$	1	1
$Initial_3$	1	1
DB_{Listed}	2	7
Tab_{Listed}	2	5
Col_{Listed}	0	1
$Tab_{Created}$	24	0
$Object_{Deleted}$	0	0
$Notify_{Admin}$	0	0

Table 8.4: CPN state after execution of query sets.

8.3.3 Performance Evaluation

To evaluate the performance of the DIMAQS plugin, we used two data sets: The *MediaWiki set* described in Section 8.3.1.2 and the synthetic benchmark sysbench [Kop18]. We use sysbench 0.4.12 with 16 active threads. We performed three performance benchmarks: (i) without the plugin as a baseline measure, (ii) operating on a newly initialized PN, and (iii) with a fully occupied PN initialized with tokens in each state. The sysbench benchmarks were run for 60 seconds per iteration, while the *MediaWiki set* was classified entirely every time. We performed each benchmark for over 50 iterations. Table 8.5 shows the resulting measurements (database transactions per second). We report average values with standard deviation and confidence intervals (5% quantile calculated using an inversed Student's t-distribution). Figure 8.3 visualizes these results.

The results show that the usage of the DIMAQS plugin results in performance degradation of about 5 % for sysbench. There is no substantial difference whether the CPN is only initialized or entirely populated (overlapping confidence intervals). This marginal difference suggests that the overhead is not a result of querying the CPN, but from analyzing and parsing the queries themselves. For the *MediaWiki set*, performance degradation is about 2% for the initialized CPN and 4% for an entirely populated net. This time, the influence of the set population has a more significant impact.

Our proof-of-concept prototype is not yet optimized for performance. Neither DIMAQS nor libPetri has received extensive profiling for potential bottlenecks.

Test	Transactions per second			relative to baseline [%]		
	mean	stdev	conf. int	mean	stdev	conf. int.
sysbench						
disabled	9 245	28	±9	100.0	0.3	±0.1
initialized	8 806	30	±11	95.3	0.3	±0.1
full	8 823	19	±7	95.4	0.2	±0.0
MediaWiki						
disabled	2 008	5	±2	100	0.2	±0.1
initialize	1 971	7	±2	98.2	0.3	±0.1
full	1 930	6	±13	96.1	2.9	±0.3

Table 8.5: Performance without the plugin (disabled), with the enabled plugin (initialized), and with tokens in each PN state (full).

Also, no compiler optimizations were enabled. Thus, performance improvements are likely possible.

8.3.4 Security Considerations

In the following, we discuss two potential attack scenarios against DIMAQS itself and show how our system defends itself against them.

Disabling DIMAQS

An attacker may try to disable DIMAQS to avoid detection. However, such a scenario would not be successful since administrative privileges to the database are insufficient to perform this task. One would need to have administrative privileges to the file system to manipulate corresponding config files. As an additional burden, it is also non-trivial for an attacker to detect that the system runs under DIMAQS observation because the query rewriter component of DIMAQS rewrites the queries in such a way that it excludes information about DIMAQS from the results.

DIMAQS Trigger Removal

Another possible attack vector is specific to the MySQL implementation, which uses triggers. An attacker may attempt to delete triggers, which are used to deliver additional information to the DIMAQS plugin.

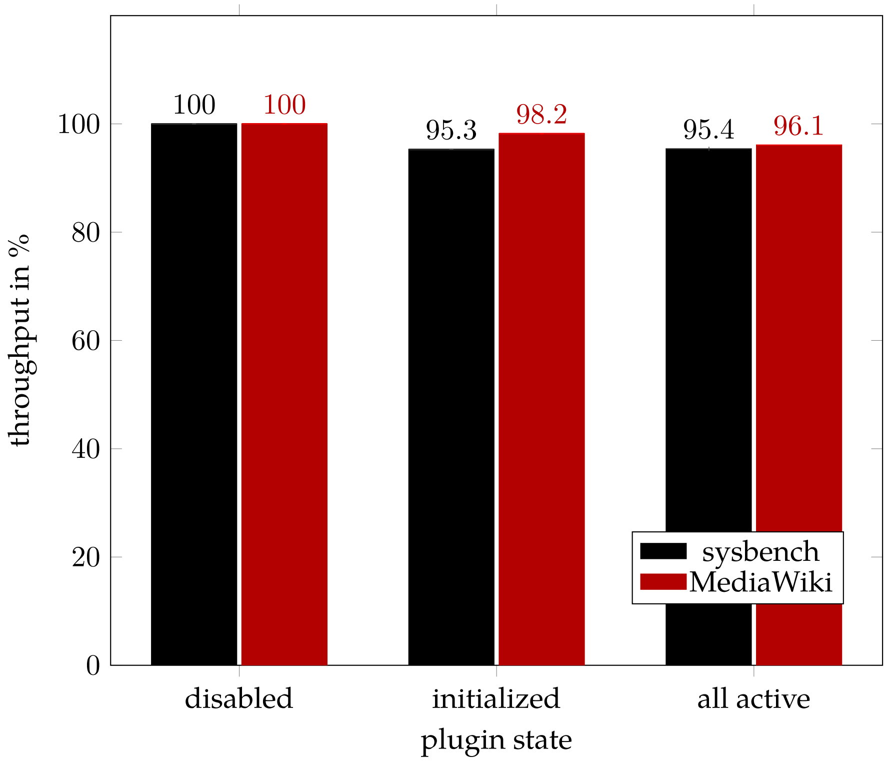


Figure 8.3: Performance influence of DIMAQS for sysbench and MediaWiki. Values are normalized to the respective value for the disabled plugin.

To defend against this attack vector, DIMAQS detects the removal of DIMAQS-specific triggers. Their absence becomes obvious whenever the plugin does not receive information about atomic values affected by the queries. Upon detection, DIMAQS generates a notification for the DIMAQS administrator and backups all the databases and tables affected by subsequent queries.

8.4 Summary and Evaluation of Research Questions

In this chapter, we presented a database IDPS against ransomware attacks consisting of multiple queries. We implemented this solution for MySQL servers and evaluated the resulting prototype regarding performance and security.

RQ 8.1a How to model multi-query database ransomware attacks?

We found that Colored Petri Nets (CPNs) are a suitable formalism to model database ransomware that uses multiple queries. Regular Petri Nets (PNs) are sufficient to model the attacks themselves to detect if an attack occurs. We were able to encode information regarding the course of the attack, by leveraging the token colors from CPNs as an information carrier. We use places inside the CPN to model the state of the system and the progress of the attacker. Transactions model every step that brings an attacker closer to his goal.

RQ 8.1b Which components does a multi-query database IDPS require and how do they interact?

Our solution Dynamic Identification of Malicious Query Sequences (DIMAQS) consists of multiple components. We require a controller, a classifier with a security policy — which includes the aforementioned CPN and an incident resolution that uses a notifier and a query rewriter. An additional monitoring component informs the system about incoming queries ready for processing. While the monitoring component and the query rewriter are MySQL specific, the remaining components are generalized.

RQ 8.2 How to integrate a prototype multi-query database IDPS into a MySQL server?

The plugin integrates by registering as an auditing plugin upon the MySQL server's star-up for multiple events. To access additional atomic information that is not provided by the plugin interface, DIMAQS uses "`before INSERT/UPDATE`" triggers for every table. When a query adds/removes tables, DIMAQS adapts its trigger sets.

RQ 8.3 How does the multi-query database IDPS perform in terms of security and performance?

We tested DIMAQS with two benign production datasets and a malicious dataset constructed from all permutation of known MySQL ransom attacks. DIMAQS processed the malicious set without any false negatives and benign sets without false positives. We then evaluated DIMAQS' performance using a benign dataset and a synthetic benchmark. The results show that DIMAQS only has a small performance impact of — in the worst case — below 5%. We also discussed potential security issues and how we hardened DIMAQS against these vulnerabilities. Thus, DIMAQS is a ready-to-use plugin with only a small performance impact that reliably performs its task as a database IDPS.

Chapter 9

Conclusion

In this chapter, we conclude this thesis and provide a summary of its contributions. It also discusses the benefits of our work and gives an overview of the potential for future work.

9.1 Summary

This thesis addresses two main issues: (i) *Many security functions do not interact well with the surrounding infrastructure. This absence of integration leads to shortcomings regarding performance and flexibility.* We address this issue by developing three SDN-based algorithms that bypass IDPS for packets that are not relevant to the IDPS, or that can not be malicious at the current state of attack. Furthermore, we design a SYN flood DPS using SDN and NFV that increases performance compared to existing solutions and eliminates existing statefulness and scaling issues. (ii) *SSFCs design is mainly static, and SSFCs do not adapt to the surrounding circumstances and the incoming traffic mix.* For this issue, we first present the general idea of attack-aware SSFC reordering and then perform performance measurements and modeling for a better understanding of SSFC behavior. With this knowledge, we then create a framework to realize our idea. Besides the two main issues, we also tackle two additional problems: (iii) *Cloud architectures rarely use the performance potential from short term CPU frequency boosting.* We design an approach for heat-aware load balancing that aims at keeping active CPUs to address this issue. (iv) *Ransomware attacks — attacks that erase or encrypt data and blackmail the user into paying for the restoration — against databases become more complex and contain a series of queries. So far, no matching signature-based security solutions exist.* We address this final issue by designing a signature-based database IDPS that can detect multi-query database ransomware. Specifically, the primary contributions presented in this thesis are:

Contribution 1: Dynamic Network Intrusion Detection System Bypassing

This thesis presents an SDN-based approach to bypass network IDPS for packets that are unlikely to contain attacks that the IDPS could detect. We, therefore, propose two dynamic SDN-based algorithms (one blacklisting and one whitelisting approach) and, for comparison, a solution using static SDN flows for selective filtering. The blacklisting algorithm and the selective filtering approach require knowledge about the IDPS and its rules, while the whitelisting approach does not require yet benefits from this knowledge.

We present an evaluation environment to assess our approaches and test them for constant load and in an overload scenario as well as when using hardware and software switches. All approaches increase the performance compared to a non-augmented inline IDPS while keeping excellent security metrics. Notably, no false-negative classifications occur. Thus, no malicious packets reach their destination. The static approach shows an approximately constant performance when using the hardware switch and the software switch. Both dynamic approaches show significantly better performance when using the software switch. Then, the dynamic approaches achieve the same throughput as a system without an IDPS for a constant load (around 13 times the performance of the inline IDPS) and 88.6% of it for overload.

In summary, the whitelisting performs best of all approaches in combination with the software switch. There, it matches the requirements of significantly improved performance, all security metrics above 95%, and less than 10% of all attacks lost in the network and not classified. Thus, the solution is working, yields a significant improvement, and is deployable without knowledge about the IDPS configuration.

Contribution 2: TCP Handshake Remote Establishment and Dynamic Rerouting using Software-defined Networking (THREADS)

Next, we develop THREADS, a novel DPS to defend against TCP SYN flood attacks. SYN flood DPSs take over the TCP handshake to protect services. Unlike previous solutions that lack in at least one category, THREADS is stateless, independently deployable, scalable, and does not require packets to run via it after connection establishment. We implement our approach as a VNF using DPDK. This VNF handles connection establishment and creates if successful SDN rules to forward the packets to the server directly. On the server-side, THREADS requires a modified Linux kernel.

Next, we evaluate THREADS in a dedicated evaluation environment. We show that THREADS correctly handles connection establishment. However, the network often loses the first packets after establishing a connection because they still reach the VNF instead of the server. We present three solutions to this issue, depending on whether it is acceptable that the server may receive packets for which no connection establishment succeeded or whether THREADS should be always-on or just in case of an attack. Then, we compare THREADS performance against the existing popular solutions SYN Cookies and SYNPROXY showing a performance increase by 208% without any optimizations.

Still, we aim at further increasing the performance and perform parallelization and parameter tuning. Therefore, we develop two parallelization strategies: (i) a locked access to the NIC's send and receive queues, and (ii) a ringbuffer. For these strategies, we specify four (locked access), respectively, five (ringbuffer) parameters to tune. We find that the performance of the locking approach with optimal parameters is best and yields another 76% of performance over the single-threaded non-optimized version. In summary, the performance of THREADS is 5.4 times as high as for existing solutions and is close to max out the theoretical limit of a 10 Gb/s link while running on older server hardware. The results also show that THREADS scales well for a small number of cores but does not profit from an extreme core count. Therefore, we suggest horizontal scaling using small to medium instances.

Contribution 3: Performance Engineering and Modeling for Security Service Function Chain Orders

Next, we introduce the idea of attack-aware SSFC reordering. This idea follows the vision to always position the security functions inside an SSFC in the optimal order for the current attack composition. Security functions have the added complexity that unlike other network functions, they also drop traffic categorized as malicious. Thus, traffic dropped at the SSFC's beginning does not reach the security functions further along the chain and, also, does not create resource demands for them. While for a single attack, the simple answer would be to just place the matching security function first, for more complicated composite attacks that require multiple security functions that have different performance characteristics, the choice is not as simple. We, therefore, propose to use a model for the SSFC to optimize its order. An instance called the FCC collects attack statistics from the security functions and decides the optimal order based on this model.

We perform measurements to prove our claim that the ordering is relevant during attacks. First, we 'show for an example firewall, IDPS, and DPS that they have significantly different performance characteristics even for benign traffic, which mainly appear in the throughput they can handle. Under the specified attacks, we show that the performance of the service increases when using systems that protect against DoS attacks while securing against software vulnerabilities even decreases performance. Furthermore, protecting against attacks that are irrelevant to the service can also hamper the overall performance. When putting the security functions in SSFCs of size two, we find that putting the defending function first always yields better throughput and, in most cases, also better latency and drop rate. This advantage can be up to three orders of magnitude.

Furthermore, better orders also reduce the total CPU load on the employed servers. At this point, we conclude that the order inside an SSFC is essential to its performance. Also, optimal orders for different attacks contradict each other, which makes a case to adapt the order always to have optimal performance dynamically.

Next, we develop a modeling formalism for SSFCs. To this end, we first model single security functions. A security function changes the composition of the incoming traffic. Thus, we categorize the traffic into multiple traffic classes comprising one class per attack type and another class for benign traffic and unmatched attacks. We then describe the effect of a security function as a function on these traffic classes. Following the composition, the resource demand also results from the traffic classes. Each packet, depending on the class, creates a fixed resource demand per frame, packet, or segment. Also, for some security functions, the resource demand depends on the size of the data unit. Multiple of these security function models form an Security Function Chaining Controller (SFCC) model where the output traffic from one function is the input traffic of the next function — until only the benign or unmatched packets remain — and the total resource demand results from the sum of all single functions. Knowledge about the traffic volume reaching the server and the number of attacks at each step of the chain allows computing the original incoming traffic's composition. Based on this composition and the models, the FCC can compute the desired SSFC order. We provide three approaches for order adaptation with complexity between factorial and linear differing in their guarantee of an optimal order as well as full and partial reordering.

Contribution 4: A Framework for Attack-aware Security Service Function Chain Reordering

Having laid out and validated the idea of attack-aware SSFC reordering before, we create a framework that realizes this idea using SDN. All security functions of an SSFC reside inside an SDN-enabled network. A *security function wrapper* co-located with every security function reports attacks to the FCC. Then, the FCC computes the desired order for the security functions and executes it via the SDN controller. We propose an authentication and communication interface that enables this interaction and provide a Proof-of-Concept (PoC) implementation. For this PoCs implementation, we also provide a minimal SDN controller.

We test our PoC inside a testbed. First, we prove the correct ordering by permuting through all possible orders and validate the correct route via the security functions. Then, putting the framework under simulated attacks, we show that the framework correctly adapts to all attacks as desired and — after the attacks subside — it returns to the initial state. Three issues can occur during reordering: (i) packets drop, (ii) packets pass through a function twice, or (iii) packets skip a function. While the first two issues are undesirable yet without security impact, the third issue could result in malicious packets reaching the service. We propose four different solutions depending on the use-case to alleviate these issues.

Contribution 5: Heat-aware and CPU Boost-oriented Server Load Rotation

Applications often make little use of short term CPU frequency boost technologies like Intel's TurboBoost. We present a solution to heat-aware load balancing, allowing to alleviate this issue and maximize the time, active CPUs spend in the state of a short-term frequency boost. Therefore, a load balancer must detect or predict the moment when a server is too hot to stay boosted and migrate the running service to another server. We present and implement a solution consisting of two components: (i) a monitoring component watching the states of all workers, and (ii) an SDN controller that creates flows based on those observations. This solution works in a way that multiple servers appear as a single server to the client by modifying IP and MAC addresses in the packet headers.

We evaluate our implementation in an evaluation environment. We show the general functionality of the approach and analyze the impact of our solution, showing that it performs best when the whole cluster is at lower load levels. Still, it increases the time spent in a boosted state even for higher load scenarios. For lower load levels, it is always possible to keep the active server boosted without significant temperature increases. Performance-wise we show that in high-load scenarios, we can increase performance per additional Watt (relative to idle mode) by 35% while reducing the average temperature of the cluster and the maximum temperature of each server. With a lower load, all performance metrics improve, the temperature levels drop below 60 degrees, and the total power consumption decreases, thereby increasing performance per additional Watt by 41%. We attribute this behavior partially to the reduced fan activity.

Contribution 6: Signature-based Database Ransomware Detection

In the final contribution, we propose a new multi-component signature-based Intrusion Detection and Prevention System (IDPS) named DIMAQS for relational databases capable of protecting against newer attacks that no longer consist of a single query but instead query sequences. It contains the following components: (i) controller, (ii) monitoring, (iii) security policy, (iv) classifier, (v) incident resolution, (vi) notifier, and (vii) query rewriter. We use a signature comprised of a CPN to model the complex attack behavior. This CPN models the stages of the attack as places and the queries as transactions. A specialty of our solution is that it creates backups to roll back malicious actions and hides them and its existence by rewriting incoming queries.

We create our prototype DIMAQS for MySQL registering as an auditing plugin with the MySQL server enhanced by multiple triggers for the protected tables. Then, we evaluate the results for benign and malicious query sets without any misclassifications. Next, we analyze DIMAQS's performance showing that for a benign data set and synthetic benchmarks, it has a performance impact of below 5%. Last, we discuss potential security issues and explain how we hardened DIMAQS against these vulnerabilities.

9.2 Benefits

The work in this thesis benefits multiple groups of people. Among others, data-center and service operators, software, and hardware developers benefit from the contributions introduced in this thesis. We see the following significant benefits of our work:

- Our dynamic bypassing approach is of benefit for data-center operators, security software developers, and researchers. Data-center operators are the primary target group of our solution and can directly employ it to increase IDPS performance. Thereby, they can reduce costs or improve service quality. Security software developers can use our approach as an inspiration for further bypassing approaches, and security researchers gain a new solution against which to compare their approaches.

- Our THREADS DPS also is of benefit to the same groups. Cloud providers can use it due to its direct applicability to increase the flexibility and scalability of their DPS infrastructure. Security software developers receive an inspiration to incorporate into their approaches, and security researchers receive a new tool against which to test and to develop attacks. Last, we hope to contribute to the open-source community by making the SYN+ packets from THREADS an adopted standard.
- The performance engineering and model for SSFCs and the framework for attack-aware SSFC reordering lay the groundwork for further research in this field. The current solution already can augment existing data-center structures and is available to cloud providers. In general, the concept of attack-awareness can change how security infrastructures work and significantly reduce costs while at the same time increasing the service quality.
- Our approach to better utilize short term CPU frequency boosting is ready-to-use for service providers. It can increase performance while reducing temperature levels and, in some scenarios, power consumption. Also, for hardware developers, we create a new use case for which to optimize their boost technologies. Last, we hope to motivate researchers in the area of performance to no longer see frequency boosting as an annoying nuisance to disable for more stable results but as an opportunity for better performance to exploit.
- Last, DIMAQS is a fully functional signature-based database IDPS usable by every service provider. It allows for an efficient way to protect against known ransomware attacks without requiring complex models of the desired behavior or complex runtime machine learning.

9.3 Future Work

Our contributions in this thesis lay the ground for future work. We see several potential directions to follow and challenges to address in the future:

Integration of SDN-enabled Security in Data Center Networks

In this work, we showed the impact of SDN-augmented single security functions and the value of reordering SSFCs using SDN. While these contributions have a clear benefit in themselves, their contribution within each other remains a complex challenge. Systems like ONOS [Ber+14] and OpenDaylight [Med+14] contribute to the simple deployment of NFV and SDN. Still, they fail to sophisticated complex SDN strategies without them having to know about each other and explicitly interact. Here, we see the potential for further research with the goal of a declarative solution in mind, that allows independent development of SSFC management tools and SDN-enabled stand-alone security functions.

Traffic-dependent SSFCs

We developed a solution for attack-aware SSFC reordering and already defined traffic classes for different attacks. We propose to develop a more fine-grained classification approach regarding traffic type (e.g., protocol) but also other factors like regional origin using GeoIP. Then, our system can determine an optimal order for groups of those types or each type. Depending on the resources, we propose to either employ a dedicated SSFC for every traffic class group or to use the same security functions for all classes and enforce different orders with class-dependent SDN flows at every step of the chain. We expect that this distinction can further improve the system's overall performance, especially when defending against composite attacks that attack the system on multiple vectors.

Automated Model-learning for Security Functions

We use manually developed models to model security functions and SSFCs. While this is acceptable for scientific works and small-scale infrastructures, for more extensive infrastructures, operators do often have neither the time nor the knowledge to model all possible attacks. We, therefore, propose an automated approach that — at times where the system operates under benign conditions — sends attacks through the network or via single functions and learns (e.g., by regression learning) the effect of the functions on the traffic composition and the generated resource demands. The system should regularly repeat these steps until reaching the desired accuracy. In a second step, the system should detect when security functions change (e.g., when upgrading a function's software stack or when adding new signatures to an IDPS) and validate that the previously created models still fit.

Consideration of Energy-Efficiency in SSFCs

So far, we mainly considered the throughput as the primary metric for SSFC performance while also discussing latency and packet loss. We also took a brief look at the impact on the CPU load. While reducing the resource demands and the number of required security function instances is a generally reasonable guide-line, we also found that sometimes load balancing across multiple servers can reduce power consumption. Thus, we propose to perform energy-efficiency research on SSFCs and incorporate the results into our security function models and as a factor for deriving the optimal SSFC order. We also predict that having power models and results that show an efficiency impact of attack-aware SSFC reordering would further motivate data-center operators to adopt this approach and researchers to contribute to it.

Introduction of CPU Frequency Boosting in Performance Models, Placement, Scaling and Deployment Strategies

We showed the potential that lies in the usage of short-term CPU frequency boosting. So far, performance modeling researchers preferred to turn off this feature to attain a higher model accuracy. While from a research point of view, this approach is valid, it hampers the real-world application of these models. Therefore, we strongly urge researchers in this area to further work on accurate modeling of the boosting process. Similarly, we suggest augmenting placement and deployment strategies with boost-awareness. As we showed, especially in low-load scenarios, hot spares that are part of rotations can even reduce power consumption. Thus deployment and scaling strategies should include this effect before scaling up or down. Since the location of the boosted server is relevant (e.g., the temperature of neighboring servers, cooling infrastructure, or various heat capacity), placement strategies should also consider these factors when deciding on which servers to place a service.

Machine-assisted Learning for Database IDPS Signatures

Our database IDPS that protects against multi-query ransomware attacks requires signatures to detect attacks. So far, we created these signatures manually. We propose a machine-assisted learning approach where a database operator or a security researcher can mark a set of queries that constitute an attack. Then, the algorithm would derive a CPN (or an alternative formalism that best matches the attack) from the branded queries. The more sets of attacks are available to the algorithm, the better it can refine the model and reduce false positives and false negatives. This learning would us allow to generalize our approach further and adapt it to further attacks.

Appendix A

Additional Security Function Configuration for the Evaluation Environment in Section 5.2.1

A.1 Additional Snort Rules

```
drop udp $EXTERNAL_NET any -> $EXTERNAL_NET 31335
(msg:"Unexpected pen is detected"; content:"HELLO";
classtype:attempted-dos; sid:232; rev:5;)

(msg:"Lennart Poettering detected"; content:"systemd";
classtype:attempted-dos; sid:232; rev:5;)
```

Listing A.1: Additional signatures for the IDPS used in Section 5.2.1 [Fel18].

A.2 Firewall Rules

```
#!/bin/bash

# iptables script
# author Christina Hempfling
# date 2017-09-28

IPTABLES="/sbin/iptables"

# ------------------------ CHAINS ------------------------ #
# all packets to chain VALID are logged  and accepted
$IPTABLES -N VALID
$IPTABLES -I VALID -j LOG --log-prefix "NewConnection "
$IPTABLES -A VALID -j ACCEPT

# all packets in the following chains are logged and dropped
# table for spoofed IP addresses
$IPTABLES -N SPOOF
$IPTABLES -I SPOOF -j LOG --log-prefix "ATTACKIPSPOOFING "
$IPTABLES -A SPOOF -j DROP

# invalid packets
$IPTABLES -N INVAL
$IPTABLES -I INVAL -j LOG --log-prefix "ATTACKINVALIDPACKET "
$IPTABLES -A INVAL -j DROP

# table for ICMP packets
$IPTABLES -N LOGICMP
$IPTABLES -I LOGICMP -j LOG --log-prefix "ATTACKICMP "
$IPTABLES -A LOGICMP -j DROP

$IPTABLES -N LOGIT
$IPTABLES -I LOGIT -m limit --limit 3/s -j VALID
$IPTABLES -A LOGIT -j LOG --log-prefix "ATTACKECHOREQUEST "
$IPTABLES -A LOGIT -j DROP

# table for udp
$IPTABLES -N UDPFLOOD
```

```
#$IPTABLES -I UDPFLOOD -m limit --limit 30/s -j VALID
$IPTABLES -A UDPFLOOD -j LOG --log-prefix "ATTACKUDPFLOOD "
$IPTABLES -A UDPFLOOD -j DROP

# table for fragmented packets
$IPTABLES -N FRAG
$IPTABLES -I FRAG -j LOG --log-prefix "ATTACKFRAGMENTEDPACKET "
$IPTABLES -A FRAG -j DROP

# table for portscanning attempts
$IPTABLES -N PORTSCAN
$IPTABLES -I PORTSCAN -j LOG --log-prefix "ATTACKPORTSCAN "
$IPTABLES -A PORTSCAN -j DROP

# table for SSH brute force
$IPTABLES -N SSHBRUTE
$IPTABLES -I SSHBRUTE -j LOG --log-prefix
↪  "ATTACKSSHBRUTEFORCE "
$IPTABLES -A SSHBRUTE -j DROP

# table for SYN floods
$IPTABLES -N SYNFLOOD
$IPTABLES -A SYNFLOOD -m limit --limit 10/s --limit-burst 3 -j
↪  VALID
$IPTABLES -I SYNFLOOD -j LOG --log-prefix "ATTACKSYNFLOOD "
$IPTABLES -A SYNFLOOD -j DROP

# --------------------------------- RULES
↪  --------------------------------- #
# --------------------------- loopback interface
↪  -------------------------- #
# Allow loopback interface to do anything.
$IPTABLES -A INPUT -i lo -j VALID
$IPTABLES -A OUTPUT -o lo -j VALID

# --------------------------- FRAGMENTED packets
↪  -------------------------- #
# Force fragments check:
# Packets with incoming fragments are to be dropped. This
↪  attack results in a Linux server panic and causes data
↪  loss.
```

```
$IPTABLES -A INPUT -f -j FRAG

# ----------------------------- ICMP packets
↪ ------------------------------ #
# Allow ping but limit to $LIMIT

$IPTABLES -A INPUT -p icmp -m icmp --icmp-type 17 -j LOGICMP
$IPTABLES -A INPUT -p icmp -m icmp --icmp-type 13 -j LOGICMP
$IPTABLES -A INPUT -p icmp -m icmp --icmp-type 8 -j LOGIT
#$IPTABLES -A INPUT -p icmp -m conntrack --ctstate ESTABLISHED
↪  $LIMITICMP -j VALID

$IPTABLES -A OUTPUT -p icmp -m icmp --icmp-type 8 -j VALID
$IPTABLES -A OUTPUT -p icmp -m conntrack --ctstate ESTABLISHED
↪ -j VALID

$IPTABLES -A INPUT -p icmp -j LOGIT

# ------------------------------ UDP flood
↪ -------------------------------- #
$IPTABLES -A INPUT -p udp -j UDPFLOOD
$IPTABLES -A FORWARD -p udp -j UDPFLOOD

# ---------------------------- SSH  ----------------------------
↪  #
$IPTABLES -A OUTPUT -p tcp -m tcp --dport 22 -j VALID
# protection from SSH brute force
# TCP packets coming in and trying to establish an SSH
↪  connection will be marked as SSH. The source of the packet
↪  is regarded.
$IPTABLES -A INPUT -p tcp -m tcp --dport 22 -m conntrack
↪ --ctstate NEW -m recent --set --name SSH --rsource
# If a packet attempting to establish an SSH connection comes,
↪  and it's the fourth packet to come from the same source in
↪  thirty seconds, it is rejected.
$IPTABLES -A INPUT -p tcp -m tcp --dport 22 -m recent --rcheck
↪ --seconds 30 --hitcount 4 --rttl --name SSH --rsource -j
↪  REJECT --reject-with tcp-reset
# If an SSH connection packet comes in, and it's the third
↪  attempt from the same guy in thirty seconds, log it and
↪  immediately dropped.
```

```
$IPTABLES -A INPUT -p tcp -m tcp --dport 22 -m recent --rcheck
 ↪  --seconds 30 --hitcount 3 --rttl --name SSH --rsource -j
 ↪  SSHBRUTE
#Any SSH packet not stopped so far is accepted
$IPTABLES -A INPUT -p tcp -m tcp --dport 22 -j VALID


# ------------------------------ SYN floods
 ↪  ------------------------------ #
$IPTABLES -A INPUT -p tcp  --syn -m conntrack --ctstate NEW -j
 ↪  SYNFLOOD

# ------------------------------ port scanning
 ↪  ------------------------------ #
# protection against port scanning
$IPTABLES -A INPUT -p tcp --tcp-flags SYN,ACK,FIN,RST RST -j
 ↪  PORTSCAN

# SYN + RST scan
$IPTABLES -A INPUT -p tcp --tcp-flags SYN,RST SYN,RST -j
 ↪  PORTSCAN
$IPTABLES -A FORWARD -p tcp --tcp-flags SYN,RST SYN,RST -j
 ↪  PORTSCAN

# SYN + FIN scan
$IPTABLES -A INPUT -p tcp --tcp-flags SYN,FIN SYN,FIN -j
 ↪  PORTSCAN
$IPTABLES -A FORWARD -p tcp --tcp-flags SYN,FIN SYN,FIN -j
 ↪  PORTSCAN

# FIN + URG + PSH scan
$IPTABLES -A INPUT -p tcp --tcp-flags ALL FIN,URG,PSH -j
 ↪  PORTSCAN
$IPTABLES -A FORWARD -p tcp --tcp-flags ALL FIN,URG,PSH -j
 ↪  PORTSCAN

# ALL scan
$IPTABLES -A INPUT -p tcp --tcp-flags ALL ALL -j PORTSCAN
$IPTABLES -A FORWARD -p tcp --tcp-flags ALL ALL -j PORTSCAN
```

```
# null scan
$IPTABLES -A INPUT -p tcp --tcp-flags ALL NONE -j PORTSCAN
$IPTABLES -A FORWARD -p tcp --tcp-flags ALL NONE -j PORTSCAN

# FIN stealth scan
$IPTABLES -A INPUT -p tcp --tcp-flags ALL FIN -j PORTSCAN
$IPTABLES -A FORWARD -p tcp --tcp-flags ALL FIN -j PORTSCAN

# xmas
$IPTABLES -A INPUT -p tcp --tcp-flags ALL
↪  URG,ACK,PSH,RST,SYN,FIN -j PORTSCAN
$IPTABLES -A FORWARD -p tcp --tcp-flags ALL
↪  URG,ACK,PSH,RST,SYN,FIN -j PORTSCAN

# Spoofing and bad address attacks try to fool the server and
↪  claim that packets had come from local addresses/the local
↪  network.
# The following IPs and network addresses are know to open this
↪  kind of attack:
# Incoming source IP address is your servers IP address. Bad
↪  incoming address from following ranges:

# ------------------------------ IP spoofing
↪  -------------------------------- #
# VALID for communication with controller
# $IPTABLES -A INPUT -s 10.0.2.15/24 -j VALID
#$IPTABLES -A INPUT -s 10.0.0.0/8 -j SPOOF
#$IPTABLES -A INPUT -s 10.0.0.0/8 -j SPOOF
$IPTABLES -A INPUT -s 172.16.0.0/12 -j SPOOF
#$IPTABLES -A INPUT -s 192.168.0.0/16 -j SPOOF
$IPTABLES -A INPUT -s 224.0.0.0/4 -j SPOOF
$IPTABLES -A INPUT -s 240.0.0.0/4 -j SPOOF
$IPTABLES -A INPUT -s 255.255.255.255 -j SPOOF
$IPTABLES -A INPUT -s 127.0.0.0/8 ! -i lo -j SPOOF
$IPTABLES -A INPUT -s 0.0.0.0/8 -j SPOOF
$IPTABLES -A INPUT -s 169.254.0.0/16 -j SPOOF
$IPTABLES -A INPUT -s 100.64.0.0/10 -j SPOOF
$IPTABLES -A INPUT -s 192.0.2.0/24 -j SPOOF
```

```
# Packets leaving the network should never have the following
↪  destination IP addresses
$IPTABLES -A INPUT -d 0.0.0.0/8 -j SPOOF
#$IPTABLES -A INPUT -d 10.0.0.0/8 -j SPOOF
$IPTABLES -A INPUT -d 224.0.0.0/4 -j SPOOF
$IPTABLES -A INPUT -d 240.0.0.0/4 -j SPOOF
$IPTABLES -A INPUT -d 255.255.255.255 -j SPOOF

# ------------------- ESTABLISHED and RELATED
↪  connections------------------ #
# Allow incoming connections related to existing allowed
↪  connections, drop invalid packets.
$IPTABLES -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED
↪ -j VALID
$IPTABLES -A INPUT -m conntrack --ctstate INVALID -j INVAL

# Allow outgoing connections EXCEPT invalid
$IPTABLES -A OUTPUT -m conntrack --ctstate ESTABLISHED,RELATED
↪ -j VALID

# ---------------------------- ACCEPTED PACKETS
↪ ----------------------------- #
# accept all other packets
$IPTABLES -A INPUT -j VALID
$IPTABLES -A FORWARD -j VALID
$IPTABLES -A OUTPUT -j VALID
```

Listing A.2: Rules used by the firewall in Section 5.2.1 [Hem17].

A.3 THREADS Modifications

```
diff --git a/dpdkapp/main.c b/dpdkapp/main.c
index 2e77cd9..d1971de 100644
--- a/dpdkapp/main.c
+++ b/dpdkapp/main.c
@@ -31,6 +31,7 @@

#define TCP_FLAG_SYN 0x02
#define TCP_FLAG_ACK 0x10
+#define TCP_FLAG_PSH 0x08

static const struct rte_eth_conf port_conf= {
  .rxmode = { .max_rx_pkt_len = ETHER_MAX_LEN }
@@ -118,16 +119,54 @@ static inline int process_packets
(struct rte_mbuf **bufs, int size) {
      uint8_t tcp_h_len = (tcp_h->data_off & 0xf0) >> 2;
      if (packet_len - (ip_h_len + tcp_h_len) == 0) {
        resp = process_ack_packet(bufs[i]);
-                               if (resp != NULL)
+                               if (resp != NULL){
          bufs[resp_i++] = resp;
+                               }
      }
-                       else {
-                               rte_pktmbuf_free(bufs[i]);
-                       }
-               }
-               else {
-                       rte_pktmbuf_free(bufs[i]);
    }
+
+               // PSH ACK slipped through
+               else if (tcp_h->tcp_flags == (TCP_FLAG_PSH
| TCP_FLAG_ACK)) {
+                       bufs[resp_i++] = bufs[i];
+       }
+
+       // ACK from server slipped through
+         else if (tcp_h->tcp_flags == TCP_FLAG_ACK
&& mac_addr_equal(ether_h->s_addr, server_mac_addr)) {
+             bufs[resp_i++] = bufs[i];
+         }
+
+         else {
+
+             bufs[resp_i++] = bufs[i];
+             // From client
+             if (mac_addr_equal(ether_h->d_addr,
server_mac_addr)) {
+                 //exit(tcp_h->tcp_flags);
+             }
+
+             // From server
+             else if (mac_addr_equal(ether_h->s_addr,
server_mac_addr)) {
+                 //exit(100 + tcp_h->tcp_flags);
+             }
+
+         }
+
+
+//                  else {
+//              if (mac_addr_equal(ether_h->d_addr,
server_mac_addr)) {
+//
+//                  if (tcp_h->tcp_flags == TCP_FLAG_ACK) {
+//                      exit(6);
+//                  } else if (tcp_h->tcp_flags ==
TCP_FLAG_SYN) {
+//                      exit(99);
+//                  } else if (tcp_h->tcp_flags ==
(TCP_FLAG_PSH |
TCP_FLAG_ACK)) {
+//                      exit(200);
+//                  }
+//                  exit(tcp_h->tcp_flags);
+//
+//             } else {
+//                 exit(4);
+//             }
+//                 }
  }

  int nb_on_ring = rte_ring_mp_enqueue_burst(ring_tx,
  (void *const *)bufs, resp_i, NULL);
@@ -292,5 +331,8 @@ int main(int argc, char *argv[])
  }

  rte_eal_remote_launch(tx_core_main, NULL, 1);
+
+        printf("Foo");
+
  return rx_core_main();
}
diff --git a/dpdkapp/packetprocessing.c
b/dpdkapp/packetprocessing.c
index a6ec040..5f449d3 100644
--- a/dpdkapp/packetprocessing.c
+++ b/dpdkapp/packetprocessing.c
@@ -59,8 +59,8 @@ struct rte_mbuf
*process_syn_packet(struct rte_mbuf *mbuf)
  ip_h->version_ihl = 0x45;
  /* no DSCP or ECN */
  ip_h->type_of_service = 0;
-        /* IP header = 20 bytes, TCP header = 20 bytes,
no payload -> 40 bytes */
-        ip_h->total_length = rte_cpu_to_be_16(40);
+        /* IP header = 20 bytes, TCP header = 20 bytes,
+ 4 byte options, no payload -> 44 bytes */
+        ip_h->total_length = rte_cpu_to_be_16(52); // TODO
  /* ignore packet id and fragment offset*/
  ip_h->packet_id = 0;
  ip_h->fragment_offset = 0;
@@ -84,18 +84,47 @@ struct rte_mbuf
*process_syn_packet(struct rte_mbuf *mbuf)
  tcp_h->recv_ack = rte_cpu_to_be_32
  (rte_be_to_cpu_32(tcp_h->sent_seq) + 1);
  /* set SYN cookie ISN */
  tcp_h->sent_seq = rte_cpu_to_be_32(isn);
-        /* no options, so data_offset /
TCP header length is 20 bytes / 4 bytes =
5 (higher 4 bytes) */
-        tcp_h->data_off= 0x50;
+        /* 4 byte options, so data_offset
/ TCP header length is 24 bytes /
4 bytes = 6 (higher 4 bytes) */
+        tcp_h->data_off= 0x80; // TODO
  /* set SYN and ACK flag */
  tcp_h->tcp_flags = 0x12;
  /* Maximum receive window */
-        tcp_h->rx_win = 0xffff;
-
+        tcp_h->rx_win = rte_cpu_to_be_16(28960);
+
+    /*TCP option MSS*/
+    uint8_t *options = tcp_h+1;
+    *options = 2;
+    options++;
+    *options = 4;
+    options++;
+    *options = 5;
+    options++;
+    *options = 0xb4;
+    options++;
+
+    /*TCP option sackOK + nop + nop*/
+    *options = 4;
+    options++;
+    *options = 2;
+    options++;
+    *options = 1;
+    options++;
+    *options = 1;
+
+    /*TCP window scaling*/
+    options++;
+    *options = 3;
+    options++;
+    *options = 3;
+    options++;
+    *options = 7;
+    options++;
+    *options = 1;

  /* MBUF DATA */
  mbuf->nb_segs = 1;
-        mbuf->pkt_len = 54;
-        mbuf->data_len = 54;
+        mbuf->pkt_len = 66; //TODO
+        mbuf->data_len = 66; //TODO

  /* checksums */
  ip_h->hdr_checksum = 0;
@@ -155,8 +184,8 @@ struct rte_mbuf
*process_ack_packet (struct rte_mbuf *mbuf)
  ip_h->version_ihl = 0x45;
  /* no DSCP or ECN */
  ip_h->type_of_service = 0;
-        /* IP header = 20 bytes, TCP header = 20 bytes,
4 bytes of payload -> 44 bytes */
-        ip_h->total_length = rte_cpu_to_be_16(44);
+        /* IP header = 20 bytes, TCP header
= 20 bytes,
4 bytes of payload -> 44 bytes + 4 byte MSS
+ 2 sackOK +
2 nop + 3 windows scaling + 1 nop*/
+        ip_h->total_length = rte_cpu_to_be_16(56);
  /* ignore packet id and fragment offset*/
  ip_h->packet_id = 0;
  ip_h->fragment_offset = 0;
@@ -172,12 +201,46 @@ struct rte_mbuf
*process_ack_packet(struct rte_mbuf *mbuf)
  tcp_h->sent_seq = rte_cpu_to_be_32
  (rte_be_to_cpu_32(tcp_h->sent_seq) - 1);
  /* SYN packet has nothing to ack, set
  to zero */
  tcp_h->recv_ack = 0;
-         /* no options, so data_offset (TCP header
length)
is 20 bytes / 4 bytes = 5 (higher 4 bytes) */
-         tcp_h->data_off= 0x50;
+         /* no options, so data_offset (TCP header
length)
is 20 bytes / 4 bytes = 5 (higher 4 bytes) */
// TODO
+         tcp_h->data_off= 0x80;
  /* set SYN flag */
  tcp_h->tcp_flags = 0x02;
  /* Maximum receive window */
-         tcp_h->rx_win = 0xffff;
+         tcp_h->rx_win = rte_cpu_to_be_16(29200);
+
+
+    uint8_t *options = payload;
+    /*TCP option MSS*/
+    *options = 2;
+    options++;
+    *options = 4;
+    options++;
+    *options = 5;
+    options++;
+    *options = 0xb4;
+    options++;
+
+    /*TCP option sackOK + nop + nop*/
+    *options = 4;
+    options++;
+    *options = 2;
+    options++;
+    *options = 1;
+    options++;
+    *options = 1;
+
+    /*TCP window scaling*/
+    options++;
+    *options = 3;
+    options++;
+    *options = 3;
+    options++;
+    *options = 7;
+    options++;
+    *options = 1;
+
+
+    payload+=3; //TODO

  /* PAYLOAD */
  /* write isn the server should use (4 bytes)
  in the payload */
@@ -193,8 +256,8 @@ struct rte_mbuf
*process_ack_packet(struct rte_mbuf *mbuf)

  /* MBUF DATA */
  mbuf->nb_segs = 1;
-        mbuf->pkt_len = 58;
-        mbuf->data_len = 58;
+        mbuf->pkt_len = 70; // TODO
+        mbuf->data_len = 70; // TODO
```

Listing A.3: Modifications to THREADS for the evaluation environment in Section 5.2.1 [Fel18].

A.4 SDN Rules for Traffic Routing in the Evaluation Environment

Name			Match	Actions
Client	→	SW2	Port-In: 25	Port-Out: 27
SW2	→	Service	Port-In: 27	Port-Out: 26
Service	→	Client	Port-In: 26	Port-Out: 25

(**a**) SDN rules at `SW1`.

Name			Match	Actions
SW1	→	SW4	Port-In: 25	Port-Out: 28
SW4	→	SW1	Port-In: 28	Port-Out: 25

(**b**) SDN rules at `SW2`.

Name			Match	Actions
SW2	→	Firewall	Port-In: 25	Port-Out: 26
Firewall	→	SW2	Port-In: 27	Port-Out: 25

(**c**) SDN rules at `SW4`.

Table A.1: SDN rules for routing traffic via the firewall only.

Name			Match	Actions
Client	→	SW2	Port-In: 25	Port-Out: 27
SW2	→	Service	Port-In: 27	Port-Out: 26
Service	→	Client	Port-In: 26	Port-Out: 25

(**a**) SDN rules at `SW1`.

Name			Match	Actions
SW1	→	DPS	Port-In: 25	Port-Out: 27
DPS	→	SW1	Port-In: 27	Port-Out: 25

(**b**) SDN rules at `SW2`.

Table A.2: SDN rules for routing traffic via the DPS only.

Name			Match	Actions
Client	→	SW2	Port-In: 25	Port-Out: 27
SW2	→	Service	Port-In: 27	Port-Out: 26
Service	→	Client	Port-In: 26	Port-Out: 25

(**a**) SDN rules at `SW1`.

Name			Match	Actions
SW1	→	SW3	Port-In: 25	Port-Out: 26
SW3	→	SW1	Port-In: 26	Port-Out: 25

(**b**) SDN rules at `SW2`.

Name			Match	Actions
SW2	→	IDPS	Port-In: 25	Port-Out: 26
IDPS	→	SW2	Port-In: 27	Port-Out: 25

(**c**) SDN rules at `SW3`.

Table A.3: SDN rules for routing traffic via the IDPS only.

Name			Match	Actions
Client	→	SW2	Port-In: 25	Port-Out: 27
SW2	→	Service	Port-In: 27	Port-Out: 26
Service	→	Client	Port-In: 26	Port-Out: 25

(**a**) SDN rules at `SW1`.

Name			Match	Actions
		IDPS → Firewall		
SW1	→	SW3	Port-In: 25	Port-Out: 26
SW3	→	SW1	Port-In: 26	Port-Out: 28
SW3	→	SW1	Port-In: 28	Port-Out: 25
		Firewall → IDPS		
SW1	→	SW3	Port-In: 25	Port-Out: 28
SW3	→	SW1	Port-In: 28	Port-Out: 26
SW3	→	SW1	Port-In: 26	Port-Out: 25

(**b**) SDN rules at `SW2`.

Name			Match	Actions
SW2	→	IDPS	Port-In: 25	Port-Out: 26
IDPS	→	SW2	Port-In: 27	Port-Out: 25

(**c**) SDN rules at `SW3`.

Name			Match	Actions
SW2	→	Firewall	Port-In: 25	Port-Out: 26
Firewall	→	SW2	Port-In: 27	Port-Out: 25

(**d**) SDN rules at `SW4`.

Table A.4: SDN rules for routing traffic via the IDPS and then the firewall and vice-versa.

Name			Match	Actions
Client	→	SW2	Port-In: 25	Port-Out: 27
SW2	→	Service	Port-In: 27	Port-Out: 26
Service	→	Client	Port-In: 26	Port-Out: 25

(**a**) SDN rules at `SW1`.

Name			Match	Actions
DPS → IDPS				
SW1	→	DPS	Port-In: 25	Port-Out: 27
DPS	→	SW3	Port-In: 27	Port-Out: 26
SW3	→	SW1	Port-In: 26	Port-Out: 25
IDPS → DPS				
SW1	→	SW3	Port-In: 25	Port-Out: 26
SW3	→	DPS	Port-In: 26	Port-Out: 27
DPS	→	SW1	Port-In: 27	Port-Out: 25

(**b**) SDN rules at `SW2`.

Name			Match	Actions
SW2	→	IDPS	Port-In: 25	Port-Out: 26
IDPS	→	SW2	Port-In: 27	Port-Out: 25

(**c**) SDN rules at `SW3`.

Table A.5: SDN rules for routing traffic via the DPS and then the IDPS and vice-versa.

Name			Match	Actions
Client	→	SW2	Port-In: 25	Port-Out: 27
SW2	→	Service	Port-In: 27	Port-Out: 26
Service	→	Client	Port-In: 26	Port-Out: 25

(**a**) SDN rules at `SW1`.

Name			Match	Actions
DPS $\rightarrow$ Firewall				
SW1	→	DPS	Port-In: 25	Port-Out: 27
DPS	→	SW4	Port-In: 27	Port-Out: 28
SW4	→	SW1	Port-In: 28	Port-Out: 25
Firewall $\rightarrow$ DPS				
SW1	→	SW4	Port-In: 25	Port-Out: 28
SW4	→	DPS	Port-In: 28	Port-Out: 27
DPS	→	SW1	Port-In: 27	Port-Out: 25

(**b**) SDN rules at `SW2`.

Name			Match	Actions
SW2	→	Firewall	Port-In: 25	Port-Out: 26
Firewall	→	SW2	Port-In: 27	Port-Out: 25

(**c**) SDN rules at `SW4`.

Table A.6: SDN rules for routing traffic via the DPS and then the firewall and vice-versa.

Appendix B

Detailed Result Tables for Section 5.2.2 and Section 5.2.3

B.1 Single Function Results

Requests per Second	Successful Requests	Error
0	0	±0
2,000	186,000	±3,603
4,000	356,000	±7,107
6,000	534,000	±3,592
8,000	712,000	±6,115
10,000	879,914	±1,491
12,000	1,068,000	±19,436
14,000	1,245,933	±19,293
16,000	1,398,098	±14,589
18,000	1,364,232	±4,316
20,000	1,322,815	±2,123
22,000	1,336,232	±3,720

Table B.1: Successful requests without any security functions as shown in Figure 5.6.

Requests per Second	Successful Requests	Error
0	0	±0
2,000	176,000	±3,603
4,000	352,000	±7,107
6,000	522,000	±3,592
8,000	712,000	±6,115
10,000	870,000	±1,491
12,000	1,056,000	±19,436
14,000	1,217,962	±19,293
16,000	1,352,155	±14,589
18,000	1,336,748	±4,316
20,000	1,309,150	±2,123
22,000	1,277,991	±3,720

Table B.2: Successful requests without only the firewall enabled as shown in Figure 5.7.

Requests per Second	Successful Requests	Error
0	0	±0
500	44,000	±3,603
1,000	88,985	±7,107
1,500	133,474	±3,592
2,000	177,853	±6,115
2,500	222,490	±1,491
3,000	258,982	±19,436
3,500	216,988	±19,293
4,000	174,466	±14,589
4,500	249,901	±4,316
5,000	198,138	±2,123
5,500	259,112	±3,720

Table B.3: Caption

Table B.4: Successful requests with only the DPS enabled as shown in Figure 5.8.

Requests per Second	Successful Requests	Error
0	0	±0
1,000	88,987	±3,603
2,000	177,187	±7,107
3,000	265,183	±3,592
4,000	236,769	±6,115
5,000	145,966	±1,491
6,000	154,919	±19,436

Table B.5: Successful requests without only the IDPS enabled as shown in Figure 5.9.

UDP flood	Successful Results			
strength in	Direct		Firewall	
MBit/s	Result	Error	Result	Error
0	122,000	±1,870	122,000	±1,674
500	124,000	±4	126,000	±1,907
1,000	112,204	±482	82,985	±458
1,500	54,289	±592	30,974	±371
2,000	27,825	±242	16,223	±8
2,500	16,832	±216	12,137	±85
3,000	12,689	±464	9,057	±142
3,500	9,544	±398	8,213	±211
4,000	6,122	±298	3,637	±8
4,500	6,697	±179	4,296	±104
5,000	6,201	±28	5,820	±216

Table B.6: Successful requests during a UDP flood attack with direct connection or only the firewall enabled as shown in Figure 5.10.

HTTP flood	Successful Results			
strength in	Direct		Firewall	
Requests/s	Result	Error	Result	Error
0	120,000	±84	120,000	±952
1,000	120,000	±2,268	120,000	±268
2,000	83,830	±1,278	120,000	±523
3,000	80,000	±230	120,000	±772
4,000	77,769	±520	120,000	±999
5,000	72,000	±277	120,000	±792
6,000	60,000	±2,024	120,000	±719
7,000	50,000	±2,492	118,000	±2,303
8,000	40,000	±1,536	118,000	±634
9,000	32,000	±1,316	120,000	±2,131
10,000	17,978	±608	120,000	±1,018
11,000	19,796	±752	120,000	±1,404
12,000	17,372	±666	120,000	±1,394
13,000	17,877	±581	120,000	±531
14,000	17,877	±658	120,000	±558

Table B.7: Successful requests during an HTTP flood attack with direct connection or only the firewall enabled as shown in Figure 5.11.

SYN flood	Successful Results			
strength in	Direct		DPS	
MBit/s	Result	Error	Result	Error
0	120,000	±352	119,872	±1,029
500	120,000	±1,217	119,872	±2,277
1,000	120,000	±1,939	119,907	±1,588
1,500	119,994	±1,128	119,905	±1,243
2,000	117,861	±1,570	117,902	±334
2,500	112,258	±526	117,883	±1,049
3,000	14,472	±368	117,884	±3,133
3,500	13,619	±650	119,882	±2,379
4,000	13,715	±201	116,277	±1,731
4,500	12,645	±228	117,245	±3,002
5,000	14,473	±331	115,471	±2,878
5,500	13,620	±541	116,621	±3,150
6,000	13,716	±551	67,239	±414
6,500	12,646	±74	33,049	±1,436

Table B.8: Successful requests during a SYN flood attack with direct connection or only the DPS enabled as shown in Figure 5.12.

Intrusion flood	Successful Results			
strength in	Direct		IDPS	
MBit/s	Result	Error	Result	Error
0	124,000	±5,819	122,000	±261
500	124,000	±2,611	124,000	±5,322
1,000	84,003	±3,660	47,955	±1,470
1,500	38,362	±84	23,664	±1,000
2,000	25,521	±562	15,558	±630
2,500	15,358	±437	9,184	±158
3,000	12,921	±611	7,141	±266
3,500	8,953	±216	7,620	±21
4,000	7,152	±139	1,490	±30
4,500	7,801	±375	5,283	±94
5,000	6,163	±104	5,183	±61

Table B.9: Successful requests during an intrusion flood attack with direct connection or only the IDPS enabled as shown in Figure 5.13.

B.2 Security Service Function Chain Results

UPD flood	Successful Results			
strength in	IDPS → Firewall		Firewall → IDPS	
MBit/s	Result	Error	Result	Error
0	124,000	±2,130	123,333	±1,812
500	2,720	±129	115,275	±1,840
1,000	2,489	±41	63,457	±1,878
1,500	1,154	±45	20,950	±740
2,000	980	±40	9,979	±209
2,500	2,315	±35	5,371	±168
3,000	1,314	±55	4,130	±70
3,500	1,656	±14	4,227	±145
4,000	2,292	±47	2,646	±102
4,500	1,002	±15	3,700	±119
5,000	3,347	±53	4,073	±115

Table B.10: Successful requests during a UDP flood attack with the IDPS and the firewall enabled as shown in Figure 5.14.

HTTP flood	Successful Results			
strength in	IDPS → Firewall		Firewall → IDPS	
Requests/s	Result	Error	Result	Error
0	118,000	±2,207	112,387	±905
1,000	112,009	±3,269	113,763	±280
2,000	92,082	±4,566	107,721	±1,916
3,000	94,233	±870	109,991	±417
4,000	77,352	±307	114,247	±2,036
5,000	56,863	±2,118	111,315	±1,253
6,000	54,136	±680	109,976	±1,420
7,000	36,865	±1,830	112,019	±2,009
8,000	33,217	±571	115,568	±1,307
9,000	28,185	±443	111,134	±521
10,000	25,146	±1,042	113,224	±2,000
11,000	24,341	±792	106,761	±862
12,000	23,634	±120	112,586	±30
13,000	28,179	±25	113,224	±404
14,000	24,341	±10	115,369	±1,256

Table B.11: Successful requests during an HTTP flood attack with the IDPS and the firewall enabled as shown in Figure 5.16.

SYN flood	Successful Results			
strength in	DPS $\rightarrow$ IDPS		IDPS $\rightarrow$ DPS	
MBit/s	Result	Error	Result	Error
0	121,992	±1,743	123,968	±1,848
500	123,952	±998	11,560	±17
1,000	123,487	±1,795	4,319	±22
1,500	123,901	±1,966	3,989	±125
2,000	123,908	±317	4,039	±128
2,500	123,897	±689	4,029	±143
3,000	123,882	±251	4,347	±100
3,500	121,870	±3,558	4,131	±127
4,000	113,662	±3,352	3,960	±113
4,500	106,433	±2,449	3,831	±66
5,000	103,728	±4,063	4,180	±154
5,500	94,168	±3,363	4,227	±205
6,000	80,695	±2,823	4,008	±46
6,500	76,719	±2,348	3,855	±191

Table B.12: Successful requests during a SYN flood attack with the DPS and the IDPS enabled as shown in Figure 5.18.

SYN flood strength in MBit/s	Firewall → DPS Result	Firewall → DPS Error	DPS → Firewall Result	DPS → Firewall Error
0	121,962	±173	121,985	±2,311
500	121,942	±280	123,914	±565
1,000	121,914	±1,351	123,930	±1,442
1,500	121,893	±1,201	123,928	±1,975
2,000	121,877	±2,015	123,775	±1,112
2,500	106,412	±1,320	121,919	±199
3,000	100,685	±1,516	121,882	±1,813
3,500	36,508	±322	115,849	±476
4,000	28,113	±1,108	111,361	±1,908
4,500	27,914	±249	94,793	±2,804
5,000	27,241	±347	90,077	±708
5,500	10,881	±420	72,374	±988
6,000	3,688	±170	51,749	±2,548
6,500	3,042	±34	44,803	±2,133

Table B.13: Successful requests during a SYN flood attack with the firewall and the DPS enabled as shown in Figure 5.20.

Intrusion flood strength in MBit/s	IDPS → Firewall Result	IDPS → Firewall Error	Firewall → IDPS Result	Firewall → IDPS Error
0	124,000	±657	122,000	±603
500	124,000	±307	31,732	±373
1,000	43,442	±353	7,238	±44
1,500	21,089	±536	5,004	±0
2,000	14,911	±56	4,862	±95
2,500	10,089	±63	1,273	±16
3,000	5,257	±7	2,344	±24
3,500	5,476	±158	1,564	±2
4,000	5,670	±260	6,748	±88
4,500	5,475	±54	7,007	±24
5,000	5,224	±174	7,267	±41

Table B.14: Successful requests during an intrusion flood attack with the IDPS and the firewall enabled as shown in Figure 5.22.

Appendix C

Routing Configuration Flows for the Security Service Function Chain Reordering Framework

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=6,
    dl_src=52:54:00:91:60:4d,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:09:38:52,
    output:7
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=8,
    dl_src=52:54:00:09:38:52,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
    output:1
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:d3:db:f1,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:d3:db:f1,
    output:9
(4) hard_timeout=300,priority=100,dl_type=0x0800,in_port=10,
    dl_src=52:54:00:d3:db:f1,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
    output:1
```

Listing C.1: Flows for the Firewall-IDPS-DPS configuration for the switch on *C39*.

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
    output:4
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=5,
    dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:d3:db:f1,
    output:1
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:93:cd:2d,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
    output:3
```

Listing C.2: Flows for the Firewall-IDPS-DPS configuration for the switch on *C48*.

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=6,
    dl_src=52:54:00:91:60:4d,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
    output:1
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:09:38:52,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:09:38:52,
    output:7
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=8,
    dl_src=52:54:00:09:38:52,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:d3:db:f1,
    output:9
(4) hard_timeout=300,priority=100,dl_type=0x0800,in_port=10,
    dl_src=52:54:00:d3:db:f1,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
    output:1
```

Listing C.3: Flows for the IDPS-Firewall-DPS configuration for the switch on *C39*.

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
    output:4
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=5,
    dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:09:38:52,
    output:1
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:93:cd:2d,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
    output:3
```

Listing C.4: Flows for the IDPS-Firewall-DPS configuration for the switch on *C48*.

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=6,
    dl_src=52:54:00:91:60:4d,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
    output:1
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:d3:db:f1,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:d3:db:f1,
    output:9
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=10,
    dl_src=52:54:00:d3:db:f1,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:09:38:52,
    output:7
(4) hard_timeout=300,priority=100,dl_type=0x0800,in_port=8,
    dl_src=52:54:00:09:38:52,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
    output:1
```

Listing C.5: Flows for the IDPS-DPS-Firewall configuration for the switch on *C39*.

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
    output:4
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=5,
    dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:d3:db:f1,
    output:1
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:93:cd:2d,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
    output:3
```

Listing C.6: Flows for the IDPS-DPS-Firewall configuration for the switch on *C48*.

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=6,
    dl_src=52:54:00:91:60:4d,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
    output:1
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:09:38:52,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:09:38:52,
    output:7
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=8,
    dl_src=52:54:00:09:38:52,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:d3:db:f1,
    output:9
(4) hard_timeout=300,priority=100,dl_type=0x0800,in_port=10,
    dl_src=52:54:00:d3:db:f1,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
    output:1
```

Listing C.7: Flows for the IDPS-Firewall-DPS configuration for the switch on *C39*.

```
(1) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:e3:6f:ac,
    output:4
(2) hard_timeout=300,priority=100,dl_type=0x0800,in_port=5,
    dl_src=52:54:00:e3:6f:ac,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:09:38:52,
    output:1
(3) hard_timeout=300,priority=100,dl_type=0x0800,in_port=1,
    dl_src=52:54:00:93:cd:2d,nw_src=192.168.66.200,
    nw_dst=192.168.66.201,actions=mod_dl_src:52:54:00:93:cd:2d,
    output:3
```

Listing C.8: Flows for the IDPS-Firewall-DPS configuration for the switch on *C48*.

List of Abbreviations

ACK	TCP ACK (Acknowledgment) packet
AMD	Advanced Micro Devices
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
AVX	Advanced Vector Extensions
BIOS	Basic Input/Output System
CPN	Colored Petri Net
CPU	Central Processing Unit
CSA	Cloud Security Alliance
CVE	Common Vulnerabilities and Exposures
DB	Database
DBMS	Database Management System
DMZ	Demilitarized Zone
DPDK	Data Plane Development Kit
DDoS	Distributed Denial-of-Service
DIMAQS	Dynamic Identification of Malicious Query Sequences
DoS	Denial-of-Service
DPI	Deep Package Inspection
DPS	DDoS Protection System

EAL	Environment Abstraction Layer
FCC	Function Chaining Controller
FCFS	First Come First Serve
FPGA	Field-Programmable Gate Array
FTP	File Transfer Protocol
GET	HTTP request to require data from an HTTP server
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ICMP	Internet Control Message Protocol
IDPS	Intrusion Detection and Prevention System
IDS	Intrusion Detection System
IoT	Internet of Things
IP	Internet Protocol
IPS	Intrusion Prevention System
ISN	Initial Sequence Number
I/O	Input/Output
JSON	JavaScript Object Notation
JWE	JSON Web Encryption
JWS	JSON Web Signature
JWT	JSON Web Token
KVM	Kernel Virtual Machine
L2	Layer 2
LAN	Local Area Network

LoC	Lines of Code
MAC	Media Access Control
MAC*	Message Authentication Code
MSS	Maximum Segment Size
NAT	Network Address Translation
NFV	Network Function Virtualization
NIC	Network Interface Card
NIDS	Network Intrusion Detection System
NMS	Network Management System
ODL	OpenDaylight
ONOS	Open Network Operating System
OF	OpenFlow
OFA	OpenFlow Agent
OID	Object Identifier
OS	Operating System
OSI	Open Systems Interconnection
PMD	Poll Mode Driver
PN	Petri Net
PoC	Proof-of-Concept
POST	HTTP request to submit data so an HTTP server (e.g., via a form)
QoE	Quality of Experience
QoS	Quality of Service
REST	Representational State Transfer
SDN	Software-defined Networking

SECaaS	Security as a Service
SFC	Service Function Chain
SSFC	Security Service Function Chain
SFCing	Security Function Chaining
SSFCing	Security Service Function Chaining
SFCC	Security Function Chaining Controller
SIMD	Single Instruction Multiple Input
SNMP	Simple Network Management Protocol
SoC	System on Chip
SR-IOV	Single-Root I/O Virtualization
SSE	Streaming SIMD Extensions
SSH	Secure Shell
SYN	TCP SYN (Synchronization) packet
SYN+ACK	TCP SYN+ACK (Synchronization and Acknowledgment) Packet
TCB	Transmission Control Block
TCP	Transmission Control Protocol
TDP	Thermal Design Power
THREADS	TCP Handshake Remote Establishment and Dynamic Rerouting using Software-defined Networking
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URG	TCP Urgent Flag

VLAN	Virtualized Local Area Network
VM	Virtual Machine
VNF	Virtualized Network Function
WAF	Web Application Firewall
WSGI	Web Server Gateway Interface

List of Figures

List of Algorithms

List of Tables

List of Listings

Bibliography

[Adv14] Advanced Micro Devices, Inc. *Advanced Power Management Helps Bring Improved Performance to Highly Integrated X86 Processors*. White Paper. Advanced Micro Devices, Inc., 2014 (see page 45).

[Alh+11] Adeeb Alhomoud, Rashid Munir, Jules Pagna Disso, Irfan Awan, and A. Al-Dhelaan. “Performance Evaluation Study of Intrusion Detection Systems”. In: *Procedia Computer Science* 5 (2011), pp. 173–180. DOI: `10.1016/j.procs.2011.07.024` (see page 50).

[AMD18] AMD. *2nd Gen AMD Ryzen™ Processors: XFR 2 and Precision Boost 2*. [Online; accessed 24. Jan. 2019]. Apr. 2018. URL: `https://www.youtube.com/watch?v=426hLGoXDbM` (see pages 44, 45).

[An17] Daniel An. *Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed*. Research rep. Google, Feb. 2017. URL: `https://think.storage.googleapis.com/docs/mobile-page-speed-new-industry-benchmarks.pdf` (see page 112).

[Ant+17] Manos Antonakakis et al. “Understanding the Mirai Botnet”. In: *USENIX Security Symposium*. 2017. URL: `https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-antonakakis.pdf` (see pages 2, 240).

[Atc87] M. Atchley. *Recommendations for Security Policy for All Networked Computers St Lbl*. 1987. URL: `https://cloudfront.escholarship.org/dist/prd/content/qt00j5f3bv/qt00j5f3bv.pdf` (see page 241).

[Axe00] Stefan Axelsson. *Intrusion Detection Systems: A Survey and Taxonomy*. Tech. rep. Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 2000. URL: `http://neuro.bstu.by/ai/To-dom/My_research/Paper-0-again/For-research/D-mining/Anomaly-D/Intrusion-detection/taxonomy.pdf` (see page 244).

[BS14] Subhashree Barada and Santosh Kumar Swain. "A Survey Report on Software Aging and Rejuvenation Studies in Virtualized Environment". In: *International Journal of Computer Science & Engineering Technology (IJCSET)* 5.5 (2014), pp. 541–546. URL: `http://www.ijcset.com/docs/IJCSET14-05-05-132.pdf` (see page 35).

[BSM15] Tom Barbette, Cyril Soldani, and Laurent Mathy. "Fast Userspace Packet Processing". In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. IEEE, May 2015, pp. 5–16. DOI: `10.1109/ancs.2015.7110116` (see page 58).

[Bec15] K. Beckers. *Pattern and Security Requirements*. Springer International Publishing, 2015. DOI: `10.1007/978-3-319-16664-3`. URL: `https://books.google.de/books?id=DvdICAAAQBAJ&pg=PA100&redir_esc=y#v=onepage&q&f=false` (see page 11).

[Bei+19a] Lukas Beierlieb, Lukas Iffländer, Samuel Kounev, and Aleksandar Milenkoski. "Towards Testing the Performance Influence of Hypervisor Hypercall Interface Behavior". In: *Proceedings of the 10th Symposium on Software Performance 2019 (SSP'19)*. Nov. 2019 (see page xiv).

[Bei+19b] Lukas Beierlieb, Lukas Iffländer, Aleksandar Milenkoski, Charles F. Gonçalves, Nuno Antunes, and Samuel Kounev. "Towards Testing the Software Aging Behavior of Hypervisor Hypercall Interfaces". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Nov. 2019. URL: `https://se2.informatik.uni-wuerzburg.de/publications/download/paper/2013.pdf` (see page xv).

[Ber+14] Pankaj Berde et al. "ONOS: Towards an Open, Distributed SDN OS". In: *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14*. ACM Press, 2014. DOI: `10.1145/2620728.2620744` (see pages 53, 271).

[Bia+14] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. "OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch". In: *ACM SIGCOMM Computer Communication Review* 44.2 (Apr. 2014), pp. 44–51. DOI: `10.1145/2602204.2602211` (see page 52).

[Ble+14] Jeremias Blendin, Julius Rückert, Nicolai Leymann, Georg Schyguda, and David Hausheer. “Position Paper: Software-Defined Network Service Chaining”. In: *2014 Third European Workshop on Software Defined Networks*. IEEE. IEEE, Sept. 2014, pp. 109–114. DOI: `10.1109/ewsdn.2014.14` (see pages 4, 60).

[Boi+17] Julien Boite, Pierre-Alexis Nardin, Filippo Rebecchi, Mathieu Bouet, and Vania Conan. “StateSec: Stateful Monitoring for DDoS Protection in Software-defined Networks”. In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, July 2017. DOI: `10.1109/netsoft.2017.8004113` (see page 52).

[Bon+15] Roberto Bonafiglia, Ivano Cerrato, Francesco Ciaccia, Mario Nemirovsky, and Fulvio Risso. “Assessing the Performance of Virtualization Technologies for NFV: A Preliminary Benchmarking”. In: *2015 Fourth European Workshop on Software Defined Networks*. IEEE. IEEE, Sept. 2015, pp. 67–72. DOI: `10.1109/ewsdn.2015.63` (see page 59).

[Bor05] J. Efrim Boritz. “IS practitioners’ views on core concepts of information integrity”. In: *International Journal of Accounting Information Systems* 6.4 (Dec. 2005), pp. 260–279. DOI: `10.1016/j.accinf.2005.07.001` (see page 12).

[Bos+14] Pat Bosshart et al. “P4: Programming Protocol-independent Packet Processors”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95. DOI: `10.1145/2656877.2656890` (see page 31).

[Bre+20] Thomas J. Breen, Ed J. Walsh, Jeff Punch, Amip J. Shah, and Cullen E. Bash. “From Chip to Cooling Tower Data Center Modeling: Part I Influence of Server Inlet Temperature and Temperature Rise across Cabinet”. In: *2010 12th IEEE Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*. IEEE, June 2020, pp. 2–5. DOI: `10.1109/ITHERM.2010.5501421` (see page 221).

[Bre+14] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. “Deep Packet Inspection as a Service”. In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies - CoNEXT ’14*. ACM Press, 2014. DOI: `10.1145/2674005.2674984` (see pages 4, 59).

[Bro19] Jesper Brouer. *Mitigate TCP SYN Flood Attacks with Red Hat Enterprise Linux 7 Beta*. [Online; accessed 11. Nov. 2019]. Nov. 2019. URL: `https://www.redhat.com/en/blog/mitigate-tcp-syn-flood-attacks-red-hat-enterprise-linux-7-beta` (see page 27).

[Bul+18] Jo Van Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution”. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, 2018, pp. 991–1008. URL: `https://www.usenix.org/conference/usenixsecurity18/presentation/bulck` (see page 35).

[Bul+19] Jo Van Bulck et al. “Breaking Virtual Memory Protection and the SGX Ecosystem with Foreshadow”. In: *IEEE Micro* 39.3 (May 2019), pp. 66–74. DOI: `10.1109/MM.2019.2910104` (see page 35).

[Cal+15] Franco Callegati, Walter Cerroni, Chiara Contoli, and Giuliano Santandrea. “Implementing Dynamic Chaining of Virtual Network Functions in OpenStack P latform”. In: *2015 17th International Conference on Transparent Optical Networks (ICTON)*. IEEE. IEEE, July 2015, pp. 1–4. DOI: `10.1109/icton.2015.7193561` (see page 61).

[Cao+15] Lianjie Cao, Puneet Sharma, Sonia Fahmy, and Vinay Saxena. “NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions”. In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. IEEE, Nov. 2015, pp. 93–99. DOI: `10.1109/nfv-sdn.2015.7387412` (see page 59).

[Cas09] J. Casazza. *First the Tick, Now the Tock: Intel Microarchitecture (Nehalem). Intel® Xeon® processor 3500 and 5500 series Intel® Microarchitecture*. White Paper. Intel Cooperation, 2009. URL: `https://www.intel.com/content/dam/doc/white-paper/intel-microarchitecture-white-paper.pdf` (see page 43).

[CN16] Ayushi Chahal and Ritu Nagpal. “Performance of Snort on Darpa Dataset and Different False Alert Reduction Techniques”. In: *3rd International Conference on Electrical, Electronics, Engineering Trends, Communication, Optimization and Sciences (EEECOS)*. 2016. URL: `https://pdfs.semanticscholar.org/9634/2f678949bcae35eabda3cfafeb0d0abe1d32.pdf` (see page 51).

[Che+05] H. Chen, L. Amodeo, F. Chu, and K. Labadi. "Modeling and Performance Evaluation of Supply Chains Using Batch Deterministic and Stochastic Petri Nets". In: *IEEE Transactions on Automation Science and Engineering* (*T-ASE*) 2.2 (Apr. 2005), pp. 132–144. ISSN: 1545-5955. DOI: `10.1109/tase.2005.844537` (see page 47).

[Chi+14] Po-Wen Chi, Chien-Ting Kuo, He-Ming Ruan, Shih-Jen Chen, and Chin-Laung Lei. "An AMI Threat Detection Mechanism Based on SDN Networks". In: *Eighth International Conference on Emerging Security Information, Systems and Technologies* (*SECUWARE 2014*). IARIA, Nov. 2014. URL: `https://www.thinkmind.org/download.php?articleid=securware_2014_9_30_30142` (see pages 3, 57).

[Chi+12] Margaret Chiosi et al. *Network Functions Virtualization* (*NFV*)*, an Introduction, Benefits, Enablers, Challenges & Call for Action*. SDN and OpenFlow World Congress, Darmstadt, Germany. White paper. 2012. URL: `http://portal.etsi.org/NFV/NFV_White_Paper.pdf` (see pages 4, 33, 39, 59).

[Cho+16] Li-Der Chou, Chia-Wei Tseng, Yu-Ki Huang, Kuo-Chung Chen, Tsung-Fu Ou, and Chia-Kuan Yen. "A Security Service on-demand Architecture in SDN". In: *2016 International Conference on Information and Communication Technology Convergence* (*ICTC*). IEEE, Oct. 2016, pp. 287–291. DOI: `10.1109/ictc.2016.7763487` (see page 41).

[Cim17a] Catalin Cimpanu. *A Benevolent Hacker Is Warning Owners of Unsecured Cassandra Databases*. Jan. 2017. URL: `https://www.bleepingcomputer.com/news/security/a-benevolent-hacker-is-warning-owners-of-unsecured-cassandra-databases/` (see page 239).

[Cim17b] Catalin Cimpanu. *Database Ransom Attacks Hit CouchDB and Hadoop Servers*. Jan. 2017. URL: `https://www.bleepingcomputer.com/news/security/database-ransom-attacks-hit-couchdb-and-hadoop-servers/` (see page 239).

[Cim17c] Catalin Cimpanu. *Massive Wave of MongoDB Ransom Attacks Makes 26,000 New Victims*. Feb. 2017. URL: `https://www.bleepingcomputer.com/news/security/massive-wave-of-mongodb-ransom-attacks-makes-26-000-new-victims/` (see pages 239, 240).

[Cim17d] Catalin Cimpanu. *MongoDB Apocalypse: Professional Ransomware Group Gets Involved, Infections Reach 28K Servers*. Jan. 2017. URL: `https://www.bleepingcomputer.com/news/security/mongodb-apocalypse-professional-ransomware-group-gets-involved-infections-reach-28k-servers/` (see page 239).

[Cim17e] Catalin Cimpanu. *MongoDB Hijackers Move on to ElasticSearch Servers*. Jan. 2017. URL: `https://www.bleepingcomputer.com/news/security/mongodb-hijackers-move-on-to-elasticsearch-servers/` (see page 239).

[Com19] Gerald Combs. *Wireshark - Go Deep*. [Online; accessed 15. Jun. 2019]. June 2019. URL: `https://www.wireshark.org` (see page 109).

[Con+17] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. "ShieldFS: The Last Word in Ransomware Resilient Filesystems". In: *Black Hat USA*. 2017. URL: `https://www.blackhat.com/docs/us-17/wednesday/us-17-Continella-ShieldFS-The-Last-Word-In-Ransomware-Resilient-Filesystems-wp.pdf` (see page 241).

[Con+16] Andrea Continella, Alessandro Guagnelli, Giovanni Zingaro, Giulio De Pasquale, Alessandro Barenghi, Stefano Zanero, and Federico Maggi. "ShieldFS: A Self-healing, Ransomware-aware Filesystem". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM. ACM, Dec. 2016. DOI: `10.1145/2991079.2991110` (see page 241).

[CS16] Stacy Cowley and Liam Stack. "Los Angeles Hospital Pays Hackers USD 17,000 After Attack". In: *The New York Times* (Feb. 2016). ISSN: 0362-4331. URL: `https://www.nytimes.com/2016/02/19/business/los-angeles-hospital-pays-hackers-17000-after-attack.html` (see page 240).

[CGS07] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Kaufmann, 2007 (see page 224).

[DSC74] Y. Dalal, C. Sunshine, and V. Cerf. *Specification of Internet Transmission Control Program*. en. Tech. rep. Dec. 1974. URL: `https://tools.ietf.org/html/rfc675` (visited on 03/10/2020) (see page 71).

[DB11] David Day and Benjamin Burns. "A Performance Analysis of Snort and Suricata Network Intrusion Detection and Prevention Engines". In: 2011. URL: `https://www.thinkmind.org/download.php?articleid=icds_2011_7_40_90007` (see pages 3, 51).

[Dow20] Oliver James Dowden. *Cyber Security Breaches Survey 2020*. Tech. rep. [Online; accessed 11. Jul. 2020]. Department for Digital, Culture, Media and Sport, Mar. 2020. URL: `https://www.gov.uk/government/publications/cyber-security-breaches-survey-2020/cyber-security-breaches-survey-2020` (see page 1).

[DPD20] DPDK. *DPDK Framework*. [Online; accessed 27. Jan. 2020]. Jan. 2020. URL: `https://www.dpdk.org` (see page 39).

[Dzu09] Muhaimin Dzulfakar. "Advanced MySQL Exploitation". In: *Black Hat USA*. 2009. URL: `https://www.blackhat.com/presentations/bh-usa-09/DZULFAKAR/BHUSA09-Dzulfakar-MySQLExploit-SLIDES.pdf` (see page 242).

[Edd07] W. Eddy. *TCP SYN Flooding Attacks and Common Mitigations*. RFC 4987. RFC Editor, Aug. 2007. DOI: `10.17487/rfc4987`. URL: `https://tools.ietf.org/html/rfc4987` (see page 25).

[Ern19] N. Ernst. *Turbo-Core ist nicht gleich Turbo-Boost - Test: Phenom II X6 1090T - AMD holt mit 6-Kerner auf - Golem.de*. [Online; accessed 24. Jan. 2019]. Jan. 2019. URL: `https://www.golem.de/1004/74709-3.html` (see pages 43, 44).

[Ext20] Extreme Networks Inc. *OpenFlow Table Match Conditions*. [Online; accessed 23. Jan. 2020]. Jan. 2020. URL: `https://documentation.extremenetworks.com/OpenFlow/EXOS_All/OpenFlow/c_OpenFlow-match-conditions.shtml` (see page 33).

[Far+19] Ivan Farris, Tarik Taleb, Yacine Khettab, and Jaeseung Song. "A Survey on Emerging SDN and NFV Security Mechanisms for IoT Systems". In: *IEEE Communications Surveys & Tutorials* 21.1 (2019), pp. 812–837. DOI: `10.1109/comst.2018.2862350` (see page 59).

[Faw06] Tom Fawcett. "An introduction to ROC analysis". In: *Pattern Recognition Letters* 27.8 (June 2006), pp. 861–874. DOI: `10.1016/j.patrec.2005.10.010` (see page 12).

[Fay+15] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. “Bohatei: Flexible and Elastic DDoS Defense”. In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC’15. Washington, D.C.: USENIX Association, 2015, pp. 817–832. DOI: 10.5555/2831143.2831195 (see pages 4, 59).

[Fel18] Nicolas Fella. “Performance Evaluation of Security Network Function Reordering”. Bachelor Thesis. Am Hubland, Informatikgebäude, 97074 Würzburg, Germany: University of Würzburg, Dec. 2018 (see pages 275, 288, 327).

[Fel20] Nicolas Fella. *Attack Reader – Small tool that reads snort alerts from a unix socket and notifies the wrapper about alerts*. [Online; accessed 21. Apr. 2020]. Apr. 2020. URL: https://gitlab2.informatik.uni-wuerzburg.de/descartes/nfv-security/attackreader (see page 193).

[Flo20] Flowgrammable. *SDN, OpenFlow, Message Layer, Match Flowgrammable*. [Online; accessed 23. Jan. 2020]. Jan. 2020. URL: http://flowgrammable.org/sdn/OpenFlow/message-layer/match/#tab_ofp_1_3 (see page 33).

[FZ14] K. Fox and A. Zaidi. “Considerations for System Power Management and Thermal Options Using i.MX 6 Series Processors”. In: *Freescale Technology Forum*. [Online; Accessed 26. Jan. 2019]. Apr. 2014. URL: http://cache.freescale.com/files/training/doc/ftf/2014/FTF-SDS-F0167.pdf (see page 45).

[Gal+15] Sebastian Gallenmuller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. “Comparison of Frameworks for High-performance Packet IO”. In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE Computer Society. IEEE, May 2015, pp. 29–38. DOI: 10.1109/ancs.2015.7110118 (see pages 4, 58).

[GR03] Tal Garfinkel and Mendel Rosenblum. “A Virtual Machine Introspection Based Architecture for Intrusion Detection”. In: *Proceedings of the Network and Distributed Systems Security Symposium*. 2003, pp. 191–206. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.8367 (see page 23).

[git18] github. *synsanity*. [Online; accessed 11. Nov. 2019]. June 2018. URL: https://github.com/github/synsanity (see page 28).

[Gol18] Markus Goldstein. *bonesi*. [Online; accessed 3. Apr. 2020]. Dec. 2018. URL: `https://github.com/markus-go/bonesi` (see page 134).

[Gol17] Dawid Golunski. *MySQL-Exploit-Remote-Root-Code-Execution-Privesc-CVE-2016-6662*. [Online; accessed 20. Jan. 2019]. May 2017. URL: `https://legalhackers.com/advisories/MySQL-Exploit-Remote-Root-Code-Execution-Privesc-CVE-2016-6662.html` (see pages 241, 246).

[GAV18] Charles F. Gonçalves, Nuno Antunes, and Marco Vieira. "Evaluating the Applicability of Robustness Testing in Virtualized Environments". In: *2018 Eighth Latin-American Symposium on Dependable Computing (LADC)*. IEEE, Oct. 2018, pp. 161–166. DOI: `10.1109/ladc.2018.00027` (see page 35).

[Gor20] Matt Gorham. *2019 Internet Crime Report*. Tech. rep. Federal Bureau of Ivestigation, Feb. 2020. URL: `https://pdf.ic3.gov/2019_IC3Report.pdf` (see page 1).

[GK17] Kannan Govindarajan and Vivekanandan Suresh Kumar. "An Intelligent Load Balancer for Software-defined Networking (SDN) Based Cloud Infrastructure". In: *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, Feb. 2017. DOI: `10.1109/icecct.2017.8117881` (see page 104).

[GC18] Oleg Yur'evich Guzev and Ivan Vladimirovich Chizhov. "SDN Load Balancing for Secure Metrics". In: *Systems and Means of Informatics* (Mar. 2018). DOI: `10.14357/08696527180110` (see page 104).

[Hag+18] Christoph Hagen, Alexandra Dmitrienko, Lukas Iffländer, Michael Jobst, and Samuel Kounev. "Efficient and Effective Ransomware Detection in Databases". In: *34th Annual Computer Security Applications Conference (ACSAC)*. ACM. Dec. 2018. URL: `https://se2.informatik.uni-wuerzburg.de/publications/download/paper/1797.pdf` (see pages xv, 10).

[Hal19] R. Hallock. *Gaming: Understanding Precision Boost 2 in AMD Community*. [Online; accessed 24. Jan. 2019]. Jan. 2019. URL: `https://community.amd.com/community/gaming/blog/2017/11/27/asdasd` (see page 44).

[Han+15] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. "Network Function Virtualization: Challenges and Opportunities for Innovations". In: *IEEE Communications Magazine* 53.2 (Feb. 2015), pp. 90–97. ISSN: 0163-6804. DOI: `10.1109/mcom.2015.7045396` (see page 34).

[Has20] HashiCorp. *Vagrant*. Apr. 2020. URL: `https://www.vagrantup.com/` (see pages 138, 224).

[Haw+14] Hassan Hawilo, Abdallah Shami, Maysam Mirahmadi, and Rasool Asal. "NFV: State of the Art, Challenges, and Implementation in Next Generation Mobile Networks (vepc)". In: *IEEE Network* 28.6 (Nov. 2014), pp. 18–26. ISSN: 1558-156X. DOI: `10.1109/mnet.2014.6963800` (see pages 34–37).

[Hem17] Christina Hempfling. "A Packet-Filtering Firewall for Cloud Computing". Project Thesis. Am Hubland, Informatikgebäude, 97074 Würzburg, Germany: University of Würzburg, Sept. 2017 (see pages 281, 327).

[Hem20] Christina Hempfling. *Fire Guardian of the Cloud*. [Online; accessed 21. Apr. 2020]. Apr. 2020. URL: `https://gitlab2.informatik.uni-wuerzburg.de/descartes/nfv-security/FireGuardian` (see page 193).

[Her+15] Nikolas Roman Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. "BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments". In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*. Acceptance rate: 29%. Firenze, Italy: IEEE, May 2015. DOI: `10.1109/seams.2015.23` (see page 224).

[Høi+18] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. "The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel". In: *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies - CoNEXT '18*. ACM. ACM Press, 2018, pp. 54–66. DOI: `10.1145/3281411.3281443` (see page 59).

[Hou20] Alastair Houghton. *netifaces*. [Online; accessed 21. Apr. 2020]. Apr. 2020. URL: `https://pypi.org/project/netifaces` (see page 192).

[HP03] Yi Hu and B. Panda. "Identification of Malicious Transactions in Database Systems". In: *International Database Engineering and Applications Symposium (IDEAS)*. IEEE Comput. Soc, 2003. DOI: 10.1109/ideas.2003.1214946 (see pages 243, 244).

[Hua19] Huawei Inc. *GPU Turbo - Crazy Fast and Power Saving - Honor (Global)*. [Online; accessed 5. Jan. 2019]. Huawei. Jan. 2019. URL: https://www.hihonor.com/global/GPU-Turbo (see page 42).

[Iff16] Lukas Iffländer. "Performance Assessment of Service Migration Strategies". Master Thesis. Am Hubland, Informatikgebäude, 97074 Würzburg, Germany: University of Würzburg, Jan. 2016 (see page xvi).

[ID17] Lukas Iffländer and Alexander Dallmann. "Der Grader PABS". In: *Automatische Bewertung in der Programmierausbildung*. Ed. by Oliver J. Bott, Peter Fricke, Uta Priss, and Michael Striewe. Vol. 6. Digitale Medien in der Hochschullehre. ELAN e.V. and Waxmann Verlag, 2017. Chap. 15, pp. 241–254 (see page xvi).

[Iff+15] Lukas Iffländer, Alexander Dallmann, Philip-Daniel Beck, and Marianus Ifland. "PABS-a Programming Assignment Feedback System". In: *Proceedings of the secod workshop for automated grading of programming exercises (ABP)*. 2015. URL: http://ceur-ws.org/Vol-1496/paper5.pdf (see page xv).

[Iff+19a] Lukas Iffländer, Alexandra Dmitrienko, Christoph Hagen, Michael Jobst, and Samuel Kounev. *Hands Off my Database: Ransomware Detection in Databases through Dynamic Analysis of Query Sequences*. Tech. rep. Universität Würzburg, July 2019. eprint: 1907.06775. URL: https://arxiv.org/abs/1907.06775 (see pages xvi, 10).

[IF19] Lukas Iffländer and Nicolas Fella. "Performance Influence of Security Function Chain Ordering". In: *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. Mumbai, India: ACM, 2019, pp. 45–46. DOI: 10.1145/3302541.3311965 (see pages xv, 8).

[IG12] Lukas Iffländer and Nils Gageik. "Entwicklung und Evaluierung eines Systems zur Bestimmung der Orientierung und Position eines Objektes durch inertiale und magnetische Sensoren". Bachelor Thesis. University of Würzburg, Dec. 2012 (see page xvi).

[Iff+18a] Lukas Iffländer, Stefan Geißler, Jürgen Walter, Lukas Beierlieb, and Samuel Kounev. "Addressing Shortcomings of Existing DDoS Protection Software Using Software-Defined Networking". In: *Proceedings of the 9th Symposium on Software Performance 2018 (SSP'18)*. Hildesheim, Germany, Nov. 2018 (see pages xv, 7).

[Iff+18b] Lukas Iffländer, Christopher Metter, Florian Wamser, Phuoc Tran-Gia, and Samuel Kounev. "Performance Assessment of Cloud Migrations from Network and Application Point of View". In: *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Mobile Networks and Management*. Ed. by Jiankun Hu, Ibrahim Khalil, Zahir Tari, and Sheng Wen. Vol. MONAMI 2017. Cham: Springer International Publishing, 2018, pp. 262–276. DOI: `10.1007/978-3-319-90775-8_21` (see page xiii).

[Iff+20a] Lukas Iffländer, Nishant Rawtani, Lukas Beierlieb, Nicolas Fella, Klaus-Dieter Lange, and Samuel Kounev. "Implementing Attack-aware Security Service Function Chain Reordering". In: *2020 Workshop on Self-Aware Computing - SEAC 2020*. May 2020 (see pages xv, 8, 9).

[Iff+20b] Lukas Iffländer, Nishant Rawtani, Hayreddin Ciner, Lukas Beierlieb, Klaus-Dieter Lange, and Samuel Kounev. "Architecture for a Dynamic Security Service Function Chain Reordering Framework". In: *1st IEEE International Conference on Autonomic Computing and Self-Organizing Systems - ACSOS 2020*. Aug. 2020 (see pages xiv, 8).

[Iff+20c] Lukas Iffländer, Norbert Schmitt, Andreas Knapp, and Samuel Kounev. "Heat-aware Load Balancing-Is it a Thing?" In: *Proceedings of the 11th Symposium on Software Performance 2020 (SSP'20)*. Nov. 2020 (see page xiv).

[Iff+19b] Lukas Iffländer, Jonathan Stoll, Nishant Rawtani, Veronika Lesch, Klaus-Dieter Lange, and Samuel Kounev. "Performance Oriented Dynamic Bypassing for Intrusion Detection Systems". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. ICPE '19. Mumbai, India: ACM, 2019, pp. 159–166. DOI: `10.1145/3297663.3310313` (see pages xiv, 7).

[Iff+18c] Lukas Iffländer, Jürgen Walter, Simon Eismann, and Samuel Kounev. "The Vision of Self-aware Reordering of Security Network Function Chains". In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*. ACM Press, 2018. DOI: `10.1145/3185768.3186309` (see pages xiv, 8).

[Inf20a] InfluxData. *InfluxDB*. [Online; accessed 15-December-2018]. Apr. 2020. URL: `https://www.influxdata.com/products/influxdb-overview/` (see pages 136, 222).

[Inf20b] InfluxData. *Telegraf*. [Online; accessed 15-December-2018]. Apr. 2020. URL: `https://www.influxdata.com/time-series-platform/telegraf/` (see pages 136, 222).

[Inf20c] InfluxData. *Telegraf Plugin: CPU*. [Online; accessed 15-December-2018]. Apr. 2020. URL: `https://github.com/influxdata/telegraf/tree/master/plugins/inputs/cpu` (see page 136).

[Inf20d] InfluxData. *Telegraf Plugin: MEM*. [Online; accessed 15-December-2018]. Apr. 2020. URL: `https://github.com/influxdata/telegraf/tree/master/plugins/inputs/mem` (see page 136).

[Int08] Intel. *Intel Turbo Boost Technology in IntelCore Microarchitecture (Nehalem) Based Processors*. White Paper. 2008 (see page 43).

[Int18] Intel. *Frequently Asked Questions for Intel Turbo Boost Technology*. [Online; accessed 2. Dec. 2018]. Intel. Nov. 2018. URL: `https://www.intel.com/content/www/us/en/support/articles/000007359/processors/intel-core-processors.html` (see page 42).

[IWS04] Blake Ives, Kenneth R. Walsh, and Helmut Schneider. "The Domino Effect of Password Reuse". In: *Communications of the ACM* 47.4 (Apr. 2004), pp. 75–78. DOI: `10.1145/975817.975820` (see page 245).

[Jak+16] A. H. M. Jakaria, Wei Yang, Bahman Rashidi, Carol Fung, and M. Ashiqur Rahman. "VFence: A Defense Against Distributed Denial of Service Attacks Using Network Function Virtualization". In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE. IEEE, June 2016, pp. 431–436. DOI: `10.1109/compsac.2016.219` (see pages 3, 52).

[Jar+14] Michael Jarschel, Thomas Zinner, Tobias Hossfeld, Phuoc Tran-Gia, and Wolfgang Kellerer. “Interfaces, Attributes, and Use Cases: A Compass for SDN”. In: *IEEE Communications Magazine* 52.6 (June 2014), pp. 210–217. DOI: `10.1109/mcom.2014.6829966` (see pages 30, 31).

[JBS15] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. `http://www.rfc-editor.org/rfc/rfc7519.txt`. RFC Editor, May 2015. URL: `http://www.rfc-editor.org/rfc/rfc7519.txt` (see page 192).

[KV16] Jakub Kicinski and Nicolaas Viljoen. “eBPF Hardware Offload to SmartNICs: cls bpf and XDP”. In: *Proceedings of netdev* 1 (2016). URL: `https://www.netronome.com/m/documents/eBPF_HW_OFFLOAD_HNiMne8_2_.pdf` (see page 59).

[KDK18] Joakim von Kistowski, Maximilian Deffner, and Samuel Kounev. “Run-Time Prediction of Power Consumption for Component Deployments”. In: *2018 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Sept. 2018. DOI: `10.1109/icac.2018.00025` (see pages 134, 224).

[Kis+18] Jóakim von Kistowski, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. “TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research”. In: *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. MASCOTS ’18. Milwaukee, WI, USA, Sept. 2018 (see page 135).

[Kis+17] Jóakim von Kistowski, Nikolas Herbst, Samuel Kounev, Henning Groenda, Christian Stier, and Sebastian Lehrig. “Modeling and Extracting Load Intensity Profiles”. In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 11.4 (Jan. 2017), pp. 1–28. ISSN: 1556-4665. DOI: `10.1145/3019596`. URL: `http://doi.acm.org/10.1145/3019596` (see page 134).

[Koc+19] P. Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *Proc. IEEE Symp. Security and Privacy (SP)*. May 2019, pp. 1–19. DOI: `10.1109/SP.2019.00002` (see page 35).

[Kol+17a] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. “DDoS in the IoT: Mirai and Other Botnets”. In: *Computer* 50.7 (2017), pp. 80–84. DOI: `10.1109/mc.2017.201` (see pages 16, 69).

[Kol+17b] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. “PayBreak: Defense Against Cryptographic Ransomware”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, Apr. 2017. DOI: `10.1145/3052973.3053035` (see page 241).

[Kop18] Alexey Kopytov. *akopytov/sysbench*. [Online; accessed 1. Jun. 2018]. June 2018. URL: `https://github.com/akopytov/sysbench` (see page 259).

[KLK20] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking. For Scientists and Engineers*. 1st ed. Springer International Publishing, 2020. DOI: `10.1007/978-3-030-41705-5` (see page 15).

[Kou+17] Samuel Kounev et al. “The Notion of Self-Aware Computing”. In: *Self-Aware Computing Systems*. Ed. by Samuel Kounev, Jeffrey O. Kephart, Aleksandar Milenkoski, and Xiaoyun Zhu. Berlin Heidelberg, Germany: Springer Verlag, 2017. URL: `https://www.springer.com/de/book/9783319474724` (see page 131).

[Kou+15] Michail-Alexandros Kourtis et al. “Enhancing VNF Performance by Exploiting SR-IOV and DPDK Packet Processing Acceleration”. In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE, Nov. 2015. DOI: `10.1109/nfv-sdn.2015.7387409` (see page 39).

[KS94] Sandeep Kumar and Eugene H. Spafford. *A Pattern Matching Model for Misuse Intrusion Detection*. Tech. rep. Purdue University, 1994. URL: `https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=2169&context=cstech` (see page 243).

[KBG19] Oleg Kupreev, Ekaterina Badovskaya, and Gutnikov Gutnikov. *DDoS attacks in Q2 2019*. Tech. rep. Kaspersky Labs, Aug. 2019. URL: `https://securelist.com/ddos-report-q2-2019/91934` (see page 16).

[Lab20] Grafana Labs. *Grafana: The Open Observability Platform*. Apr. 2020. URL: `https://grafana.com/` (see page 136).

[LT19] Klaus-Dieter Lange and Michael G. Tricker. *Server Efficiency Rating Tool (SERT) Design Document 2.0.3*. Tech. rep. Standard Performance Evaluation Corporation (SPEC), Nov. 2019 (see page 224).

[Lan+15] Stanislav Lange et al. “Performance Benchmarking of a Software-Based LTE SGW”. In: *2015 11th International Conference on Network and Service Management (CNSM)*. IEEE, Nov. 2015, pp. 378–383. DOI: 10.1109/cnsm.2015.7367386 (see page 59).

[Li+17] Guanglei Li, Huachun Zhou, Guanwen Li, and Bohao Feng. “Application-aware and Dynamic Security Function Chaining for Mobile Networks”. In: *J. Internet Serv. Inf. Secur.* 7 (2017), pp. 21–34. DOI: 10.22667/JISIS.2017.11.30.021 (see page 59).

[Li+18] Guanwen Li, Huachun Zhou, Bohao Feng, Guanglei Li, Hongke Zhang, and Teng Hu. “Rule Anomaly-Free Mechanism of Security Function Chaining in 5G”. In: *IEEE Access* 6 (2018), pp. 13653–13662. DOI: 10.1109/access.2018.2810834 (see page 62).

[Lip+18] Moritz Lipp et al. “Meltdown”. In: *arxiv* (Jan. 2018). arXiv: http://arxiv.org/abs/1801.01207v1 [cs.CR]. URL: http://arxiv.org/pdf/1801.01207v1 (see page 35).

[Lor+17] Claas Lorenz et al. “An SDN/NFV-Enabled Enterprise Network Architecture Offering Fine-Grained Security Policy Enforcement”. In: *IEEE Communications Magazine* 55.3 (Mar. 2017), pp. 217–223. DOI: 10.1109/mcom.2017.1600414cm (see pages 4, 59).

[LO09] G. Loukas and G. Oke. “Protection Against Denial of Service Attacks: A Survey”. In: *The Computer Journal* 53.7 (Aug. 2009), pp. 1020–1037. DOI: 10.1093/comjnl/bxp078 (see page 12).

[Lue20] Knud Lasse Lueth. *State of the IoT 2018: Number of IoT devices now at 7B - Market accelerating*. [Online; accessed 31. Jan. 2020]. Jan. 2020. URL: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b (see page 16).

[Lui+15] Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol, Marinho Pilla Barcellos, and Luciano Paschoal Gaspary. “Piecing Together the NFV Provisioning Puzzle: Efficient Placement and Chaining of Virtual Network Functions”. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, May 2015. DOI: 10.1109/inm.2015.7140281 (see pages 4, 62).

[Mac+12] Fumio Machida, Jianwen Xiang, Kumiko Tadano, and Yoshiharu Maeno. "Aging-related Bugs in Cloud Computing Software". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops* (Nov. 2012), pp. 287–292. DOI: `10.1109/ISSREW.2012.97` (see page 35).

[Mar+14] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. "Clickos and the Art of Network Function Virtualization". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 459–473. URL: `https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-martins.pdf` (see pages 38–40).

[Mat+12] Rubens Matos, Jean Araujo, Vandi Alves, and Paulo Maciel. "Characterization of Software Aging Effects in Elastic Storage Mechanisms for Private Clouds". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops* (Nov. 2012), pp. 293–298. DOI: `10.1109/ISSREW.2012.82` (see page 35).

[McB+13] M. McBride, M. Cohn, S. Deshpande, M. Kaushik, M. Mathews, and S. Nathan. "SDN security considerations in the data center". In: *Open Networking Foundation-ONF SOLUTION BRIEF* (2013), pp. 15–16 (see page 57).

[MB06] C. McCarthy and Hossein Bidgoli. "Digital Libraries: Security and Preservation Considerations". In: *Handbook of Information Security, Key Concepts, Infrastructure, Standards, and Protocols*. John Wiley & Sons, 2006, pp. 49–76 (see page 12).

[McK+08] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74. DOI: `10.1145/1355734.1355746` (see page 30).

[Med18] MediaWiki. *MediaWiki/de — MediaWiki, The Free Wiki Engine*. [Online; accessed 4-June-2018]. 2018. URL: `https://www.mediawiki.org` (see page 257).

[Med+14] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. "OpenDaylight: Towards a Model-Driven SDN Controller architecture". In: *Proceeding of IEEE International Symposium on a World of Wire-*

less, Mobile and Multimedia Networks 2014. IEEE, June 2014. DOI: 10.1109/wowmom.2014.6918985 (see pages 53, 271).

[MLK14] Weizhi Meng, Wenjuan Li, and Lam-For Kwok. "EFM: Enhancing the Performance of Signature-based Network Intrusion Detection Systems Using Enhanced Filter Mechanism". In: *Computers & Security* 43 (2014), pp. 189–204. DOI: 10.1016/j.cose.2014.02.006 (see page 50).

[Mil+14] Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, and Samuel Kounev. "Experience Report: An Analysis of Hypercall Handler Vulnerabilities". In: *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, Nov. 2014, pp. 100–111. DOI: 10.1109/issre.2014.24 (see page 35).

[Mil+15] Aleksandar Milenkoski, Bryan D. Payne, Nuno Antunes, Marco Vieira, Samuel Kounev, Alberto Avritzer, and Matthias Luft. "Evaluation of Intrusion Detection Systems in Virtualized Environments Using Attack Injection". In: *Research in Attacks, Intrusions, and Defenses*. Springer International Publishing, 2015, pp. 471–492. DOI: 10.1007/978-3-319-26362-5_22 (see page 35).

[Mil+16] Aleksandar Milenkoski et al. *Security Position Paper: Network Function Virtualization*. Published by Cloud Security Alliance (CSA) - Virtualization Working Group. Mar. 2016. URL: https://cloudsecurityalliance.org/download/security-position-paper-network-function-virtualization/ (see pages 2, 4, 62).

[Moh+16] A. A. Mohammed, Molka Gharbaoui, Barbara Martini, Federica Paganelli, and Piero Castoldi. "SDN Controller for Network-aware Adaptive Orchestration in Dynamic Service Chaining". In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE. IEEE, June 2016, pp. 126–130. DOI: 10.1109/netsoft.2016.7502458 (see page 60).

[Mor17] Steve Morgan. *Cybersecurity Business Report. Ransomware Damage Costs predicted to hit USD 11.5B by 2019*. 2017. URL: https://www.csoonline.com/article/3237674/ransomware-damage-costs-predicted-to-hit-115b-by-2019.html (see page 239).

[NSV16] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. "NFVPerf: Online performance monitoring and bottleneck detection for NFV". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2016, pp. 154–160. DOI: `10.1109/nfv-sdn.2016.7919491` (see page 34).

[Ope12] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.3.0*. June 2012. URL: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/OpenFlow/OpenFlow-spec-v1.3.0.pdf` (see page 31).

[Ope15] Open Networking Foundation. *OpenFlow Switch Specification, Version 1.5.1*. 2015. URL: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/OpenFlow/OpenFlow-switch-v1.5.1.pdf` (see page 32).

[Ope16a] Open Networking Foundation. *Impact of SDN and NFV on OSS/BSS - ONF Solution Brief*. Mar. 2016. URL: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-OSS-BSS.pdf` (see page 31).

[Ope16b] Open Networking Foundation. *OpenFlow Message Layer*. 2016. URL: `http://flowgrammable.org/sdn/OpenFlow/message-layer/` (see page 32).

[Opp97] Rolf Oppliger. "Internet Security: Firewalls and Beyond". In: *Communications of the ACM* 40.5 (May 1997), pp. 92–102. DOI: `10.1145/253769.253802` (see page 28).

[Ora18] Oracle Corporation. *MySQL 5.7 Manual*. Oracle Corporation. 2018. URL: `https://dev.mysql.com/doc/refman/5.7/en` (see pages 245, 252).

[Pad20] Jose Padilla. *PyJWT*. [Online; accessed 21. Apr. 2020]. Apr. 2020. URL: `https://pypi.org/project/PyJWT` (see page 192).

[PHS16] Kartik Palani, Emily Holt, and Sean Smith. "Invisible and Forgotten: Zero-day Blooms in the IoT". In: *2016 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. IEEE, Mar. 2016. DOI: `10.1109/percomw.2016.7457163` (see page 16).

[PA00] V. Paxson and M. Allman. *Computing TCP's Retransmission Timer*. RFC 2988. RFC Editor, Nov. 2000. DOI: `10.17487/rfc2988` (see page 109).

[Per15] Chad Perrin. "The CIA Triad". In: *TechRepublic* (July 2015). URL: `https://www.techrepublic.com/blog/it-security/the-cia-triad` (see page 11).

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981 (see page 46).

[PR18] Roberto Pietrantuono and Stefano Russo. "Software Aging and Rejuvenation in the Cloud: A Literature Review". In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, Oct. 2018, pp. 257–263. DOI: `10.1109/issrew.2018.00016` (see page 35).

[Pos83a] J. Postel. *TCP maximum segment size and related topics*. RFC 879. `http://www.rfc-editor.org/rfc/rfc879.txt`. RFC Editor, Nov. 1983. DOI: `10.17487/rfc0879`. URL: `http://www.rfc-editor.org/rfc/rfc879.txt` (see page 16).

[Pos83b] J. Postel. *TCP maximum segment size and related topics*. RFC 879. `http://www.rfc-editor.org/rfc/rfc879.txt`. RFC Editor, Nov. 1983. DOI: `10.17487/rfc0879`. URL: `http://www.rfc-editor.org/rfc/rfc879.txt` (see page 27).

[Pra+21a] Thomas Prantl, Lukas Iffländer, Stefan Herrnleben, Simon Engel, Samuel Kounev, and Christian Krupitzer. "Performance Impact Analysis of Securing MQTT Using TLS". In: *2021 ACM/SPEC International Conference on Performance Engineering (ICPE)*. ICPE'21. Apr. 2021 (see page xiv).

[Pra+20] Thomas Prantl, Peter Ten, Lukas Ifflander, Alexandra Dmitrenko, Samuel Kounev, and Christian Krupitzer. "Evaluating the Performance of a State-of-the-Art Group-oriented Encryption Scheme for Dynamic Groups in an IoT Scenario". In: *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, Nov. 2020, pp. 1–8. DOI: `10.1109/mascots50786.2020.9285948` (see page xiii).

[Pra+21b] Thomas Prantl, Peter Ten, Lukas Iffländer, Stefan Herrnleben, Alexandra Dmitrenko, Samuel Kounev, and Christian Krupitzer. "Towards a Group Encryption Scheme Benchmark: A View on Centralized Schemes with focus on IoT". In: *2021 ACM/SPEC International Conference on Performance Engineering (ICPE)*. ICPE'21. Apr. 2021 (see page xiv).

[Pro+18] Andrew Prout et al. "Measuring the Impact of Spectre and Meltdown". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, Sept. 2018, pp. 1–5. DOI: `10.1109/hpec.2018.8547554` (see page 35).

[QW15] Mao Qilin and Shen Weikang. "A Load Balancing Method Based on SDN". In: *2015 Seventh International Conference on Measuring Technology and Mechatronics Automation*. IEEE, June 2015. DOI: `10.1109/icmtma.2015.13` (see page 104).

[Rei20] Kenneth Reitz. *requests*. [Online; accessed 21. Apr. 2020]. Apr. 2020. URL: `https://pypi.org/project/requests` (see page 192).

[Ron20] Armin Ronacher. *Flask*. [Online; accessed 21. Apr. 2020]. Apr. 2020. URL: `https://pypi.org/project/Flask` (see page 192).

[RBR17] Raphael Vicente Rosa, Claudio Bertoldo, and Christian Esteve Rothenberg. "Take Your VNF to the Gym: A Testing Framework for Automated NFV Performance Benchmarking". In: *IEEE Communications Magazine* 55.9 (2017), pp. 110–117. DOI: `10.1109/mcom.2017.1700127` (see page 59).

[Ryg17] Piotr Rygielski. "Flexible Modeling of Data Center Networks for Capacity Management". PhD thesis. University of Würzburg, Germany, Mar. 2017. URL: `https://opus.bibliothek.uni-wuerzburg.de/frontdoor/index/index/docId/14623` (see page 84).

[Ryg18] Piotr Rygielski. *vikin91/BibSpace*. [Online; accessed 31. May 2018]. May 2018. URL: `https://github.com/vikin91/BibSpace` (see page 257).

[San19] Salvatore Sanfilippo. *Hping - Active Network Security Tool*. [Online; accessed 15. Jun. 2019]. June 2019. URL: `http://www.hping.org` (see page 109).

[San20] Salvatore Sanfilippo. *Hping3*. Apr. 2020. URL: `http://www.hping.org/hping3.html` (see page 136).

[Sca+16] Nolen Scaife, Henry Carter, Patrick Traynor, and Kevin R. B. Butler. "CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, June 2016. DOI: `10.1109/icdcs.2016.46` (see page 241).

[SM07] Karen Scarfone and Peter Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS)*. Tech. rep. NIST Special Publication 900-94. 2007. DOI: `10.6028/nist.sp.800-94` (see page 22).

[Sch+03] Lambert Schaelicke, Thomas Slabach, Branden Moore, and Curt Freeland. "Characterizing the Performance of Network Intrusion Detection Sensors". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2003, pp. 155–172. DOI: `10.1007/978-3-540-45248-5_9` (see pages 3, 50).

[SR18] P. Schmid and A. Roos. *AMD: Turbo CORE - CORE Or Boost? AMD's And Intel's Turbo Features Dissected*. July 2018. URL: `https://www.tomshardware.com/reviews/turbo-boost-turbo-core-six-core,2690-2.html` (see page 43).

[Sch+19] Norbert Schmitt, Lukas Iffländer, André Bauer, and Samuel Kounev. "Online Power Consumption Estimation for Functions in Cloud Applications". In: *Proceedings of the 16th IEEE International Conference on Autonomic Computing (ICAC)*. Umea, Sweden: IEEE, June 2019. DOI: `10.1109/icac.2019.00018` (see page xiii).

[Sch19] Arne Schönbohm. *Die Lage der IT-Sicherheit in Deutschland 2019*. Tech. rep. Bundesamt für Sicherheit in der Informationstechnik, Oct. 2019 (see page 1).

[SNS16] Sandra Scott-Hayward, Sriram Natarajan, and Sakir Sezer. "A Survey of Security in Software Defined Networks". In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 623–654. DOI: `10.1109/comst.2015.2453114` (see pages 3, 53, 54).

[sem18] semantic-mediawiki.org. *Semantic MediaWiki*. [Online; accessed 4-June-2018]. 2018. URL: `https://www.semantic-mediawiki.org/` (see page 257).

[Sen06] Soumya Sen. "Performance Characterization & Improvement of Snort As an IDS". In: *Bell Labs Report* (2006). URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.720.2007&rep=rep1&type=pdf` (see pages 3, 49, 50).

[Sez+13] Sakir Sezer et al. “Are we ready for SDN? Implementation challenges for software-defined networks”. In: *IEEE Communications Magazine* 51.7 (July 2013), pp. 36–43. DOI: `10.1109/mcom.2013.6553676` (see page 56).

[Sha+19] Alireza Shameli-Sendi, Yosr Jarraya, Makan Pourzandi, and Mohamed Cheriet. “Efficient Provisioning of Security Service Function Chaining Using Network Security Defense Patterns”. In: *IEEE Transactions on Services Computing* 12.4 (July 2019), pp. 534–549. DOI: `10.1109/tsc.2016.2616867` (see page 62).

[Shi+13a] Seungwon Shin, Phillip A. Porras, Vinod Yegneswaran, Martin W. Fong, Guofei Gu, and Mabry Tyson. “FRESCO: Modular Composable Security Services for Software-Defined Networks.” In: *NDSS*. The Internet Society, 2013, pp. 1–16. URL: `http://dblp.uni-trier.de/db/conf/ndss/ndss2013.html#ShinPYFGT13` (see pages 3, 52).

[Shi+13b] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. “AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS ’13*. ACM. ACM Press, 2013, pp. 413–424. DOI: `10.1145/2508859.2516684` (see pages 52, 113).

[SWK16] Simon Spinner, Jürgen Walter, and Samuel Kounev. “A Reference Architecture for Online Performance Model Extraction in Virtualized Environments”. In: *Proceedings of the 2016 Workshop on Challenges in Performance Methods for Software Development (WOSP-C’16) co-located with 7th ACM/SPEC International Conference on Performance Engineering (ICPE 2016)*. Delft, the Netherlands: ACM Press, Mar. 2016. DOI: `10.1145/2859889.2859893` (see page 209).

[Sta18] Wired Staff. *The Average Webpage Is Now the Size of the Original Doom*. Mar. 2018. URL: `https://www.wired.com/2016/04/average-webpage-now-size-original-doom` (see page 83).

[Sta20] Statista. *Internet of Things Units Installed Base by Category 2014-2020*. [Online; accessed 11. Jul. 2020]. Feb. 2020. URL: `https://www.statista.com/statistics/370350/internet-of-things-installed-base-by-category` (see page 1).

[Sys20] Cisco Systems. *TRex*. [Online; accessed 3. Apr. 2020]. Mar. 2020. URL: `https://trex-tgn.cisco.com` (see page 134).

[Tal+15] T. Taleb, M. Corici, C. Parada, A. Jamakovic, S. Ruffino, G. Karagiannis, and T. Magedanz. “Ease: EPC As a Service to Ease Mobile Core Network Deployment Over Cloud”. In: *IEEE Network* 29.2 (Mar. 2015), pp. 78–88. ISSN: 1558-156X. DOI: 10.1109/MNET.2015.7064907 (see page 38).

[TK13] T. Taleb and A. Ksentini. “Follow Me Cloud: Interworking Federated Clouds and Distributed Mobile Networks”. In: *IEEE Network* 27.5 (Sept. 2013), pp. 12–19. ISSN: 1558-156X. DOI: 10.1109/MNET.2013.6616110 (see page 37).

[Tay14] Martin Taylor. *A Guide to NFV and SDN*. White Paper, Online. 2014. URL: http://www.metaswitch.com/download-guide-to-nfv-sdn (see page 33).

[TW17] Thomas N. Theis and H.-S. Philip Wong. “The End of Moore’s Law: A New Beginning for Information Technology”. In: *Computing in Science & Engineering* 19.2 (Mar. 2017), pp. 41–50. DOI: 10.1109/mcse.2017.29 (see pages 1, 123).

[TL12] Siwnart Thian-ngam and Mayuree Lertwatechakul. “False Positive Decrement for Snort Intrusion Detection”. In: *Engineering and Applied Science Research* 36.3 (3 June 2012), pp. 251–259. URL: https://ph01.tci-thaijo.org/index.php/easr/article/view/1767 (visited on 03/24/2020) (see page 123).

[Tjh+08] Gina C. Tjhai, Maria Papadaki, S. M. Furnell, and Nathan L. Clarke. “Investigating the Problem of Ids False Alarms: An Experimental Study Using Snort”. In: *IFIP International Information Security Conference*. Springer. 2008, pp. 253–267. DOI: 10.1007/978-0-387-09699-5_17.pdf (see page 51).

[VH02] Theuns Verwoerd and Ray Hunt. “Intrusion Detection Techniques and Approaches”. In: *Computer Communications* 25.15 (Sept. 2002), pp. 1356–1365. DOI: 10.1016/s0140-3664(02)00037-3 (see page 244).

[VRB04] Giovanni Vigna, William Robertson, and Davide Balzarotti. “Testing Network-based Intrusion Detection Signatures Using Mutant Exploits”. In: *Proceedings of the 11th ACM conference on Computer and communications security - CCS 04*. ACM. ACM Press, 2004, pp. 21–30. DOI: 10.1145/1030083.1030088 (see page 23).

[Wal19] J. Walrath. *AMD's Turbo Core Technology | PC Perspective*. [Online; accessed 24. Jan. 2019]. Jan. 2019. URL: `https://www.pcper.com/reviews/Processors/AMDs-Turbo-Core-Technology` (see page 44).

[Wam+15] Florian Wamser, Lukas Iffländer, Thomas Zinner, and Phuoc Tran-Gia. "Implementing Application-Aware Resource Allocation on a Home Gateway for the Example of YouTube". In: *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Mobile Networks and Management*. Ed. by Ramón Agüero, Thomas Zinner, Rossitza Goleva, Andreas Timm-Giel, and Phuoc Tran-Gia. Vol. MONAMI 2014. Cham: Springer International Publishing, 2015, pp. 301–312. DOI: `10.1007/978-3-319-16292-8_22` (see page xiii).

[Wam+14a] Florian Wamser, Thomas Zinner, Lukas Iffländer, and Phuoc Tran-Gia. "Demonstrating the Prospects of Dynamic Application-aware Networking in a Home Environment". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM Press, 2014, pp. 149–150. DOI: `10.1145/2619239.2631450` (see page xv).

[Wam+14b] Florian Wamser, Thomas Zinner, Lukas Iffländer, and Phuoc Tran-Gia. "Demonstrating the Prospects of Dynamic Application-aware Networking in a Home Environment". In: *ACM SIGCOMM Computer Communication Review* 44.4 (Aug. 2014), pp. 149–150. ISSN: 0146-4833. DOI: `10.1145/2740070.2631450` (see page xiii).

[Wan+14] An Wang, Yang Guo, Fang Hao, T. V. Lakshman, and Songqing Chen. "Scotch: Elastically Scaling up SDN Control-Plane using vSwitch based Overlay". In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies - CoNEXT '14*. ACM Press, 2014. DOI: `10.1145/2674005.2675002` (see page 56).

[Wei+19] Ofir Weisse et al. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-order Execution*. Tech. rep. KU Leuven, 2019. URL: `https://lirias.kuleuven.be/retrieve/515917` (see page 35).

[WA19] Harald Welte and Pablo Neira Ayuso. *Netfilter Project Homepage - the Netfilter.org Project*. [Online; accessed 26. Jan. 2020]. Dec. 2019. URL: `https://netfilter.org` (see pages 27, 38).

[Xin+14] Tianyi Xing, Zhengyang Xiong, Dijiang Huang, and Deep Medhi. "SDNIPS: Enabling Software-Defined Networking Based Intrusion Prevention System in Clouds". In: *10th International Conference on Network and Service Management (CNSM) and Workshop*. IEEE, Nov. 2014. DOI: 10.1109/cnsm.2014.7014181. URL: http://www.cnsm-conf.org/2014/proceedings/pdf/47 (see pages 4, 57).

[YY15] Qiao Yan and F. Richard Yu. "Distributed Denial of Service Attacks in Software-defined Networking with Cloud Computing". In: *IEEE Communications Magazine* 53.4 (Apr. 2015), pp. 52–59. DOI: 10.1109/mcom.2015.7081075 (see pages 3, 53, 55, 56).

[YL16] Ch Yoon and S. Lee. "Attacking SDN Infrastructure: Are We Ready for the Next-gen Networking". In: *BlackHat-USA-2016* (2016), pp. 17–18 (see pages 3, 53, 55).

[Yoo+15] Changhoon Yoon, Taejune Park, Seungsoo Lee, Heedo Kang, Seungwon Shin, and Zonghua Zhang. "Enabling Security Functions with SDN: A Feasibility Study". In: *Computer Networks*. Vol. 85. Elsevier BV, July 2015, pp. 19–35. DOI: 10.1016/j.comnet.2015.05.005 (see pages 4, 58).

[Zha+16] Peng Zhang, Huanzhao Wang, Chengchen Hu, and Chuang Lin. "On Denial of Service Attacks in Software Defined Networks". In: *IEEE Network* 30.6 (Nov. 2016), pp. 28–33. DOI: 10.1109/mnet.2016.1600109nm (see page 56).

[Zha+13] Ying Zhang et al. "StEERING: A Software-defined Networking for Inline Service Chaining". In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE. IEEE, Oct. 2013, pp. 1–10. DOI: 10.1109/icnp.2013.6733615 (see page 61).

[Zhe+18] Jing Zheng, Qi Li, Guofei Gu, Jiahao Cao, David K. Y. Yau, and Jianping Wu. "Realtime DDoS Defense Using COTS SDN Switches via Adaptive Correlation Analysis". In: *IEEE Transactions on Information Forensics and Security* 13.7 (July 2018), pp. 1838–1853. DOI: 10.1109/TIFS.2018.2805600 (see pages 3, 52, 113).

[Ziv17] Ofri Ziv. *0.2 BTC strikes back, now attacking MySQL databases*. GuardiCore. Feb. 2017. URL: https://www.guardicore.com/2017/02/0-2-btc-strikes-back-now-attacking-mysql-databases (see pages 239, 240, 244, 257).