Bachelor Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Testbed for Security Testing of Smart Contracts

Lukas Denk
Department of Computer Science
Chair of Computer Science II (Secure Software Systems)

Prof. Dr. Alexandra Dmitrienko
Reviewer

Christoph Sendner
Advisor

Submission
16. November 2020

www.uni-wuerzburg.de

Abstract

Ethereum, one of the most popular decentralized blockchain-based platforms [1], manages cryptocurrency worth more than 40 billion US Dollars [2]. A lot of the money is controlled by autonomous programs, so called *smart contracts*. They are executed on the platform and everyone can call their functions; vulnerabilities are therefore easily exploited. The immutable nature of the blockchain sharpens the problem even further by prohibiting belated bug fixes. In fact, there were several cases where attackers were able to steal cryptocurrency worth several millions of US Dollar [3].
This emphasises the need of soundly testing a smart contract before deployment. Unfortunately, existing security analysing tools each cover a different subsets of vulnerabilities [4]. Moreover, even when they test for the same security issues, they often find different results. A solid test run should therefore contain several tools, which makes the whole procedure very laborious.

For these reasons, we provide a testbed which runs on a virtual machine (VM) with several smart contract security analysing tools integrated. It does not only offer a command line- but also an intuitive web interface. Both interfaces allow the user to analyse his contract with all the underlying tools in a single step. Additionally, our testbed provides a detailed report for each of the tools' test runs as well as an overview of the overall security issues the different tools found.
Our evaluation shows that our testbed identifies significantly more potential security issues than each of the compared smart contract analysing tools individually does. Indeed, we found 92% of our analysed contracts as potentially being vulnerable, while even the most sensitive tool only flagged 76% of its successfully analysed contracts. Furthermore, we observed that occasionally, two tools testing for the same vulnerability, contradict themselves in more than 81% of the contracts successfully analysed by both tools.

Zusammenfassung

Ethereum, eine der beliebtesten Blockchain-basierten Plattformen [1], verwaltet virtuelles Geld im Wert von mehr als 40 Milliarden US-Dollar [2]. Ein großer Teil wird dabei von autonomen Programmen, sogenannten *Smart Contract*s, kontrolliert. Da Ethereum frei zugänglich ist, kann jeder die Funktionen solcher Programme aufrufen. Eventuell vorhandene Sicherheitslücken werden folglich leicht ausgenutzt. Die Unveränderlichkeit der Blockchain verschärft diese Problematik weiter, indem sie nachträgliche Fehlerkorrekturen verhindert. Tatsächlich gab es bereits mehrere Fälle, in denen Kryptowährungen im Wert von je mehreren Million US-Dollar entwendet werden konnte [3, 5].
Dementsprechend wichtig ist es, Smart Contracts bereits im Vorhinein ausgiebig zu testen. Leider decken existierende Analysetools jedoch immer nur einen Teil der bekannten Sicherheitsprobleme ab. Und selbst wenn mehrere Tools auf die gleiche Sicherheitslücke hin testen, liefern sie dennoch oft unterschiedliche Ergebnisse. Ein guter Testdurchlauf sollte daher mehrere dieser Tools beinhalten, was die ganze Prozedur allerdings sehr aufwändig macht.

Aus diesen Gründen haben wir ein Testbett entwickelt, das auf einer virtuellen Maschine läuft und mehrere Analysetools vorinstalliert hat. Unser Testbett stellt dabei nicht nur eine benutzerfreundliche Kommandozeilenschnittstelle, sondern auch ein intuitives Web-Interface zur Verfügung. Beide Schnittstellen erlauben dem Benutzer, seinen Smart Contract mit allen eingebetteten Tools in einem einzigen Schritt zu analysieren. Zusätzlich liefert unser Testbett eine Übersicht über die gefundenen Sicherheitslücken der verschiedenen Tools, sowie einen detaillierten Einzelbericht für jedes Tools.
Unsere Evaluation zeigt, dass unser Testbett bedeutend mehr potentielle Sicherheitslücken aufdeckt, als jedes der Tools für sich allein genommen. Während unser Testbed in über 92% der analysierten Verträge Schwachstellen vermutete, stufte selbst unser empfindlichstes Tool nur 76% der erfolgreich getesteten Verträge als problematisch ein. Des weiteren beobachteten wir, dass sich manche Tools in mehr als 81% der Fälle uneinig sind, ob ein Smart Contract für ein bestimmtes Angriffsszenario anfällig ist oder nicht.

Contents

1. Introduction

A blockchain is an append-only data structure, where old entries can never be changed again. The technology became popular when Satoshi Nakamoto introduced Bitcoin in 2008, a decentralized network which uses a blockchain to manage a financial ledger [6]. For the first time in history, users could anonymously send money to each other without the need of a central authority like a bank. Vitalik Buterin later extended the idea and introduced Ethereum in 2014 [7]. The Ethereum platform can not only record financial transactions but also store and execute smart contracts [7].

A smart contract represents a set of rules on which its users agree on. The rules are enforced by the code of the contract and can be invoked by any member of the network. Since Ethereum is a public and permissionless platform, anyone can participate in the network and therefore easily exploit bugs of a contract. Every smart contract is also associated with a contract account, making them a popular target for attackers [7]. In some cases, like the DAO exploit or the Parity wallet, attackers were able to steal cryptocurrencies worth millions of US Dollars, respectively [3, 5]. When a security bug in a deployed smart contract is detected, it cannot be fixed due to the append-only property of the blockchain [7]. Consequently, it is crucial for a developer to test a contract extensively before deployment.
Fortunately, there already exist a lot of smart contract security analyzing tools. Many of them are even free and open source. However, they mostly use different analysing strategies and look for different security issues [4]. Hence, a good practice is to test a contract with a variety of tools. Unfortunately, this would be very cumbersome, mainly because of the following reasons:

1. It takes a lot time to test a contract with several tools manually.

2. Moreover, most tools only provide a command line interface. Therefore, the user must invest further time into reading the tool's documentation.

3. Every tool reports its findings in an individual way. Many tools, e.g., only provide the security issues they have detected but do not specify the ones they have searched for but have not found. Consequently, it is very exhausting to compare the various findings of the tools. However, such an overview can be very helpful: A security issue looked for by many tools but only flagged by a few of them might indicate a false positive. On the other hand, a security issue found by many tools might imply a true positive.

This lack of usability will either result in a high and unnecessary time consumption or in an unsound or non-existing testing.

To solve the issues mentioned above, our thesis contributes in the following ways:

1. **A Testbed Combining Multiple Analysing Tools.** We propose a testbed for smart contracts, embedded on a VM. It has eight tools pre-installed and can test a contract with all of them in just a single step. The built-in tools are Maian [8], Manticore [9], Mythril [10], Osiris [11], Oyente [12], Securify 2.0 [13], SmartCheck [14] and Vandal [15].
2. **Intuitive User Interfaces for the Testbed.** Furthermore, we do not only provide an intuitive command line interface for the testbed but also a user-friendly web page. Consequently, the user does not need to read any documentation to test his contract. As an additional benefit, he can also access the testbed remotely and does not have to install it himself.
3. **Overview of the Tool's Findings.** At the end of a test run, our testbed also gives an overview of the tools' results. The overview shows the security issues the tools searched for and found as well as the ones they searched for but did not find.
4. **Evaluation.** Finally, we supply an evaluation of our testbed on real world smart contracts. To comply with our resources, we limit our test set to 120 contracts. Furthermore, we particularly focus on eight security issues. For each of them, we check on which contracts our testbed, running all of the built-in tools, assumes the appropriate issue. We then compare these results with the findings of each individual tool.
   While our testbed identified at least one of the eight investigated security issues on 111 contracts, the most sensitive tool only suspected 88 contracts for these issues. Partly, the discrepancy can be explained with the fact that even our most versatile tool only covers six of the eight evaluated vulnerabilities. However, we have also discovered, that even when two tools look for the same weakness, they often disagree on whether a contract contains the weakness or not. For some vulnerabilities, this has even been the case in 81% of the contracts successfully analysed by both tools. Additionally, we analysed when our tools failed to test a contract and how much time they took for a successful test run.

Our evaluation proves the benefits of testing a contract with multiple tools and therefore shows the advantage of our testbed. By only running a single tool on a contract, a programmer might miss many false negatives for certain vulnerabilities while he would not get any feedback for other security issues.

Outline. The remaining parts of the thesis are structured as follows. In Chapter 2, we provide background information and explain all the concepts which are important to understand what a smart contract is and how it can be exploited. Afterwards, in Chapter 3, we overview the related work and discuss existing smart contract security analysing tools. In Chapter 4, we describe the architecture of our testbed before providing the actual details of our implementation in Chapter 5. In Chapter 6, we outline the evaluation of our testbed and give conclusive remarks in Chapter 7.

2. Background

In this chapter, we provide the background information necessary for understanding smart contracts and their vulnerabilities.
In Section 2.1, we explain what a blockchain is and, in Section 2.2, introduce a consensus protocol which allows a decentralized platform to manage this data structure. Afterwards, in Section 2.3, we introduce the concept of smart contracts and later, in Section 2.4, present a real world platform, which operates them - the Ethereum Network. In Section 2.5, we then describe Solidity, a popular programming language for writing smart contracts. Next, in Section 2.6, we illustrate common smart contract vulnerabilities and finally, in Section 2.7, demonstrate how an adversary can exploit these weaknesses.

2.1 Blockchain

In a blockchain, new data entries are first combined in a block and the block is then appended on the last block. Thus, they form a chain of blocks. Each block has a header which includes two cryptographic hashes: One over its data entries and another one over the previous block's header [6].
The blockchain's structure makes it easy to detect modifications on data entries. When someone changes a block, its hash value becomes incorrect and hence the next block's hash, all propagating to the newest block. Everyone who knows the hash of the latest block can therefore recalculate all the hash values and check if the blockchain is valid [6].

The Merkle Tree. A popular data structure to store a block's data is a Merkle tree. It is a binary hash tree, where all the leaves are data entries. Every parent is a hash over its children. When a data entry becomes irrelevant, it would be beneficial to delete it to free disk space. If the data were hashed with a simple hash function, the hash would still be needed to validate other entries. In a Merkle tree, however, only the root of a subtree must be stored to verify entries outside the subtree [6].

Figure 2.1. The concept of a blockchain using a Merkle tree can be seen in Figure 2.1. It shows three block headers, each one with the fields *Prev Hash* and *Merkle Root*. Prev Hash stands for the hash of the previous block's header while the Merkle Root stores the hash of the block's data. The second block also shows the tree itself, with the removed leaves *Data0* and *Data1*. As we can see, we do not need to keep these data entries, since *Hash01* enables us to still validate the rest of the tree.

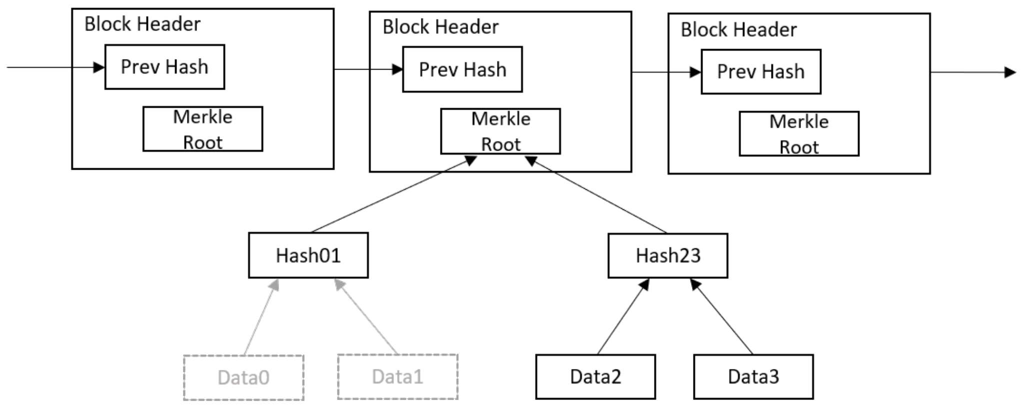


Figure 2.1: The structure of a blockchain using a Merkle tree. Data0 and Data1 are already deleted (cf. [16]).

2.2 Proof of Work

A decentralized platform consists of several independent computers, also called nodes. If the platform is *public*, everyone can join the network and become a node [17].
Operating a blockchain in such a network is not trivial. The blockchain cannot be managed by a central server because of the decentralized nature of the network. Instead, every node stores its own copy. To add a new block, a node broadcasts its updated blockchain version directly to the other nodes. However, this could result in conflicts, for instance when two nodes append new blocks at the same time. For this reason the nodes must listen to a consensus protocol which tells them how to agree on a common state of the blockchain. Furthermore, the protocol must ensure the integrity of the network by punishing dishonest nodes and awarding honest ones [6].
There exist several such protocols, like Proof of Work (PoW) [6], Proof of Space [18] or Proof of Stake [19]. In this thesis, we focus on PoW, since the popular Bitcoin [6] and Ethereum [7] networks implement it.

Functionality. In PoW, each header has an additional value, a *nonce*, which is a bit vector with a fixed length [20]. Before adding a block, a node must find a nonce, so that the hash over the block's header has a certain number of leading 0 bits. There is no better strategy than trying each value in a brute force manner. Consequently, the probability to find a valid bit vector is proportional to the node's computing power [6].
Since the communication between the node takes time, it should not be too easy to compute a fitting nonce. On the other hand, making the calculations too hard would unnecessarily slow the network. For this reason, the network adjusts the number of required leading 0 bits to the current computing power of the network [21].

Awarding the Nodes. Obviously, adding blocks to the blockchain costs resources. Therefore, every time a node adds a block, its effort is compensated with the network's cryptocurrency. A part of the reward comes from newly created cryptocurrency. For this reason the nodes are often called miners. On many platforms, the miners also charge a fee from users who add data to their block [16].

Solving the Nothing-at-Stake Problem. Implementing the PoW protocol has many benefits. For one, it solves the nothing-at-stake problem [22]. On a decentralized network, a node cannot ask a central server about the current blockchain version. Therefore, it can never be sure about the version on which the network currently agrees on. When the node adds a new block, it naturally wants to append it to the commonly shared blockchain version. If adding new blocks was cheap, it would append the new block on every recently received version. Soon, multiple versions would exist and the network would not agree on a single one anymore [23]. PoW, however, prevents this scenario. Adding a block to a blockchain version costs a lot resources, hence the node has a lot on stake. Consequently, it only appends its block to the blockchain version which it believes is most likely the one held by the network's majority. This is always the longest one or, more precisely, the one which was created with the most computing power [6].

Ensuring Integrity. Another benefit of PoW is that it makes it very hard to tamper with the blockchain. When an adversary modifies an old block, all the blocks after it become invalid. Just broadcasting the manipulated blockchain without the successors of the altered block is useless. The other nodes would not accept it since it is not the longest chain. Therefore, the attacker must either append new blocks on the modified one or recalculate the headers of the originally succeeding blocks. However, to outpace the current network with a reasonable probability, it would need the majority of the network's computing power. Otherwise, the adversary's chances decrease exponentially with every block she has to catch up [6, 23].

Distributed Trust. A direct consequence of the last paragraph is that trust in a decentralized system is distributed [24]. This is a strong contrast to a centralized system in which a user must rely entirely on a single entity. A decentralized system, however, always behaves honest, as long as the majority of its nodes is honest [6].

2.3 Smart Contracts

Some blockchain-based, decentralized networks do not only manage financial transactions, but can also execute arbitrary code. Therefore, they serve as a state-based machine, where the current blockchain version is the current machine state.
Physically, such a system is not implemented on a single computer. Instead, every node has a local copy of the current machine state, in this case the current blockchain version. When a blockchain-based, decentralized system executes code, in reality, each node executes the code locally. Since they execute the code on an identical copy of the blockchain, they all end up with the same, new blockchain version [25].

Replacing Real World Contracts. A conventional contract is essentially a set of rules. Therefore it can often be realized as a program. Running on a decentralized network, such programs can replace many real world contracts, since everyone can trust them to be enforced correctly. In fact, this is their main use case, hence they are called *smart contract*s [26].

Example of Selling a House on the Blockchain. Replacing a conventional contract with a smart contract eliminates many disadvantages of a central system. For instance, if Alice wants to sell her house to Bob, they need to go to a notary who ensures that Alice receives the money and Bob the ownership of the house. Such a central authority usually takes high fees. Transferring the house to Bob first and then waiting for him to pay the money is no alternative for Alice. If Bob is dishonest, he might never pay her. The same problem occurs vice versa.
However, in the future, a smart contract might handle the ownership of the house. The contract could store Alice as the current owner of the house and only allow *her* to set a price and mark it for sale. As soon as Bob pays the money to the contract, it would pass it forward to Alice and set Bob as the new owner. Bob can trust the contract since it is public and executed by a decentralized platform. Therefore it is almost safe that the code is executed correctly.

Such a future is not unrealistic. In 2017, the first property, an apartment in Kiev, Ukraine, was sold with merely using a smart contract on the blockchain [27].
Nonetheless, the deal was not legally binding [27]. It will still take time until our society and lawmakers allow smart contracts to handle properties like real estates. The financial industry, on contrary, is faster in adopting new technologies to manage their products. For this reason, smart contracts are currently more often used to trade financial assets [28].

2.4 The Ethereum Network

The Ethereum Network is the most popular decentralized platform for smart contracts [1]. In our thesis, we only consider contracts for this platform. Like other blockchain-based platforms, Ethereum also has an own cryptocurrency, called Ether. Similar to Euros or Dollars, it has a smaller unit called Wei, where 10^{18} Wei are one Ether [7]. Furthermore, the platform has two account types: Externally owned accounts (EOAs) and contract accounts [7]. We now explain how they can be used to execute code and send money to each other.

2.4.1 Externally Owned Accounts

To create an externally owned account, a user needs to generate a pair of a public and a private key. The last 20 bytes of the public key automatically serve as its the address. Each account also has a balance, which is the amount of Ether owned by it [29].

With an EOA, a user can create a transaction. There are two reasons to initiate a transaction: To establish a contract account and deploy a contract on it or to invoke a message call. In both cases, the user can send Ether along with his transaction. In the first case, the Ether is the balance the contract account starts with. Otherwise, it is sent to the address specified in the message. If the address belongs to a contract, the message can additionally activate its code [7].
To execute a transaction, the user must sign it with his private key and send it to a node. The node then verifies the signature and checks whether the balance of the sender covers the costs of the transaction [7].

If all verification passes, the miner executes the transaction on his local copy of the blockchain and includes it into his current block. Other nodes, who later receive the block, can then recalculate the transaction to ensure that the node did not change the blockchain's state in a forbidden way [7].
To guarantee that every node executes a transaction identically, they all run the same virtual machine, the so-called Ethereum Virtual Machine (EVM). The EVM defines an own set of bytecode instructions and has a limited call-stack depth of 1024 [30].
To avoid DoS attacks from users, the Ethereum network introduced the concept of *gas*. Each EVM instruction costs a certain amount of gas, where the amount depends on the expensiveness of the instruction. When a user sends a transaction to a miner, he must compensate the miner's effort by paying Ether for the consumed gas [25].
For this reason, every transaction also includes how much gas it is allowed to spend and how much Ether the sender is willing to pay per gas unit. The money for the gas goes to the node which mines the transaction. If the limit is exceeded, the EVM raises an exception and reverts the transaction, though the miner still keeps the gas money [30].

2.4.2 Contract Accounts

A contract account is controlled by the code of the associated contract. Everyone can call its functions. However, this does not mean that anyone can execute arbitrary code on the contract. The creator could, for example, store his address in the contract. A function could then always check the caller's address before executing further code [7].
Like an EOA, a contract account also has an address and a balance. Though it cannot create a transaction, it still can send message calls or deploy other contracts. However, it neither signs these actions nor are they explicitly recorded. The reason is that every contract can only be activated by another contract or an EOA. Eventually, each message call or contract creation has its origin in a transaction. Therefore, everyone who retraces the transaction can exactly verify any activity triggered by it [30].

Like an EOA, a contract can also limit the gas of its outgoing actions. Furthermore, when a message call or contract creation raises an exception, it is rolled back and the failure is reported to the calling entity [30].

2.5 Solidity

The most popular programming language for smart contracts on Ethereum is Solidity. It is a strictly-typed, high-level language similar to JavaScript [31, 32].
In this section, we introduce some of its basics. Though newer versions have added further constructs, they are unnecessary to understand its concepts. For simplicity we therefore refer to version `0.4.0`.

The Pragma Directive. In the past, the Solidity language has often changed its syntax [33, 34, 35]. As a result, many contracts can only be compiled by certain compiler versions. For this reason, an optional `pragma` directive can be used to specify these versions [36].

Calling a Function. Solidity knows two types of functions: Internal functions, which are implemented as a `JUMP` instruction and externally ones, which are realized as a message call. To call a function `foo` internally, a programmer merely writes `foo()`. If a function `foo` in a contract `Foo` wants to call a function `bar` in a contract `Bar` externally, it can either use a high- or a low-level call [37]. High-level calls are similar to other programming languages. First, contract `Foo` must import the declaration of `Bar`. Afterwards, an instance of contract `Bar` is created, with a statement like `Bar contractBar = Bar(<address of Bar>)`. Now, contract `Foo` can execute the function with `contractBar.foo()` [37].
On the other hand, `foo` can also be called with the low-level function `<address of Bar>.call(<data>)`. In this case, `<data>` consists of the leading four bytes of the hashed function signature and, furthermore, the function's arguments [38, 37]. The signature, in turn, consists of the function's name and its argument types. If `<data>` is either empty or does not match with any of `Bar`'s functions, `Bar`'s `fallback` function is invoked instead. The `fallback` function is always declared as a nameless function without any arguments. If contract `Bar` does not have this function, an exception is thrown [39]. When a function `foo` invokes a function `bar` with the `call` statement, exceptions raised in `bar` are not re-thrown in `foo`. Instead, the `call` invocation simply returns `false` [37].

Sending Money and Limiting Gas. The caller of a function can also send money along with a function call or specify the amount of gas the function is allowed to consume. For a high-level call, the syntax is `foo{value:<amount of wei>, gas:<units of gas>}([<args ...>])`. For a low-level call, it is `call([<data>]).gas(<units of gas>).value(<amount of wei>)` [37, 3].
Whenever money is attached to a call, the invoked function has to be declared `payable`. Otherwise it is not executed and the control flow is forwarded to the `fallback` function, which in turn has to be `payable` [39].
If a contract merely intends to send money, it can also use the `send` function. The invocation of `send` is equivalent to the invocation of `call` with an empty `<data>` argument and a gas limit of 2300. This instruction exists to send money and therefore the tight gas limit only allows for a function call without any expensive, state-changing operations [3].

Global Variables. Every function has access to certain global variables. For example, `msg` provides parameters of the message call which invoked the function, like the address of the sender (with `msg.sender`) or the amount of money sent with the call (with `msg.value`). The `block` variable, on the other hand, contains data of the block, like its timestamp (with `block.timestamp`). Furthermore, a contract can use `this.balance` to check how much Ether it possesses [40].

2.6 Smart Contract Vulnerabilities

Like other programs, smart contracts are susceptible to attacks [3]. In this section, we present examples of vulnerabilities caused by misconceptions of the Solidity language, the EVM or the blockchain itself.

2.6.1 Vulnerabilities Caused by Properties of Solidity

Reentrancy Bug. Whenever a function `foo` in a contract `Foo` calls a function `bar` in an unknown contract `Bar`, the function `bar` might unexpectedly re-enter function `foo`. A Solidity programmer must be especially attentive, since a `call` instruction without an argument or a wrong function signature triggers the `fallback` function. If contract `Foo` updates important variables after the call to `bar`, they are not yet updated when `bar` re-enters `foo`. This is a gateway for *Reentrancy*-Attacks. Attack #1, explained in Section 2.7.1, exploits this property [3].

Over- and Underflows. Other vulnerabilities are caused by unchecked *Over- or Underflows*. For example, when Solidity casts an unsigned integer to a signed one, it does not return the absolute value. Instead, it keeps its bit-wise representation [11]. Attack #2, presented in Section 2.7.1, capitalizes this peculiarity.

Exception Disorder Vulnerability. Most programmers are also unfamiliar with the propagation rules of exceptions thrown in low-level function calls. Therefore, a programmer might just assume that any code after `call` is only executed when `call` has terminated successfully. In this case, a contract possibly has an *Exception Disorder* vulnerability [3]. As a consequence, some tools warn the user, if a `call`'s or `send`'s return value is not checked. An abuse of such a weakness can be seen in Attack #1 in Section 2.7.2.

2.6.2 Vulnerabilities Caused by Properties of the Ethereum Virtual Machine

Callstack-Depth Vulnerability. When the EVM exceeds the limit of its call-stack depth of 1024, it raises an exception. If a contract does not take this into consideration, it possibly has a *Callstack-Depth* vulnerability [3]. Attack #1 in 2.7.2 illustrates an example of a possible assault.

Locked Ether. Some contracts cannot send Ether, even though they can receive it. Transferring money to them is lost forever. Such a vulnerability is called *Locked Ether* [3].

Unprotected Selfdestruct. A contract can also be destroyed to reclaim space and deactivate it. The contract's Ether is then transferred to a given address. Contracts, which do not perform a proper authority check for a self-destruction have an *Unprotected Selfdestruct* weakness [3].

2.6.3 Vulnerabilities Caused by Properties of the Blockchain

Transaction Order Dependence Vulnerability. When a node mines a block, it can freely sort incoming transactions. Therefore, if a user does not mine his transaction himself, he cannot forecast the exact state of the Ethereum network, in which it is executed. Hence, if the different order of a sequence of transactions changes the outcome of a contract, it has a *Transaction Order Dependency (TOD)* vulnerability. Attack #2 in Section 2.7.2 shows how a miner can take advantage of such a scenario.

Timestamp Weakness. There is also some scope in which a miner can tamper with the timestamp. Thus, if the execution of a contract changes with slight modifications of the timestamp, it has a *Timestamp* weakness.

Random Number Vulnerability. Some contracts use pseudo-random numbers based on block values like a block's timestamp. However, the miner of a block can influence these values to a certain degree. Therefore, it is not safe to generate pseudo-random numbers in this way. Consequently, some analyzers warn their users if they use such a pseudo-*Random Number* generator [3].

2.7 Example Attacks

For a better understanding of the security issues described above, we now introduce two contracts and describe how their vulnerabilities can be exploited.

2.7.1 The SimpleDAO contract

The Decentralized Autonomous Organization (DAO) contract was a smart contract implementation of a crowdfunding platform. Unfortunately, it had severe security flaws. In 2016, attackers were able to exploit some of them and stole 3.6 million Ether, which was worth about 60 million US Dollars back then [41].

Here, we introduce `SimpleDAO`, a "simplified version of the DAO, which shares some of the vulnerabilities of the original one" [3]. It is illustrated in Listing 2.1.
With the `donate` function (see line 4), a user can contribute money to another account. The `SimpleDAO` contracts then records the account's address and the devoted money to a `mapping` called `credit` (see line 2). The beneficiary can later call the `withdraw` function to get his money (see line 8). First, the function checks whether the caller is allowed to receive the requested amount of Ether (see line 9). If so, it sends the money and *afterwards* updates `credit` (see lines 10-11).

```
1  contract SimpleDAO {
2      mapping (address => uint) public credit;
3
4      function donate(address to) payable {credit[to] += msg.value;}
5
6      function queryCredit(address to) returns (uint){return credit[to];}
7
8      function withdraw(uint amount) {
9          if (credit[msg.sender]>= amount) {
10             msg.sender.call.value(amount)();
11             credit[msg.sender]-=amount;}
12     }
13 }
```

Listing 2.1: The `SimpleDAO` contract [3].

Attack #1 is similar to the attack against the original DAO contract. With publishing the `Mallory` contract (cf. Listing 2.2), an attacker obtains all of `SimpleDAO`'s Ether by abusing its *Reentrancy* weakness.
The adversary first donates `x` Ether to the `Mallory` contract. In the next step, she invokes `Mallory`'s `fallback` function (see line 7). This function, in turn, calls `SimpleDAO`'s `withdraw` function to demand all the money belonging to the `Mallory` contract. Now, the `SimpleDAO` confirms that the `Mallory` contract is allowed to do so and transfers the money (see Listing 2.7.1 line 9-10).
However, `withdraw` uses `call` with an empty `<data>` argument to send the money. Therefore, the `fallback` function is invoked for another time. Since `SimpleDAO` has not yet updated `credit`, the `withdraw` call, again, succeeds and sends another `x` Ether to `Mallory`. As we can see now, this results in a loop until one of the following conditions occurs: Either the transaction's gas limit is consumed, the call stack is full or `SimpleDAO`'s account is emptied. With several attacks, the attacker can collect all the money from the contract [3].

```
1  contract Mallory {
2      SimpleDAO public dao = SimpleDAO(0x354...);
3      address owner;
4
5      constructor(){ owner = msg.sender; }
```

```
 6
 7      function() payable { dao.withdraw(dao.queryCredit(this)); }
 8
 9      function getJackpot(){ owner.send(this.balance); }
10  }
```

Listing 2.2: The `Mallory` contract [3].

In **Attack #2**, an attacker uses an *Underflow* to steal `SimpleDAO`'s money.
She first publishes `Mallory2` (see Listing 2.3) and then invokes its `attack` function in line 8. The function donates 1 Ether to `SimpleDAO` and immediately withdraws it. Like in the attack introduced above, this leads to the execution of `Mallory2`'s `fallback` function (see line 13).
Now, `Mallory2` withdraws another Ether. `SimpleDAO`'s check in line 9 is passed a second time and `Mallory2`'s `fallback` function called again. This time, though, the `fallback` function returns and `SimpleDAO` updates `credit` (see line 11 in Listing 2.1). However, the mapping is updated twice: Once for the first and another time for the second call to `withdraw`. The first update sets `Mallory2`'s `credit` to 0 and the second one to $2^{256}-1$. The `amount` argument in the `withdraw` function is an unsigned integer and, thus, the EVM performs an underflow if a negative value is assigned to it [11, 3].
To terminate the robbery, the attacker calls `getJackpot()` on `Mallory2` (see line 19) and gains all of `SimpleDAO`'s money [3].

Both attacks could have easily been avoided, if the `SimpleDAO` contract would have first updated the `mapping` and then sent the Ether. Alternatively, it could have also used `send` to transfer the money. In this case, the bound gas limit would have prevented the attackers from calling `withdraw` in their `fallback` function.

```
 1  contract Mallory2 {
 2      SimpleDAO public dao = SimpleDAO(0x818EA...);
 3      address owner;
 4      bool performAttack = true;
 5
 6      constructor(){ owner = msg.sender; }
 7
 8      function attack() {
 9          dao.donate.value(1)(this);
10          dao.withdraw(1);
11      }
12
13      function() payable{
14          if (performAttack) {
15              performAttack = false;
16              dao.withdraw(1); }
17      }
18
19      function getJackpot(){
20          dao.withdraw(dao.balance);
21          owner.send(this.balance);
22      }
23  }
```

Listing 2.3: The `Mallory2` contract [3].

2.7.2 The GovernMental contract

`GovernMental` is a game, where a player can invest half of its `jackpot` to become the `lastInvestor` (see lines 13-18 in Listing 2.4).
If no one contributes to the contract for at least a minute, a call to the `resetInvestment` function pays the `jackpot` to the `lastInvestor` (see line 20-23). Additionally, it transfers the

remaining Ether to the `owner` of the contract, except for 1 Ether (see line 24). Finally, the last Ether becomes the new `jackpot` and the game starts again (see lines 26-28) [3].

```
1  contract GovernMental {
2      address public owner;
3      address lastInvestor;
4      uint public jackpot = 1 ether;
5      uint public lastInvestmentTimestamp;
6      uint public ONE_MINUTE = 1 minutes;
7  
8      constructor() {
9          owner = msg.sender;
10         if (msg.value<1 ether) throw;
11     }
12 
13     function invest() payable {
14         if (msg.value<jackpot/2) throw;
15         lastInvestor = msg.sender;
16         jackpot += msg.value/2;
17         lastInvestmentTimestamp = block.timestamp;
18     }
19 
20     function resetInvestment() {
21         if (block.timestamp < lastInvestmentTimestamp+ONE_MINUTE) throw;
22 
23         lastInvestor.send(jackpot);
24         owner.send(this.balance-1 ether);
25 
26         lastInvestor = 0;
27         jackpot = 1 ether;
28         lastInvestmentTimestamp = 0;
29     }
30 }
```

Listing 2.4: The `GovernMental` contract [3].

In **Attack #1**, the adversary is the `owner` of the contract. She abuses the *Call-Stack Depth* and the *Exception Disorder* vulnerabilities by publishing contract `Mallory3` (cf. Listing 2.5).
Normally, the `lastInvestor` can request his asset when no one contributes to the contract for at least a minute. However, the attacker can pre-empt him and instead call `Mallory3`'s `attack` function (see line 2). This function invokes itself recursively until the call-stack depth reaches 1022 (see line 3). At depth 1023, it calls `GovernMental`'s `resetInvestment` function, which then takes the last slot on the call-stack. Now, `resetInvestment` tries to send the `jackpot` to the `lastInvestor` and fails (see line 23). Since it does not check for `send`'s return value, it still resets the `jackpot` and the `lastInvestor` (see lines 26-28). At another payout, the attacker can omit the attack and receive the money of the stolen `jackpot` [3].

```
1 contract Mallory3 {
2     function attack(address target, uint count) {
3         if (count<=1022) this.attack(target, count+1);
4 
5         else GovernMental(target).resetInvestment();
6     }
7 }
```

Listing 2.5: The `Mallory3` contract [3].

In **Attack #2**, the attacker is a miner and a player at the same time. Whenever she receives transactions relevant for the `GovernMental` game, she does not include them directly into the block but rather withholds them. After a while, she invests into the contract by her own until the `jackpot` is at least two times higher than any investment of

the transactions she has received before. Next, she appends the other transactions to the block which consequently fail to invest in the game. Finally, she submits the block to the network. With luck enough time passes and she wins the `jackpot` with all the investments of the transactions she has mined. Because of this scenario, the `GovernMental` contract suffers from a *TOD* vulnerability [3].
In an alternative scheme, the miner could also just block any transaction investing in the `GovernMental` contract [3].

3. Related Work

We have now seen how many security issues smart contracts can have and how much a programmer must keep in mind to avoid them. Fortunately, a variety of security analysing tools can be used to test a smart contract. In this chapter, we present fifteen tools relevant for our thesis and explain the approaches they apply. Relevant, in this case, means tools we considered to integrate into our testbed. Consequently, we only introduce tools which are freely available. In Section 3.1, we start with tools implementing symbolic execution. Later, in Section 3.2, we continue with tools using an intermediate representation (IR) and finally, in Section 3.3, present tools applying the fuzzing technique.
Table 3.1 gives an overview of the tools and their analysing strategies. Additionally, it shows when a tool searches for one of the vulnerabilities introduced in 2.6. Many security issues are also present in the Smart Weakness Classification (SWC) Registry, which tries to establish a standard for categorizing smart contract vulnerabilities [42]. When a security issue is listed in the registry, we also provide their SWC-ID.

3.1 Symbolic Execution

Seven of the introduced tools use symbolic execution to generate a control flow graph (CFG). The nodes of the graph represent basic code blocks which do not contain any `JUMP` instructions. The directed edges represent `JUMP`s between these blocks. Usually they are labeled with the conditions needed for the corresponding `JUMP`. To determine the edges, the tools execute the code symbolically. Instead of assigning distinct values to a variable, like $x = 5$, a symbolic execution always assigns ranges, like $0 < x < 10$. Therefore, it can evaluate every execution path which is not infinitely long [12].
After the generation of the CFG, the tools search for critical program states, discovered by certain patterns. When they reach a critical state, they usually check whether the input parameters leading to this path can be manipulated by an arbitrary user or the mining node. If so, they flag a vulnerability [8, 11, 12].
Unfortunately, the number of possible paths often increases exponentially with the program length. Additionally, some paths might have an unbound length. Therefore it is usually infeasible and sometimes even impossible to execute every path [8].

For this reason, some tools limit the number of loop iterations [8], the call depth [8] or their execution time [12, 11]. Consequently, they cannot find every security vulnerability. To avoid false positives, Maian [8], Manticore [9] and Mythril [10] all validate vulnerabilities

	Analyse methods			Checked security issues								
				Solidity			EVM			Blockchain		
Tool ↓	Symbolic Execution	Intermediate representation	Fuzzing	Reentrancy	Over- or Underflow	Unchecked Call	Call-Stack Depth	Locked Ether	Unprotected Selfdestruct	TOD	Timestamp	Randomness
SWC-ID →				107	101	104	-	-	106	114	116	120
Maian [8]	✓	✗	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗
Manticore [9]	✓	✗	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓
Mythril [10]	✓	✗	✗	✓	✓	✓	✓	✗	✗	✓	✓	✓
Osiris [11]	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗
Oyente [12]	✓	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗
Securify 2.0 [13]	✓	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✓
Vandal [15]	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗
NeuCheck [43]	✗	✓	✗	✓	✓	✓	✗	✓	✗	✗	✓	✓
Securify [44]	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓	✓	✗
Slither [45]	✗	✓	✗	✓	✗	✓	✗	✓	✓	✗	✓	✗
SmartCheck [14]	✗	✓	✗	✓	✓	✓	✗	✓	✓	✗	✓	✗
ContractFuzzer [46]	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗
Echidna [47]	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
ILF [48]	✗*	✗	✓	✗	✗	✓	✗	✓	✓	✗	✓	✗
sFuzz [49]	✗	✗	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓

Table 3.1: An overview of some smart contract analyzing tools and the security issues they test for [4, 46, 13, 42, 11, 50, 51, 48, 49, 43, 45, 52].
*: ILF only analyses the contracts of its training set with symbolic execution.

dynamically on a private blockchain. They first deploy the contract on it and then try to attack it. To do so, they create a transaction with concrete input values derived from the symbolic input values. Only if the security issue results in an actual exploit, it is reported to the user. Since some contracts can only be attacked with multiple transactions, Mythril also enables the possibility to verify a weakness with several transactions. However, since the execution time can increase exponentially with the number of transactions tested per vulnerability, it only tries a single one by default [10].
Other tools using symbolic execution are Oyente [12], Osiris [11], Securify 2.0 [13] and Vandal [15].

3.2 Intermediate Representation

Four of the described tools represent the EVM bytecode or Solidity source code abstractly in an IR before they start their analysis. This has several benefits: It is often easier to inspect the IR than the original programming language, especially, when software to analyse specifications of the IR already exist [14]. Furthermore, when someone wants to test source code of different programming languages, he does not need to write several analyzers. Instead, he can translate each language in the IR and then only analyse the IR [53].

Vandal [15] decompiles the EVM bytecode into an IR, executes it symbolically and then abstracts the CFG into another IR, consisting of Horn clauses [54]. Soufflé [55], the program which can be used to solve constrains written using declarative, logic programming language, finally checks the contract code against violation of compliance patterns written in DataLog [54].
SmartCheck [14] and NeuCheck [43] translate the contract code into an Extensible Markup Language (XML) syntax tree and find potential security issues by traversing this tree.
Slither [45] decompiles the EVM bytecode into a single static assignment (SSA) IR. In this language, every variable can only be assigned once [56]. In the next step, it runs several detectors on the SSA to discover potential security vulnerabilities.
Securify [44] also decompiles the EVM bytecode into an SSA IR. It then acquires facts from the IR and checks them against violation or compliance patterns. Both, the facts and the patterns are written in DataLog.
Its successor, Securify 2.0 [13], translates Solidity code into an IR [53, 57] and afterwards analyses the IR with the help of symbolic execution.

3.3 Fuzzing

A fuzzer tests a program iteratively. On each iteration, it produces input values and tests the program with these values. It then checks whether the program crashes or meets certain program states. If this is the case, it reports the problematic input values to the user [46]. For a smart contract fuzzer, these input values are usually transactions calling a function on the tested contract [46, 47, 49, 48]. Choosing these transactions uniformly by chance is mostly ineffective, since it often results in a low code coverage [48]. For this reason, the four fuzzers we introduce leverage sophisticated policies to generate more effective transactions.

ContractFuzzer [46], e.g., invokes each contract function with multiple calls. Each time, it generates random function parameters in a nonuniform way. For fixed-sized types, like `int`, it prefers values which are frequently used on the Ethereum blockchain. For non-fixed-size types, like `string`, it randomly chooses a value as a length and then randomly generates a value for that type.

Imitation Learning based Fuzzer (ILF) [48], on the other hand, uses a neural network to create more effective transactions. The network was trained on a data set of more than 18,000 smart contracts. Each contract was labeled with a set of transactions which guarantee a high code coverage. This set was retrieved by analysing the contract with symbolic execution.

On each iteration, sFuzz [49] executes a set of transaction sequences. In the first iteration, these transactions are random function calls to the contract. At the end of an iteration, the set is modified. For one, it is reduced to sequences which have either discovered new branches in the contract's CFG or were close to do so. This closeness is calculated by a distance function. Now the remaining sequences still mutate. First, the set is grouped into pairs of sequences. Both sequences in a pair are then split at the same position. Next, their second parts are swapped to create new sequences. Finally, the contract is tested with the new set.
This procedure of executing the set, reducing it and mutating it is repeated until a timeout is met. Finally, transaction sequences which caused a program state associated with a vulnerability are reported to the user.

To test a contract with Echidna [47], a user must first write property checks into his contract. The tool then tests a contract in two steps. In the first one, it analyses a contract with Slither to leverage "useful constants and functions that handle Ether [...] directly"

[47]. In the second step, it repeatedly generates new function-calling transactions and tests the contract with them. These transactions are generated based on the information gained in step one as well as observations of the execution of previously tested transactions. If any of the specified properties does not hold for a transaction, Echidna reports it to the user.

4. The Architecture of the Testbed

Existing analysing tools each cover a different subset of smart contract security issues. To still test for a wide spectrum of vulnerabilities, a user must analyse his contract with a variety of tools. The motivation of our thesis is to simplify this task by presenting a testbed which combines a number of these tools.

On the testbed, a user can test a smart contract with the built-in tools. To test a bytecode contract, he only needs to specify the file containing the contract and select the tools he wants to use. Solidity files, on the other hand, can contain multiple contracts. Therefore, when he tests such a file, he can additionally provide the name of the contract. Otherwise, the testbed simply takes the first contract in the file. In the next step, it runs the chosen tools in the background. As soon as a tool has terminated, the user receives a report of the tool's findings. At the end, when none of the tools are running anymore, he additionally gets an overview of the overall testing process.
In Section 4.1, we first define important design criteria our testbed has to fulfil. Afterwards, in Section 4.2, we give an overview of the testbed's architecture. In Section 4.3, we subsequently demonstrate a simple user scenario to clarify how the different components of the architecture interact. At the end, in Section 4.4, we still show how the testbed satisfies the previously evaluated design criteria.

4.1 Requirement Analyses

Usability. One of the main motivations for developing the testbed is the often complicated execution of existing tools. Therefore, we must ensure that our testbed accomplishes a high level of usability.

Compatibility and Simple Deployment. Furthermore, a complicated installation process is a high initial hurdle and will prevent many users from using our testbed. Thus, our program should be easy to deploy and compatible with other software.

Efficiency. Many smart contract analysing tools are very time-consumptive. Since our testbed can run multiple tools at once, it is of upmost importance to execute them as efficient as possible. Otherwise, a user would only test his contract with a few of the embedded tools. Consequently, our testbed would practically loose one of its key benefits, which is analysing a contract with multiple tools at once.

Reliability. Unfortunately, it is not uncommon that a tool crashes during a test run. Thus, to provide a reliable software, it is substantial to prevent such errors from propagating to other parts of the testbed.

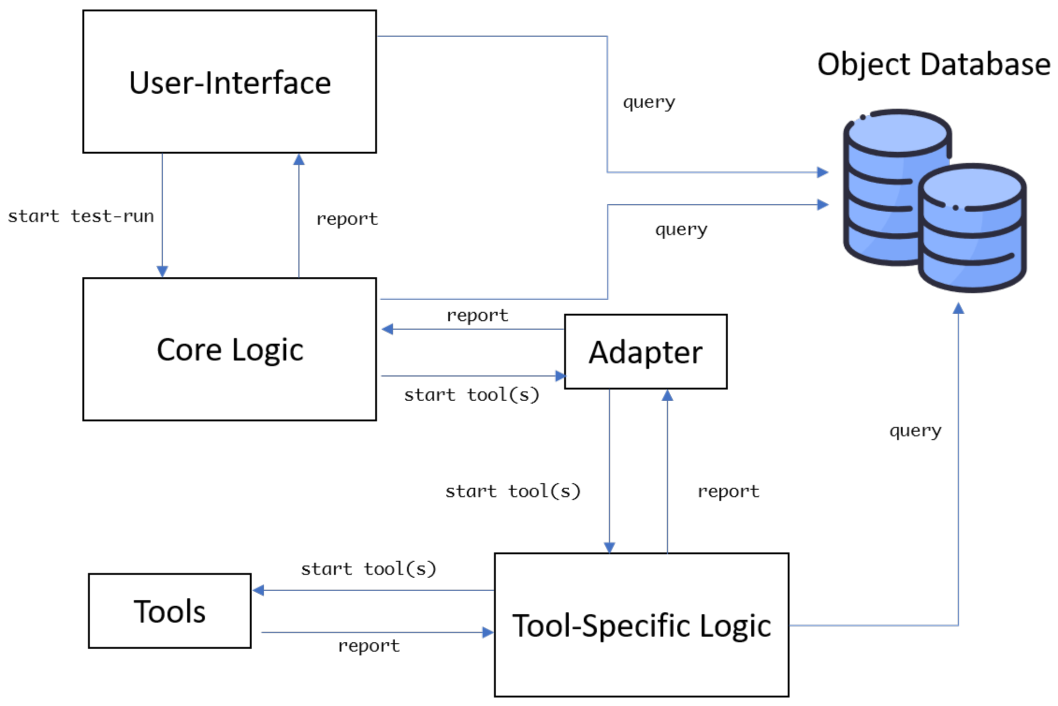


Figure 4.1: The components of the testbed's architecture. The user interface component stands for both interfaces, the *Command Line Interface* and the *Web Interface*.

Maintainability. Keeping a software up to date is especially important for a program testing for security issues. Attackers will always find new ways to exploit a contract and therefore new tools covering these weaknesses have to be integrated into the testbed. Also, some tools become obsolete after a while. For these reasons, it should be easy to add or remove tools to or from the testbed. To keep our testbed relevant, it must therefore be maintainable.

4.2 Architectural Design

The testbed consists of five major components: The two user interfaces, namely the *Command Line Interface* and the *Web Interface*. Furthermore the *Core Logic*, the *Tool-Specific Logic* and the *Object Database*. An illustration of them and how they interact can be seen in Figure 4.1.

The Command Line Interface and the Web Interface. Depending on how the user accesses the testbed, either the command line interface or the web interface manages the interaction between user and testbed. It passes the user's requests forward to the core logic and is in charge of displaying the core logic's results.

The Core Logic. The core logic administers the test run. It delegates the execution of each tool to the tool-specific logic and summarizes the results of the different tools. The summary is then, along with a report for each individual tool, forwarded to the user interface.

The Tool-Specific Logic. The tool-specific logic executes the different tools and analyses their results. It furthermore provides an adapter, so that the core logic has a unique way to interact with it.

The Object Database. All the various components use the object database. It abstracts an underlying relational database, which, e.g., provides information about the

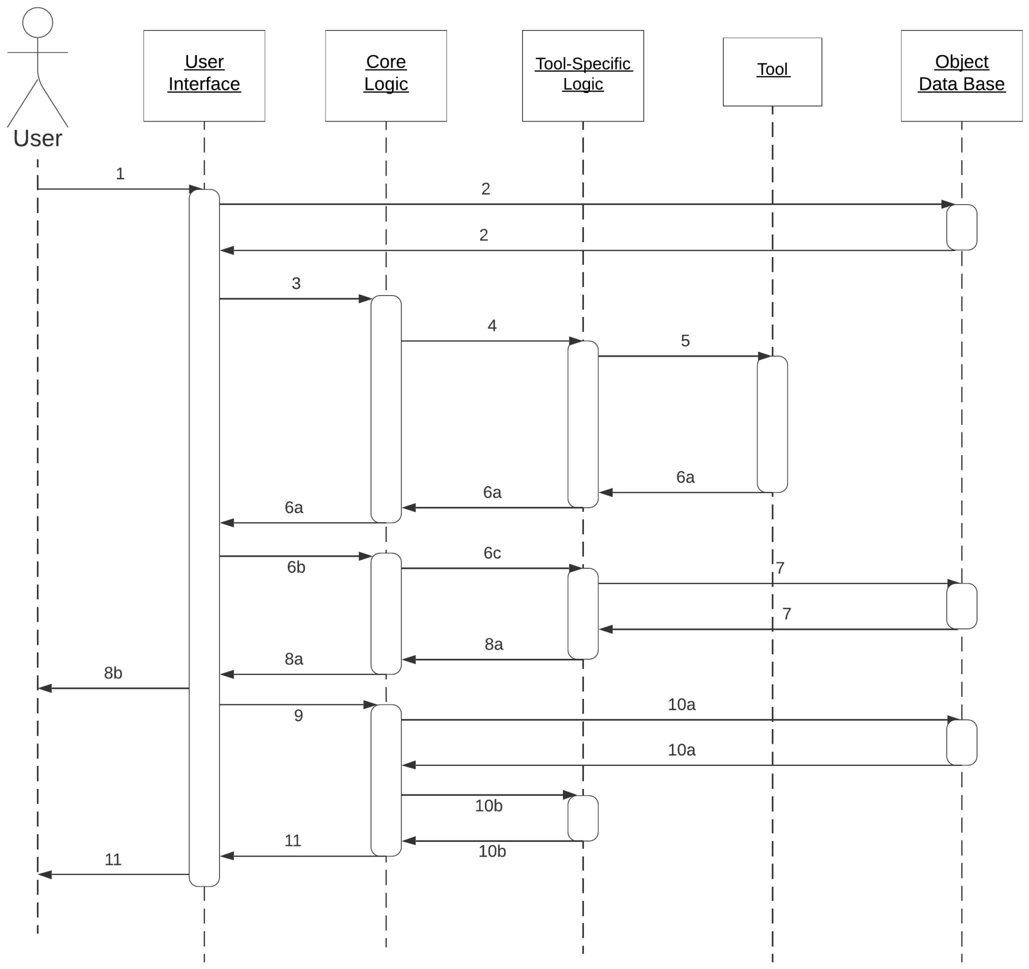


Figure 4.2: A sample test run with a single tool. For simplicity, we do not display the adapter, since it merely serves as a plain interface.

tools. Additionally, it defines classes which can be used to easily store and pass data of important concepts like a smart contract.

4.3 A Sample User Scenario

In Figure 4.2, we demonstrate a sample user scenario. It shows the major steps from the beginning until the end of a test run with the command line interface:

1. On the interface, the user uploads his contract and selects the tools he wants to test with. In our scenario, he only picks a single tool.
2. Subsequently, the interface queries the object database for information about the selected tools.
3. In the next step, the interface passes the user request and the information about the tools to the core logic.
4. Now, the core logic does not directly communicate with the tool-specific code. Instead, every tool is operated by a sub-component of the tool-specific logic. Thus, the core logic tells the appropriate sub-component to test the contract with the selected tool. To unify the communication between the core logic and the various sub-components, each sub-component uses the same adapter. Since the adapter is

merely a plain interface, we do not display it in Figure 4.2. This way, we keep the illustration clear and simple.

5. Immediately, the sub-component runs the tool.
6. As soon as the tool has terminated (step 6a), the user interface asks the core logic for a report file of the tool (step 6b). The core logic then passes the request to the sub-component (step 6c).
7. The sub-component now inquires the database for information on how to recognize the discovered security issues in the tool's output.
8. In the next step, it returns the report file containing the detected security issues and other information about the testing process (step 8a). The user interface then prints the path to the file on the command line (step 8b).
9. If none of the tools are running anymore, the user interface additionally asks for a final report with an overview of all of the tools' findings. Apparently, this is the case and therefore, it requests an overview from the core logic.
10. Now, the core logic creates the overview with help of the object database (step 10a) and the sub-component (step 10b).
11. At the end, the testbed writes the final report on the command line.

Testing a contract with the web interface is quite similar except that the user must manually request for updates about the testing process.

4.4 Design Criteria

In Chapter 1, we have already described how our testbed meets the important design criteria usability. In this section, we clarify how its architecture also meets the other criteria defined in Section 4.1.

Compatibility and Easy Deployment. The testbed is installed on a virtual machine. Consequently, it can easily be deployed and does not interfere with other software.

Efficiency. Furthermore, when testing a contract, the testbed speeds up the execution by running every tool on a different process. This does not only cause the tools to run in parallel but also allows the operating system to use multiple cores. Accordingly, the available resources are used as efficiently as possible.

Reliability. To avoid that an exception thrown by a tool impacts other parts of the testbed, each tool is executed in an own Bash shell. This way, only the appropriate shell crashes while the execution of the rest of the testbed is unaffected.

Maintainability. With the separation of the tool-specific logic from the core logic, our testbed makes it very easy to integrate new tools in the testbed or remove old ones. To add a tool, the administrator of the testbed only has to install it, implement a simple class and type in a single command on the command line. Removing a tool only means to uninstall the tool and execute the appropriate command.

5. Implementation

In this chapter, we describe the implementation of the testbed. Section 5.1 starts with the setup of the virtual machine including the prerequisites needed to run the testbed. Next, we explain the five core components already mentioned in 4.2: The Object Database in Section 5.2, the Core- and Tool-Specific Logic in Section 5.3 and 5.4 and finally, in Section 5.5 and 5.6, the Web Interface and the Command Line Interface.

5.1 Setup

The testbed is installed on a VM running Ubuntu 20.04 Long-Term-Support (LTS). We created the VM with the Oracle VM VirtualBox Manager and use a VirtualBox Disk Image as its virtual hard drive. The testbed itself is written in Python 3.8 and embedded in a Python virtual environment, along with all its package dependencies.

The Integrated Tools. From the tools introduced in Chapter 3, we installed eight of them on our VM and integrated them into our testbed. Together, they cover all of the vulnerabilities outlined in Section 2.6. The tools are Maian [8], Manticore [9], Mythril [10], Osiris [11], Oyente [12], SmartCheck [14], Securify 2.0 [13] and Vandal [15].
We did not consider the other tools for various reasons. Some of them [47, 49, 48], e.g., require the users to provide more information than the contract and its name [58, 59, 60]. This would prevent us from keeping the testbed simple. Others were deprecated [44, 61], did not provide installation guidelines [43, 62] or were only a reduced version of a commercial tool [45]. We also did not implement tools which require to implement the communication to a private blockchain [46, 63], since this is beyond the scope of our thesis.
The embedded tools all provide a command line interface to interact with them. Table 5.1 gives an overview of them. Almost every tool can process Solidity source code directly. Only for Vandal, the testbed must first compile it. Maian [8], Mythril [10], Osiris [11] and Oyente [12] all internally analyse bytecode and thus provide an option to pass a bytecode file to them. Manticore does the same, but unfortunately only accepts Solidity files [64]. SmartCheck [14] and Securify 2.0 [13] analyse the Solidity source code directly.
By default, Osiris, Oyente, Securify 2.0 and SmartCheck analyse every contract contained in a Solidity file. Securify 2.0 also allows to define a single contract to analyse. Our testbed always chooses the latter option, since Securify 2.0 can be very time consuming and a user might only want to test a single contract in a file. The other tools also only analyse the specified contract.
Manticore returns its report in a simple text file, while Vandal hands over a file in the

Tool	Virtual Environment	Pre-installed Solidity Compiler Version	Analysis of Multiple Contracts	Input Types		Output Format	Version/ Commit
				Solidity	Byte-Code		
Maian	Docker Container	`0.4.20`	✗	✓	✓	Command Line	`latest` [69]
Manticore	Docker Container	`0.4.25`	✗	✓	✗	Text File	`0.3.4` [76]
Mythril	Docker Container	-	✗	✓	✓	Command Line	`0.22.4` [77]
Osiris	Docker Container	`0.4.21`	✓	✓	✓	Command Line	`latest` [78]
Oyente	Docker Container	`0.4.21`	✓	✓	✓	Command Line	`latest` [79]
Securify 2.0	Docker Container	`0.5.12`	✗	✓	✗	Command Line	`71c22dd3d6` [13]
SmartCheck	Docker Container	-	✓	✓	✗	Command Line	`2.0.1` [80]
Vandal	Python Virtual Environment	-	✗	✗	✓	JSON	`d2b004326f` [70]

Table 5.1: The tools integrated in the testbed.

JacaScript Object Notation (JSON). The remaining tools print their findings directly to the command line.

The Installation of the Tools. Installing the tools directly on the system is quite difficult and erroneous, especially since the various tools often have conflicting dependencies. Therefore, most of them are installed in a virtual environment.
For some tools, a pre-built Docker image with the tool and all its dependencies is installed, which is available on `docker.com`. Integrating these tools simply means installing Docker and pulling the relevant Docker images. The providers of these images are either the creators of the tools themselves, like in the case of Manticore [65], Mythril [66], Osiris [67] and Oyente [68] or by a third party, like in the case of Maian [69]. The authors of Securify 2.0, however, only maintain a Dockerfile [13], which can be used to build a Docker image with a single command.
Vandal, which is written in Python, can be easily integrated into a Python virtual environment. To install it, we cloned its Git repository, created the environment and used Python's package manager Pip to install the dependencies specified in the `requirements.txt` file. Additionally, Vandal needs Soufflé [70], which can be installed by copying some commands from Soufflé's homepage [71].
SmartCheck is deployed with the Node Package Manager (NPM) which simplifies the installation to a single command [72]. It is the only tool on the testbed, which is not deployed in a virtual environment. However, to obviate conflicts with tools integrated in the future, we also embed it inside a docker container.

The Solidity Compilers. Except for SmartCheck, all the tools running on a Docker container need a Solidity compiler. Otherwise, they cannot work with Solidity source code. Even Securify 2.0 needs one to get the abstract syntax tree (AST) of the contract [73]. Mythril can automatically download the compiler version it needs for a given contract [66]. The other containers have a certain version pre-installed. In these cases, our testbed prefers to use this version. The tools are probably well tested with it and might break with others. However, if a contract is not compatible with the favored version, our testbed must pass the required compiler to the container. For these cases, we installed multiple Solidity compiler versions on our testbed. We installed each version from `0.4.11` up until the latest version, `0.7.4` [74]. We did not include prior Solidity compiler versions since they are rarely used anymore [75].

5.2 The Object Database

On the testbed, complex data, consisting of more than a single attribute, is represented as an object. Some of these objects must be available in several sessions and therefore be stored persistently.
A state-of-the-art way to store data are relational databases. However, these databases cannot manage instances of custom objects but only define basic data types. For this reason, we use the object-relational mapping (ORM) pattern [81] to translate between objects and table rows. It abstracts an underlying relational database to an object database. Every object of a class implementing the pattern can be tracked by the object database. Changes on this object can then easily be reflected on the relational database.
On our testbed, we realize ORM with the SQAlchemy library. In this section, we explain important components of the library and introduce the classes of the testbed which implement this pattern.

5.2.1 SQLAlchemy

By using SQLAlchemy, we can simplify our code and separate the business logic from the data layer. In our case, SQLAlchemy uses SQLite [82] as the underlying database.

Defining Classes for the Object Database. Classes which implement the ORM pattern are usually represented by a table in the relational database. Each table row stores the attributes of an instance. A primitive attribute can directly be recorded on the database. On the other hand, an attribute holding a reference to another object is stored as a single or several foreign keys. These keys then link to the table row representing the referenced object.
One way to tell SQLAlchemy how to manage instances of a class is to let it inherit from the class returned by the `sqlalchemy.ext.declarative.declarative_base` factory function. The name of the relational database table is then set in the class attribute `__tablename__` while each column is defined by a class attribute holding an `sqlalchemy.schema.Column` object. In the `Column`'s constructor, we can determine properties like its type, whether it can be set to `None` or if it is a part of the primary or a foreign key [81].

Defining Many-To-Many Relationships. A many-to-many relationship can either be defined in an extra `sqlalchemy.Table` object or another class. Both approaches must define the foreign keys of the tables they connect. A relationship holding additional properties can only be specified in a class [83].
However, establishing such a relationship only with foreign key columns does not reflect on the objects. Though a user could find out the relationship between two objects manually by matching the appropriate foreign and primary key, the objects themselves do not hold a reference to each other. Such behaviour must be implemented explicitly by defining a `sqlalchemy.orm.RelationshipProperty` on one of the related classes [83].

Defining Inheritance. Inheritance can be, among other approaches, realized as "joined table inheritance". In this case, one table stores the parent's attributes while another one stores the child's attributes [84].
To implement it, the parent class must specify another `Column` which tells SQLAlchemy the class type of a particular row. Moreover, a foreign key in the child links to the corresponding row in the table of the base class. Both classes also have to declare a `__mapper_args__` attribute. It tells SQLAlchemy, how it should name the two class types and, for the base class, it also defines the name of the `Column` which stores the class type [84].

Querying the Database. The database can then be queried or updated in an object-oriented way with a `sqlalchemy.orm.session.Session` instance. Internally, SQLAlchemy translates the query to an SQL statement and then loads the retrieved data into the appropriate

objects [81]. It never calls a constructor and therefore, attributes which are not loaded from the database must be created in a method decorated with the `@reconstructor` instruction [85].
The `Session` also tracks all the objects associated with it. These are, for example, objects returned by a `Session`'s query or newly created objects added to the `Session`. Each time, the testbed commits a `Session`, the states of its associated objects are written to the underlying database [81].

5.2.2 The Tool Class

The `Tool` class represents a security analysing tool. It provides a `Column` for the name of the tools, one for the scripts containing the tool-specific logic and another one which tells whether the tool tests multiple contracts in a Solidity file or just a single one. For the tools deployed with a Solidity compiler, we must furthermore store the version of the compiler. Our testbed needs this information to know when it has to force a tool to use a different one than the pre-installed version (cf. Section 5.1).

5.2.3 The SecurityIssue and ToolSecurityIssue Class

For the overview of the tools' results, we need to know which security issue each tool looks for and how it prints an issue to its output. To find this information, we analysed their documentation [8, 86, 50, 13, 87], source code [88, 89] and command line output [12, 8]. Unfortunately, the tools do not use a uniform nomenclature for the security issues. Therefore, we categorized the vulnerabilities according to the SWC-Registry. If a security issue is not listed in it, we simply adopted the name one of the tools searching for it uses.

The `SecurityIssue` class represents a security issue. It stores the title of the vulnerability, a description, a link for further information and their SWC-ID, if available. For the latter vulnerabilities, we take the descriptions and links of the registry. Otherwise, we use the explanation of one of the tools and reference to the corresponding web page.
The `ToolSecurityIssue` class implements the many-to-many relationship between a `SecurityIssue` and a `Tool`. It stores their foreign keys and has a `Column` for an identifier. This identifier holds a string which recognizes the security issue in the tool's output. Since some issues can be found with different identifiers, the `ToolSecurityIssue` class allows several instances defining the same relationship.

5.2.4 The Error and ToolError Class

Whenever an error occurs during the testing process of a tool, we want to display it to the user and supply an explanation. Our testbed detects such errors in three ways:

1. Some errors are straightforwardly detected by the `ToolTestRun` subclass which interacts with the tool.
2. Other errors are directly raised by a tool and printed to its command line output. We diagnose these errors with regular expressions which we retrieved by heavily testing the tools.
3. Vandal also writes some errors directly into a JSON dictionary contained in its report file [90]. In this case we simply load them from the dictionary. A dictionary is a mapping between keys and values.

Each error is represented by an instance of the `Error` class. Similarly to a `SecurityIssue`, it stores a title, a description and a link. We have created the description by ourselves and, if helpful reference to a web page with more information about the error.
The `ToolError` class is identical to the `ToolSecurityIssue` class, except that it connects a `Tool` with an `Error` instead of a `SecurityIssue`.

5.2.5 The Contract and SolidityContract Class

In a normal test run, a contract is not stored on the testbed. However, the contracts used in the evaluation are kept permanently, and therefore the classes representing them implement the ORM pattern.

A bytecode contract is represented by the `Contract` class. Its most important attribute is the path to the contract file. Because some contracts are used in the evaluation, the class additionally has a `Column` for the contract's Ethereum address, one for a reference to the source where it was downloaded from and another one for the size of the contract's bytecode.

The `SolidityContract` class represents a Solidity contract. It extends `Contract` with further attributes and some methods. The underlying database realizes the inheritance with joined tables.
One of `SolidityContract`'s additional attributes is the name of the contract to test. If it is not given, the constructor automatically sets it to the name of the first contract in the Solidity file. Furthermore, `solc_from` and `solc_to` define the minimal and maximal compiler version which can be used to compile the contract file. The constructor automatically finds the range by scanning this file. Some files define a separate `pragma` directive for each contract (cf. Section 2.5). In their cases, we find the version range which satisfies all of the directives and raise an exception if they do not share a common scope.
If the `get_byte_code_file` method is called for the first time, it compiles the contract defined by the `name` attribute and then returns the path to the bytecode file. On the next request, it simply returns the path to the already compiled file. However, this path is not kept in the database, instead the file is removed on the object's destruction.
Since several tools might call the method simultaneously, it always requires a lock at the beginning of its execution. Naturally, the lock must always be available, no matter how the `SolidityContract` instance is created. For this reason, it is constructed in a method decorated with the `@reconstructor` instruction, which is then called by the constructor or SQLAlchemy.

5.3 The Core Logic

The Core Logic is completely contained in the `TestRun` class. Every instance of the class represents a test run with a contract and the tools the user has selected. It is additionally associated with the appropriate tool-specific subclasses of the `ToolTestRun` class, which contain the code to interact with the tools.
Its `run` method starts the test run, while the `get_security_issues_statuses` and `get_errors_statuses` methods summarize the output of the tools. Additionally, the `get_tool_test_run` method returns the requested `ToolTestRun`. Furthermore, the `get_status` method gives the current status of the tool, which is either *Before Run*, *Running* or *Terminated*.

5.3.1 The Run Method

The `run` method can only be called once; on further attempts it throws an exception. If it is invoked for the first time, it creates the corresponding `ToolTestRun` subclass instances for the selected tools. Next, it calls their `run` method to start the analysis of the tools.
In Python, classes and modules are normally imported at the beginning of a script [91]. However, to keep tool-specific code out of the core logic, the `TestRun` class imports the `ToolTestRun` subclasses dynamically. Each subclass is inside an own module which offers a `create_tool_test_run` function to retrieve a new instance of the subclass. The `TestRun` class identifies the module by the path contained in the `script` attribute of the corresponding tool instance.

5.3.2 Retrieving the Status of the Security Issues and Errors

The `get_security_issues_statuses` method essentially returns a table with the tools and the statuses of the `SecurityIssues` they look for. The table is returned as a dictionary of dictionaries and every status can be one of the following:

- *Loading*, if the tool is still running.
- *Found*, whenever the tool has found the `SecurityIssue`,
- *Not Found* otherwise and
- *Not Checked*, in the case that the tool does not look for the `SecurityIssue` at all.

At the beginning, the method fetches a list of `SecurityIssues` from the object database. It only queries for issues at least one of the selected tools looks for and demands them to be primary sorted by their SWC-IDs and secondary by their titles.
In the next step, a dictionary is created. Now, for every one of the queried `SecurityIssues`, a key-value pair is added, where the key is the `SecurityIssue` and the value another dictionary. The value-dictionary, in turn, has the selected tools as its keys and the status of the corresponding `SecurityIssue` as its value.
To check whether a tool has terminated, the `get_terminated` method of the relevant `ToolTestRun` subclass is called. The class also offers a `get_security_issues` method, which is called to return the discovered `SecurityIssues` of the terminated tools.
Finally, the testbed sorts the `SecurityIssues` in the main dictionary by the count of their occurrences, with the most frequent ones first. If two `SecurityIssues` were found by the same number of tools, the testbed keeps their original order. Since they were already presorted by the database, they are now sorted by the following parameters, with descending priority:

1. The number of tools which have discovered the issue.

2. The SWC-ID, starting at the lowest ID, with the `None`-values coming last.
3. The issues' `title`s.

The order is very important since the functions creating the summaries of the discovered security issues all keep this order. Thereby, the user sees the most relevant security issues first.

The `get_error_statuses` method does basically the same, only that it reports the statuses of the `Error`s and uses the subclasses' `get_errors` method rather than the `get_security_issues` method. Naturally, it also does not sort for SWC-IDs, since `Error`s have none.

5.4 The Tool-Specific Logic

The code to execute a tool and analyse its output must be contained in a custom subclass of the `ToolTestRun` class. Each instance of the subclass represents the test run of a contract with the associated tool. It is responsible for executing the tool and analysing the tool's output.

5.4.1 The ToolTestRun Class

Requiring the subclasses to derive from the common `ToolTestRun` base class has several advantages:

- For one, the base class automatically serves as an interface so that other modules have a unified way to communicate with the different subclasses. This enables the testbed to easily integrate new tools.
- Furthermore, it eliminates the necessity of duplicate code.
- Moreover, it guarantees certain properties of the subclasses. For example, many tasks are split into two methods. A public one implements the common part of the code and then delegates the tool-specific part to a protected, abstract method.
  This also has a benefit for administrators, who want to integrate a new tool into the testbed. They need less knowledge to implement the code for their tool and at the same time, have fewer opportunities to make mistakes.

Starting the Tool. The execution of the tool is started in the `run` method. First, it checks whether it has already been called and raises an error if this is the case. Otherwise, it starts a new thread, which in turn executes the `__run` method. The two methods must be split because they run on different threads. The `__run` method then executes the abstract `_execute` method and stores the execution time of the `_execute` method. The constructor of the `ToolTestRun` class also allows to apply a timeout. If it is set and exceeds, the `_execute` method should throw an exception, which the `__run` method catches. The `Error` corresponding to the exception is then appended to a set, which is held by a variable named `_exceptions`. Thus, the subclass can include it in its report with the other `Error`s returned by `get_errors`. Finally, `__run` sets the status of the `ToolTestRun` to *Terminated.*
Two leading underscores in a method- or variable-name mean that it is private, a single underscore indicates protection [92].

Get Methods. The `get_execution_time`-, `get_security_issues`-, `get_errors`- and `get_report` -methods all raise an error if they are requested before the tool has terminated. If not, `get_execution_time` returns the time period measured in the `__run` method. On their first invocation, the other methods still need to create the requested information by calling the relevant abstract methods implemented in the subclasses. The `get_report` method then prints the information to a file, while the other methods store it to a variable. Consequently, they can answer directly in potential future requests.
Finally, the `get_security_issues` method returns the `SecurityIssues` the tool has found while the `get_errors` method returns the `Errors` raised during the execution of the tool, including the ones of the `_exceptions` set. The `get_report` method provides the path to the report file which summarizes the testing process of the tool. The `get_terminated` method simply specifies whether the tool has already terminated or not.

Helper Methods. Some tasks are similar for many subclasses but not all of them. For this purpose, the `ToolTestRun` class provides several helper methods, which simplifies programming the subclasses.
One of them, e.g., is the `get_solc_bin` file. If the `ToolTestRun`'s contract is in Solidity, it returns the path to the minimal installed Solidity compiler which is allowed to compile the

contract.
The other methods only carry out trivial tasks, therefore we do not always explicitly mention their usage when we describe the subclasses.

5.4.2 The Execution of the Tools

The execution of the tools is managed in the `_execute` method, which is implemented very similar across all of the subclasses. First, they create the commands to run the tool. If the tool is embedded in a virtual environment, this includes the setup of the environment. Afterwards, they run the tool in an own process and capture its command line output to a file, so that the other methods can use it.
Only two of the tools have to slightly modify their testing process. Vandal's subclass must compile Solidity files first. Maian can test for three unique weaknesses, yet it can only test for one at a time [93]. Because of this circumstance, its subclass starts the tool for every option in an own process, resulting in three separate output files. After starting the processes, it still waits for them to finish.

Setting up the Virtual Environments. Before starting the embedded tools, we must still activate their environment. For Vandal, this plainly means executing the `activate` script of its Python virtual environment. The process for the tools integrated in a Docker image requires more knowledge. In their cases, we must first built their container. The tool can then be started in the same command:

```
docker run <Docker Command Options> <Docker Image> <Command>
```

`docker run` tells Docker to start a container from the image specified in *Docker Image* [94].

The *Docker Command Options* are:

- `--rm`, which tells Docker to remove the container as soon as it has terminated [94].
- The mounting of the contract's base directory to the container path where the tool expects the contract to be.
- The mounting of the directory containing the Solidity compiler the tool should use, unless the appropriate tool is Mythril, SmartCheck or the contract is given in byte-code. Since most tools already have a Solidity version pre-installed, we do not always have to specify this directory. However, it simplifies the code and does not change the outcome, thus we do so anyways.
- For Manticore, a temporary directory, where it can create the file with its findings, is needed.
- Sometimes, Docker has to start in another working directory than the pre-defined one. In these cases, we set `-w <path to the working directory>`.
- The authors of Manticore also recommend to increase the stack size the tool can use, hence `--ulimit stack=100000000:100000000` is passed to Docker [65].

Docker Image is the name of the Docker image which has the tool installed [94].

Command is the command to start the tool [94].

Forcing a Certain Solidity Compiler. When a Solidity contract cannot be compiled with the tool's default compiler, we must tell it to use a different one. Mythril and Securify 2.0 have a built-in option to do this. SmartCheck does not need one. For the other tools, we must change the container's `PATH` variable before we run the tool. Unfortunately, Docker only allows to execute a single command [94]. Luckily, we can bypass this restriction by calling the Bash shell and passing the required instructions to it [95].

The Commands to Execute the Tools. The commands to run the tool are built with the following arguments:

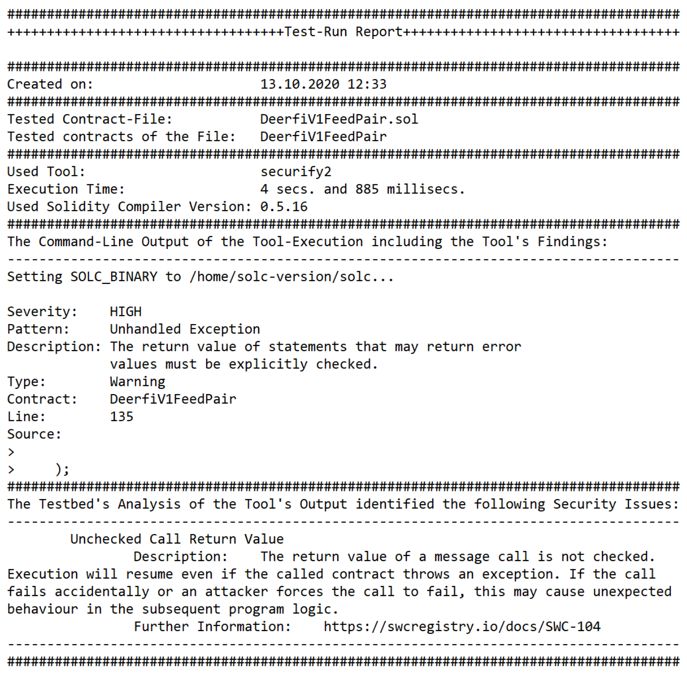
- Obviously, we must pass the path to the contract file. Almost all tools accept the path directly. Only Vandal takes a directory and a regular expression. It then checks the files in the directory and tests if their name matches the pattern. As we only want to test for one contract, our pattern merely fits to a single filename.
- If we test a Solidity file with a tool which does not analyse multiple contracts at once, we must still define the contract's name.
- When Mythril or Securify 2.0 analyse Solidity source code, we must specify the compiler version they should use.
- The tools usually expect the input file to be in Solidity source code. Maian, on the other hand, always needs to know the input type. For the other tools, we only explicitly define it when we want to test a bytecode file.
- If we test Solidity source code with Mythril or Securify 2.0, we must additionally pass the compiler version they should use.
- Furthermore, we need to tell Manticore and Vandal where they should put their results.
- In the case of Maian, we must still determine the vulnerability it should look for.
- As already mentioned in Section 3.1, Mythril verifies potential security issues on a private blockchain. Moreover, it allows to set the number of transactions it should use to validate a vulnerability. To get a good trade-off between analysis quality and operation time, we orient ourselves on the example of the tool's homepage and specify three transactions [66].

5.4.3 The Summary of the Discovered Security Issues

The `_identify_security_issues` methods implemented in the subclasses deliver the security issues a tool has discovered. They retrieve them from the tool's output by using the identifiers of the associated `ToolSecurityIssue` instances.
If a tool prints its findings directly to the command line or a text file, the identifier represents a regular expression. If a pattern matches the tool's output, the script assumes that the tool has found the `SecurityIssue` related to the `ToolSecurityIssue` instance.
For Vandal, the `identifier` gives the name the tool internally uses for the appropriate `SecurityIssue`. These names can be directly loaded from the dictionary stored in the JSON output file.

5.4.4 Identifying Raised Errors

As already mentioned in Section 5.2.4, there are three ways how the testbed detects an `Error`: In one of the `ToolTestRun` subclasses, from the tool's command line output or, in the case of Vandal, from its JSON output file. The first type is always appended to the `_exceptions` set of the `ToolTestRun` base class and therefore directly returned by its `get_errors` method. The other types are identified with help of the `ToolError` instances and returned in the subclasses' `_identify_errors` methods.



```
#####################################################################################
+++++++++++++++++++++++++++++++++++Test-Run Report+++++++++++++++++++++++++++++++++++

#####################################################################################
Created on:                     13.10.2020 12:33
#####################################################################################
Tested Contract-File:           DeerfiV1FeedPair.sol
Tested contracts of the File:   DeerfiV1FeedPair
#####################################################################################
Used Tool:                      securify2
Execution Time:                 4 secs. and 885 millisecs.
Used Solidity Compiler Version: 0.5.16
#####################################################################################
The Command-Line Output of the Tool-Execution including the Tool's Findings:
-------------------------------------------------------------------------------------
Setting SOLC_BINARY to /home/solc-version/solc...

Severity:    HIGH
Pattern:     Unhandled Exception
Description: The return value of statements that may return error
             values must be explicitly checked.
Type:        Warning
Contract:    DeerfiV1FeedPair
Line:        135
Source:
>
>     );
#####################################################################################
The Testbed's Analysis of the Tool's Output identified the following Security Issues:
-------------------------------------------------------------------------------------
        Unchecked Call Return Value
                Description:    The return value of a message call is not checked.
Execution will resume even if the called contract throws an exception. If the call
fails accidentally or an attacker forces the call to fail, this may cause unexpected
behaviour in the subsequent program logic.
                Further Information:    https://swcregistry.io/docs/SWC-104
-------------------------------------------------------------------------------------
#####################################################################################
```

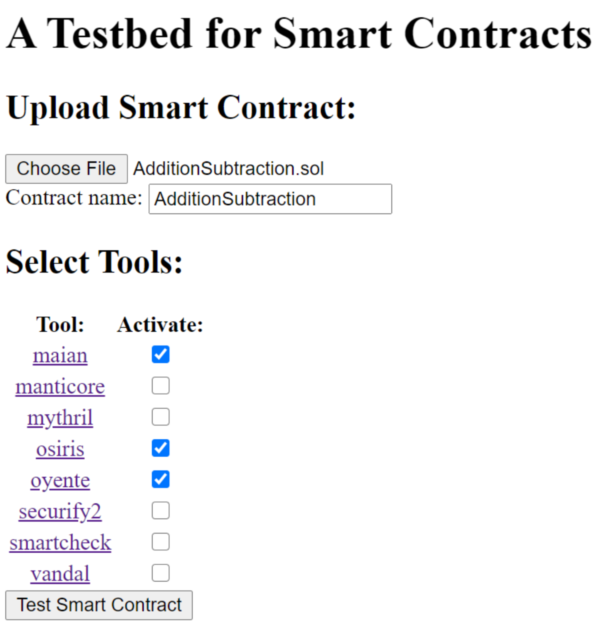
Figure 5.1: A sample report file.

5.4.5 The Report File

The subclasses' `_create_report` method must generate the content for the report file the user gets. In most cases, the subclass can use `ToolTestRun`'s `_create_standard_report` helper method, which guarantees a consistent output format. The report must include the following information:

- The tool's name.
- Whether the tool analyzed multiple contracts or just the defined one. This point is obsolete when the tested contract is in bytecode.
- The contract's filename and, for Solidity contracts, its name.
- The execution time of the `_execute` method.
- Configuration parameters of the testing process. For example, this might be the Solidity compiler version the tool used.
- The command line output of the tool. This is especially helpful if the tool breaks.
- The errors the tool has encountered.
- A detailed record of the security issues the tool has discovered.

A sample report file can be seen in Figure 5.1.



A Testbed for Smart Contracts

Upload Smart Contract:

Choose File AdditionSubtraction.sol
Contract name: AdditionSubtraction

Select Tools:

Tool:	Activate:
maian	✓
manticore	☐
mythril	☐
osiris	✓
oyente	✓
securify2	☐
smartcheck	☐
vandal	☐

Test Smart Contract

Figure 5.2: The Upload Web Page.

5.5 The Web Interface

Now, we present our web interface. It offers two web pages: `upload` and `results`. On the `upload` web page, the user can upload a contract and select the tools he wants to test with. As soon as he hits the submit button, our server starts the requested test run and redirects the user to the `results` web page. There, he gets an overview of the testing process, which he can update by refreshing the web page.

5.5.1 Building a Flask App

Both our web pages are created and managed with the Flask web framework. To use the framework, we create an instance of the `flask.Flask` class, by convention called `app`. Every time, a user visits one of our web pages, the request first goes to the server. A certain prefix of the associated Uniform Resource Locator (URL) then tells the server to pass the request to the Flask app. Next, Flask uses the remaining part of the URL to determine which function it should call. These functions can be registered on the `app` instance and return a `flask.Response` with the requested web page, file or similar [96, 97].

Since it is very cumbersome to create dynamic HTML pages in Python, Flask provides a render engine which generates HTML web pages from templates. We can also pass arbitrary arguments to the render engine. On the template, we later declare how the engine builds the web page. To do so, we can use common code statements as well as the passed arguments [96].
A template can also inherit from another template. We use this to define a base template for both our web pages. This template loads a Cascading Style Sheet (CSS), sets common parts of the web page and determines placeholders in which its children can fill their content. The style sheet also determines how our tool-tips are displayed [98].

5.5.2 The Upload Web Page

On the `upload` web page, the user provides the contract he wants to analyse and chooses the tools he wants to test with. If he tests a Solidity contract, he can furthermore specify which contract he wants to test. A sample of the web page can be seen in Figure 5.2.

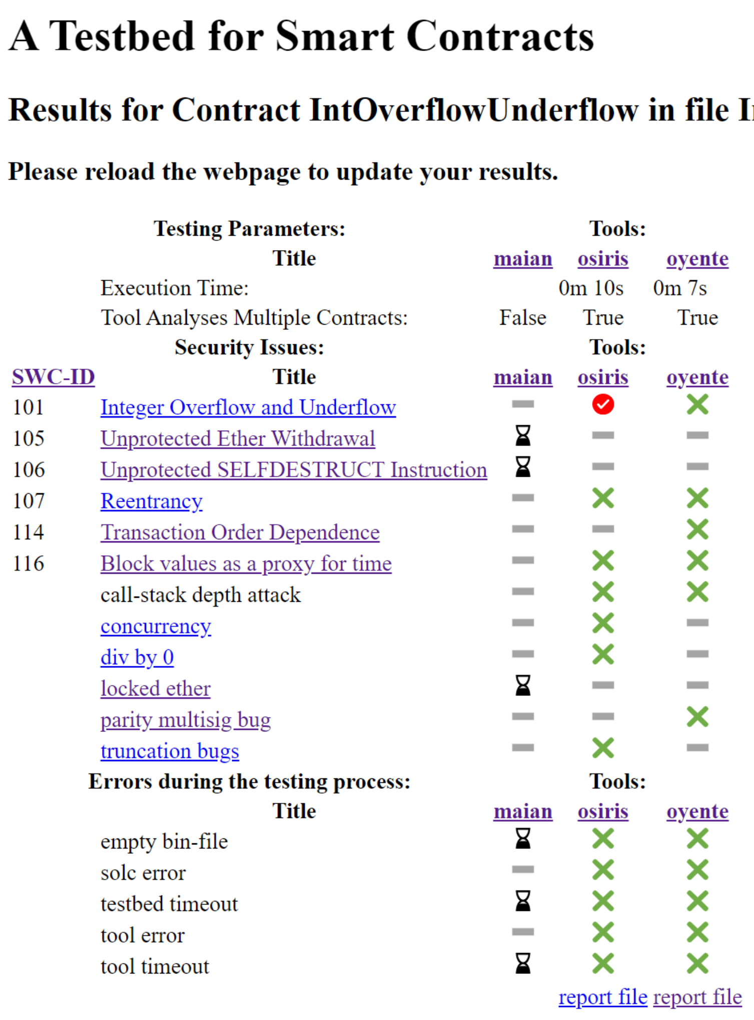
Whenever a user requests the `upload` web page, Flask invokes the `upload` function. The function starts with preparing parameters for the template. It creates a list of all the installed tools as well as another list including only the tools which can analyse bytecode contracts. Furthermore, the extensions of the different bytecode contracts are provided. Now, the `flask.render_template` function [98] constructs the `Response` with the given parameters and the `upload.html` template. In the last step, the `upload` function returns the web page to the user.

The Template. First, the `upload.html` template loads the JavaScript `upload.js` with the web page's client-side logic. Afterwards, it places a form, which is an HTML element utilized to collect input data from the user [99]. In our case, the input is the contract file, the tools the user selects and, if he tests a Solidity contract, also the contract's name.
The file is uploaded with a file picker dialogue which only allows to choose files with one of the extensions passed to the template. Whenever the user uploads a contract, a function in the JavaScript file `upload.js` is called. For a bytecode contract, it deactivates and deselects all the tools incompatible with bytecode contracts. Uploading a Solidity file reactivates them and shows a input text field. Here, the user can specify the `name` of the underlying `SolidityContract` file. If he omits it, the first file in the contract is taken. The text field, in turn, disappears when the user replaces the Solidity source file with a bytecode file.
Next, the tools are presented in a table and linked to their homepage. The link is retrieved from the corresponding `Tool` instance. To select a tool, the user simply marks the checkbox next to the tool's name.
Finally, the `submit` button is placed. It is only enabled when a contract is uploaded and at least one tool is selected. This behaviour is achieved by another JavaScript function which is always triggered when a contract is uploaded or a checkbox is clicked. When the user pushes the button, the form sends its data to the `results` web page. Before doing so, it still deletes the client-side session data to let the `results` web page know that a new test run should start.

5.5.3 Storing the TestRun Objects

When a user reloads the `results` web page, he naturally wants to see the results for his contract. For this reason, an identifier stored on the client-side links to the user's `TestRun` object managed by the server. The identifier is stored as a session cookie on the browser. Flask automatically encrypts these cookies, therefore a user cannot manipulate them to get another user's results [97].
The common data structure to map identifiers to objects is a dictionary. However, allowing an infinite number of test runs at the same time might overload the server. Therefore, the number of active test runs on the dictionary should be limited. Also, accessing a dictionary naively is very dangerous in a multi-threaded application. If one thread finds an available key and wants to map data to it, another thread might have already used it in the meantime.

The TestRunsManager Class. To satisfy all the mentioned requirements, the storage of the `TestRun` objects is managed by an instance of the custom `TestRunsManager` class. This instance only allows to store a limited number of active test runs. On construction, the limit and a time span can be set. The class has two methods. The first one, `put`, takes a `TestRun` and tries to store it in the underlying dictionary. First it request a lock, blocking other threads from executing the method simultaneously. As soon as it receives the lock, it checks whether the number of active test runs plus the number of not started test runs reaches the limit. In this case, it gives back the lock and raises the custom `OverloadError`. Otherwise, it creates a timestamp and uses it as a key to store the `TaskRun` value. Finally, `put` releases the lock and returns the identifier.
The second method, `get`, expects an identifier as its input and returns the corresponding



A Testbed for Smart Contracts

Results for Contract IntOverflowUnderflow in file IntOverflowUnderflow.sol:

Please reload the webpage to update your results.

	Testing Parameters:	Tools:		
	Title	maian	osiris	oyente
	Execution Time:		0m 10s	0m 7s
	Tool Analyses Multiple Contracts:	False	True	True
	Security Issues:	Tools:		
SWC-ID	Title	maian	osiris	oyente
101	Integer Overflow and Underflow	▬	✓	✗
105	Unprotected Ether Withdrawal	⌛	▬	▬
106	Unprotected SELFDESTRUCT Instruction	⌛	▬	▬
107	Reentrancy	▬	✗	✗
114	Transaction Order Dependence	▬	▬	✗
116	Block values as a proxy for time	▬	✗	✗
	call-stack depth attack	▬	✗	✗
	concurrency	▬	✗	▬
	div by 0	▬	✗	▬
	locked ether	⌛	▬	▬
	parity multisig bug	▬	▬	✗
	truncation bugs	▬	✗	▬
	Errors during the testing process:	Tools:		
	Title	maian	osiris	oyente
	empty bin-file	⌛	✗	✗
	solc error	▬	✗	✗
	testbed timeout	⌛	✗	✗
	tool error	▬	✗	✗
	tool timeout	⌛	✗	✗
			report file	report file

Figure 5.3: A Sample of the Results Web Page.

TestRun. If the identifier is not on the dictionary, it raises a KeyError.

5.5.4 The Results Web Page

The `results` web page displays important test parameters as well as the results of the test run. For each tool, it shows the statuses of the `SecurityIssues` and `Errors` returned from the `TestRun`'s `get_security_issue_statuses` or the `get_error_statuses` method (cf. Section 5.4.3). The status *Not Found* is indicated by a green cross, while *Found* is represented by a red check. A black sand glass implies *Loading* and a grey beam stands for *Not Checked*. If a tool has already terminated its test run, the web page also provides a link to download the corresponding report file. A sample `results` web page is shown in Figure 5.3.

Directing to the `results` web page causes Flask to call the `results` function, which then returns the web page. The function starts with checking the user's session data for the identifier. Subsequently, one of the following scenarios happens next:

1. If the session does not contain an identifier at all, the `results` function tries to start a new test run. From the form, it retrieves the names of the tools the user has selected, his contract and possibly the contract's name. It then saves the uploaded file, queries for the requested tools and creates a new `TestRun` instance. Next, it tries

to store the instance in the `TestRunManager`. If it fails, it removes the contract and answers with a `503 Service Unavailable` Hypertext Transfer Protocol (HTTP) error.

2. On success, it starts the `TestRun` and signalizes Flask to store the received identifier in the session cookies.

3. When the session already contains a valid identifier from the very beginning, the `results` function gets the corresponding `TestRun` from the `TestRunManager`.

Now, in the last two error-free scenarios, `results` passes the `TestRun` object to `render_template` and returns the web page.

The Template. The template creates a table with three subsections. The first one shows the test parameters, and the second and third one are the statuses of the `SecurityIssues` or `Errors`.
The test parameters displayed by the first subsection are the execution time of the tool and, if the uploaded file was a Solidity file, whether the tool tests for multiple contracts or just a single one.
The remaining two sections display the titles of the `SecurityIssues` or `Errors`, respectively, as well as their current status. Each title is also linked to the web page associated with the `SecurityIssue` or `Error` instance; hovering over the title shows the description provided by the corresponding instance.
At the bottom of the web page, the terminated tools provide a download link in the form `results/<tool name>`. Following the link passes the `<tool name>` to the `get_result_file` function. The user then receives the report file for the specified tool.

5.6 The Command Line Interface

The second option to interact with the testbed is the command line interface. While the web pages only allow to analyze a smart contract, the command line interface also offers an option to add or remove tools to or from the testbed.
To run a task with the interface, the user has to pass the relevant arguments to the `testbed` Bash script stored in the testbed's root directory. In the next step, the script sets up the testbed's virtual environment and starts the command line interface along with the passed arguments.

5.6.1 Parsing Arguments

The given arguments are parsed with help of the standard library module `argparse`. It does not only simplify the parsing procedure but also prints useful help- and error-messages to the command line [100].
At the beginning, the command line interface creates an `argparse.ArgumentParser` object, which expects the arguments to be in the following order:

```
<positional argument(s)> [<optional argument(s)>]
```

A *Positional Argument* is, as its name already suggests, identified by its position. It must always be present and can either be the value of an argument or the name of a sub-parser. A sub-parser is another `ArgumentParser` registered to the root parser. It consumes all the arguments coming after it and parses them itself [100].

An *Optional Argument* does not need to be provided. Therefore, it cannot be found by its position and must be preceded by its name. Most arguments have a long and a short name. The long name is led by two and the short one by a single dash, like `--command` or `-c`, respectively [100].

The Help Message. By default, every `ArgumentParser` automatically offers the options `--help` or `-h`, which print out a help message. The help message lists the `ArgumentParser`'s usage and every argument registered on the parser. A programmer can also provide a short explanation at the time when he adds an argument to the parser. These explanations are then additionally printed out with the arguments.
If an argument is a sub-parser, only its name and its potentially existing description is printed out. To find out how to use the sub-parser itself, the user must pass the `--help` option directly to the sub-parser, like `testbed sub-parser --help` [100].

Adding a Sub-Parser. To register a sub-parser to a parser, a "special action object" [100], returned by the parser's `add_subparsers` method, must first be obtained. By calling the object's `add_parser`, a new sub-parser is afterwards added to the parser [100].

Adding an Argument. Other arguments are registered with the parser's `add_argument` method. By providing the relevant parameters, we can define the argument's properties [100].
For example, we can limit the possible values to a set of choices or a certain type. The type can either be a built-in type or a function expecting a single string and returning a parameter with the demanded type. We use the latter when we expect a path to a file or a directory. In this case, we call a function which checks whether the path exists and whether we have the required permissions on it. For files, we check if it is a file and has the desired file extension, for directories, we make sure that it is a directory. Finally, the function converts the path to an absolute path and returns it [100].
Optional arguments can also have a default value or be set to expect multiple values or no value at all [100].

5.6.2 Options of the Command Line Interface

The root parser of the command line interface has three different sub-parsers installed. The `analyze` sub-parser tests a given smart contract, the `update` sub-parser adds a tool or updates an existing one and the `remove` sub-parser removes a tool. To select one of them, a user must simply type its name followed by the arguments of the sub-parser.

The Analyze Sub-Parser. The `analyze` sub-parser takes the following arguments:

- `contract_path` is the only positional argument and specifies the path to the contract. The file extension must either be `.sol` for a Solidity contract or `.bin` or `.hex` for a bytecode contract.
- `--contract_name` or `-n` determines the name of a Solidity contract. If it is not provided, the name of the first contract in the file is chosen.
- `--tools` or `-t` takes the names of the tools the user wants to test with. By default, these are all the tools on the testbed.
- With `--output` or `-o`, the user can tell where he wants the folder with the results to be stored. If omitted, the testbed creates the folder in the current working directory.

When the interface recognizes that the user wants to analyse the given contract, it first creates a `Contract` or `SolidityContract` instance, depending on the `contract_path`'s file extension. In the next step, it loads the selected tools and creates a `TestRun` instance to test the contract with the given tools. Afterwards, it permanently asks the `TestRun` for terminated tools. If a tool has terminated, it copies its report file to the specified output directory and tells the user where to find it. At the end, when all tools have finished, the testbed still provides a text file with an overview of the discovered `SecurityIssues` and the occurred `Errors`. The overview contains the same information as the table presented by the `results` web page (cf. Section 5.5.4).

The Update Sub-Parser. With the `update` sub-parser, an administrator adds or updates the information the testbed needs to run a tool (cf. Section 5.2). If he updates an existing tool, given options overwrite the old values, if not stated otherwise. The arguments are listed below:

- `name` is a positional argument and gives the name of the tool.
- `--script` or `-s` determines the path to the python script containing the tool-specific code. Naturally, the file must end with `.py`. This argument must be provided when a new tool is integrated.
- `--link` or `-l` allows to specify a link to the tool's homepage.
- Providing the `--bytecode` or `-b` option indicates that the tool can also handle bytecode files.
- `--solidity` or `-sol` sets the preferred Solidity compiler version of the tool.
- Supplying the `--analyses_all_contracts` or `-a` argument states that the tool analyses all the contracts in a Solidity file.
- `--tool_security_issue` or `-i` expects the path to a `.csv`-file, where each row presents a `ToolSecurityIssue` associated with the tool. The first column contains the title of the security issue while the second column contains the identifier. If the specified tool already exists, the new `ToolSecurityIssues` are simply added to the old ones.
- `--tool_errors` or `-e` is similar to `--tool_security_issue`, except that the columns represent `ToolErrors` instead of `ToolSecurityIssues` and that a further

If a user needs to access the database in a more sophisticated way, for example to add new `SecurityIssue`s, he needs to interact with the database directly.

The Remove Sub-Parser. The remove sub-parser only has the positional argument `name`, which identifies the tool to remove.

6. Evaluation

Finally, we want to analyse how our testbed works on real world smart contracts. In Section 6.1, we explain how we retrieved our test set. We then outline our experimental setup in Section 6.2. Next, in Section 6.3, we evaluate how often the underlying tools break while we pay extra attention on their compatibility with different Solidity versions. Afterwards, in Section 6.4, we evaluate their execution time. Eventually, in Section 6.5, we inspect how many security issues the individual tools and the entire testbed found and how often they agreed or disagreed on a certain weakness.

6.1 Choosing the Data Set

Unfortunately, we cannot directly use the contracts from the blockchain, since it only stores their bytecode. Therefore, we retrieved our data set from the Etherscan project which provides a big database with verified open source smart contracts [75]. It also offers a free JSON-application programming interface (API), so that we could download the latest 8,000 smart contracts from their web site [101]. However, analysing the whole data set is challenging due to our limited resources. For this reasons, we set some restrictions on our testing process:

1. We limit our test set to 120 contracts.
2. Many contracts begin with the `pragma` directive which defines the Solidity version scope a contract is compatible with (cf. Section 2.5). Since big changes of the language come with major version changes, most contracts are only allowed to be compiled by Solidity compilers of a major version. We want to know if there is a correlation between a failed test run of a tool and the major version a contract is written for.
   Picking 120 contracts uniformly might not give us enough contracts for every major version. Therefore, we derive four different subsets which only allow their contracts to be compiled with the compiler versions `0.4.x`, `0.5.x`, `0.6.x` and `0.7.x`, respectively. We did not consider prior versions, since our testbed does not support them.
3. Moreover, we omit contract files containing several contracts, since we cannot fairly compare them across all tools.
4. Obviously, we also do not want to waste any resources on duplicates. For this reason we use the `difflib.SequenceMatcher`, which enables us to calculate the similarity of

sequences [102]. With its help, we can guarantee that our test set does not contain any pair of contracts where more than 80 percent of the bytes in their bytecode matches.

5. Furthermore, we cap the maximal execution time to 30 minutes for each tool.
6. As a direct consequence for the last constraint, some tools frequently time out for contracts with a bytecode size of more than 30,000 bytes. For this reason we only analyse contracts with a size below this threshold, which shrinks our data set to 79% of the original size. To also guarantee that the contract size in the subsets is approximately equally distributed, we still divided each subsets in three further subsets, containing contracts with a bytecode length of up to 10,000, 10,001 to 20,000 and 20,001 to 30,000 bytes, respectively.

Eventually, we drew 10 contracts from each of the twelve different subsets. In Table 6.1, we list the subsets and the addresses of the contracts we retrieved from them.

Contract Size→ / Solidity Version↓	0-10,000 bytes	10,001-20,000 bytes	20,001-30,000 bytes
0.4.x	0x247ba9557b84c2e4557389be3a47929a17345802	0xaa79b0c8455ca04c3733de0ce412f4cab5d3ac0a	0xb0399c2fb7958d8d0fde93ec58c4efa1ba501375
	0x4cd2f77c388e8d266921ba1bef2e519e3118703f	0xc6a2aa7ebe041336d433919bd4dcc1ef5d98ab99	0x0c2e3bbc4646c872a3F5A745D2902aFFda82C58a
	0x2ad770f563b752215eb093501ad3ac8672983904	0x4c7b965c22562ba4cd06de4177afc09311f0bf03	0xe5F5D824232636Db9d97748f61069a336295EF2c
	0xa854184bd106ec2529f9840bdf2512ffdae71515	0x8a5ad873a1a615001acc1757214f67e1ba145cc9	0xf0a95500f3ac1b2d70e1d8dfa7cd2d863afb6069
	0xe5912f71baf926c2de0db39f182de797d356a5e3	0xe0388fa2fa04557a82616386ec0e7cf59883ae35	0x04d23f2b1867e66320ec8066005b6da27031c200
	0xca7f86f12e7b99c0ae6b4310873312dedf74d0f9	0x2f82d5e4370d526187c15e0703d64d903517571a	0xe17f017475a709de58e976081eb916081ff4c9d5
	0xcc0b7707ba4d7d7f9acdd16ab2e0b1997e816166	0x4a008fbf74c4fef109050ed692e11e25fa57f31f	0x8a28b1ef158a3511a0629b3ee7293bd95cbf2c97
	0xae5a363b170aa6417cb9f64df51195c34b188f9f	0xa4f27e7968bc0b4f805728fa84ed0307c9b44e20	0x3d76cd9723e0cc8875907cf944c147ee4bafb29e
	0x8033646e311bc101a1ccf4b2fa76663884e42c33	0x49d2b7568eeea849d6d1799c83e8590dae55fa4c	0xc1697d340807324200e26e4617ce9c0070488e23
	0xdd371794b0267fe6dc878f5ccbfdc90dc8f8cb28	0xadcc785d2f82d9e6f5bf796d07d6f9c6bb0bf8df	0x01fa1B31766c0e58a2C66b6FBa3C36128aea60E4
0.5.x	0x0d932921814b416b9170390669b63fc5c50cebf9	0xa45b966996374E9e65ab991C6FE4Bfce3a56DDe8	0x67bd1e1f72c130c3243b6f9eda708e6d9461ae4f
	0x0231f96bcbccd3d7b64ef63b55c652447cac2a29	0x02525172418dfd0c828c567ddbfab064e60831a2	0x3Cc63313c1241d832E6d6D5c48c0E523589326f7
	0x872605588d8C0688ACC7BfFdD2C6fdB9f1b2232d	0x75eaa6a51a5a0438db9e5a1a5fe9819ccd407131	0xdf4441155ca1f7443f7ed51a556970f7037a9a45
	0x7449e15abcb212a746ad6749d957328c35c1337d	0x8d4193fb3871c43c442ea17c6d9bb33d0644e81d	0xe29c11ac01bcf8776c7a6ef3595ca0529b5e2521
	0x91dd243ec20b1a37bef8a5cf02c991d622fbfd72	0x766a6494612400cefd0622c8cbf572fe8c4e875e	0x13c1542a468319688b89e323fe9a3be3a90ebb27
	0xeeae0c791d38e1e1fc9f7989d081c9fc290716dd	0x6a8e6abbdadecf0da5c0053951e3742807a8a3e1	0x6d313883c6db5ea58bdd5546f403f1fe27e92690
	0xCc72149a6e99f6376f75EfDeF0C803A5e12cC72a	0xa66c9810bb78bdbca1cdb60184c8560f4f5bd213	0x8A27B668e9C53E793dF468693cb2a5278927aefD
	0xeefB1D06d28286216E5d7068f9Ef21e01F4DD793	0xbefe81ce680df6cea99f357ed764cf626f05a4c5	0x3A10cc47571b341e7edcB97bcA9F287994bAa58C
	0x65f8b410e25dff32453b60b6b02bb07af7fa9068	0xddefdbf7c5a75001ebfd5ea602e0637355a24609	0x9d0901bedcea5c2c803cd2f4f5902be2a62a3c7d
	0x8ba56ad0e17ef0bb39c52d2c313fc328dbf3a789	0xD8Ee69652E4e4838f2531732a46d1f7F584F0b7f	0x9403d608515f7346EC44B998984d7741Fd0D9bd4
0.6.x	0xd9BFB71F7BD6fc9fDCF69Be352CAa77da885A007	0xecc3114de29372d321cddde64aceb6c6c8921c19	0x7c28310cc0b8d898c57b93913098e74a3ba23228
	0x5e5a7b76462e4bdf83aa98795644281bdba80b88	0x527CAF0E399F83CfBdBd2C65Fb691Ca39ad4E1d9	0x332af4ddd9553710a5633d86f0cffc93f4763fde
	0xdb5fe0d4ea64e3264fc1cd963462a246f079b55c	0xf48f47c959951b1a8b0691159a75a035dfed2d1d	0x821406dbdbbceaf41282cb0e6374b2db06c91252
	0xca1d63c7173c994a8e608836806dc594ad3b5e66	0x034455c8a9882bf44c9704c780a55198e05ba559	0xb434dcdb16819bc99e0df069f10a5d6a8f8cad2b
	0xd8f5ae485a2b62bc2f14525e526d4f15b41453cd	0x700f3f509f8361998418f88df7d40ac411441928	0xa3b69e77840ea3e655eba3822539d498bea72156
	0x16F0a50305c6aCf1A44A3DAa225531e731f39A9F	0xd2c1a6d52b6f4a808d08b722d6a6641df5d98fb2	0xeca82185adce47f39c684352b0439f030f860318
	0x22bbcd36fb376561be1146b950b4f76c446d0605	0x9B53D7Bb657F5f14a7F73C2d2B90cc084Cae1fE4	0x674c6ad92fd080e4004b2312b45f796a192d27a0
	0x003288d46471359320c9cc47696a55e6c1698bb8	0x5bEaBAEBB3146685Dd74176f68a0721F91297D37	0x6c5fa7a3c2cd98a689b1305bd38b56120fe15744
	0x210473E4c6bdFbf612D7289097891c93E976ce1E	0xcaf234ae8112d30d8b20712895937d23787c052d	0x99b8341f7855e56f98aae25568722ac02ab08c86
	0xf0c9cc5531aa340245f093330f4c6f29a8ed6252	0x41a2e2f0e3a93c16d124c635ba4afab4abaff2b7	0x0a312995811cee77e3e2eb3e017bffdc7fe6b165
0.7.x	0x5d13507eaf2e22d1822ebc8011d46d5641bc92da	0x96853087c9e364fe9554fd6fcfe13bcd942b04d5	0xEB2f5eC2DCeF6efC53935015f1Cf367E32EDcb5D
	0xcf3c059a249183a07a9dd9d4cb88e1a93e7311e0	0x3797e27d486873cbd9f36758cec27b27bb525e25	0x54e1EB00b7978314D33e0dbB113e745Eddd18FFe
	0x6E5952C033a0daDB2d93B594Dea9eBF2dE6A9cC2	0x155ff1a85f440ee0a382ea949f24ce4e0b751c65	0x4f4cd2b3113e0a75a84b9ac54e6b5d5a12384563
	0x6639d71af9ce0bbf668ffacb6fa8cf08d6c78758	0x3ecabbc9f7aad28d454084ea9949339baa745b96	0xc04e4fc8b9efdef1badd0430497c5f99410ee66e
	0x575b10b62a41f269d6d2a65a78297ba0ea455ee8	0xa0dceab8eedd571d0a8bcab1fe50678297e898eb	0xb422567B17a70CAd2Eb0D191490690b1e7022bcB
	0xa1f1cf045fd9d964feb29196c6fd3758319a5e49	0x7afb8ba591f0e22486e2e040ee213aeb86413820	0x2924ab10b6b94e7206f301fc044a0c85b7a1d4b3
	0xbf2a61B16d3BC44e35db46E8081aB0Fd29dcCadF	0xaf0d10bc1fdc60823c718836726f06ccdb35abf0	0x9ceb84f92a0561fa3cc4132ab9c0b76a59787544
	0x33729ec887c974cc6f692646c28137d22517b03f	0x65fafd78795c4bb2734633a000c122be0e3e6f37	0x5d3a4F62124498092Ce665f865E0b38fF6F5FbEa
	0x948adb4087929d119964d3d783b28e91e1160def	0x14eb60f5f270b059b0c788de0ddc51da86f8a06d	0x72bea880b07b3c814c1753a1d1c3bdefe23af3dd
	0x4fe83213d56308330ec302a8bd641f1d0113a4cc	0x188d315b1c698ca678c8e3891ac21e097763afde	0x2ff5b448bfdd0c04b9ab13612a40d5d4fb28b3b7

Table 6.1: The addresses of the contracts used in the evaluation. Their source code can be found at `https://etherscan.io/address/<address>`

6.2 Experimental Setup and Performance

After the selection of the contract we ultimately analysed them on our testbed. We hosted the VM on a computer running Windows 10. It operates two 8 GB DDR3 memories, each one with a clock rate of 1,600 MHz. Their exact model name is G.Skill F3-1600C11S-8GNT. The CPU model is the six-core AMD FX(tm)-6350 processor with a clock rate

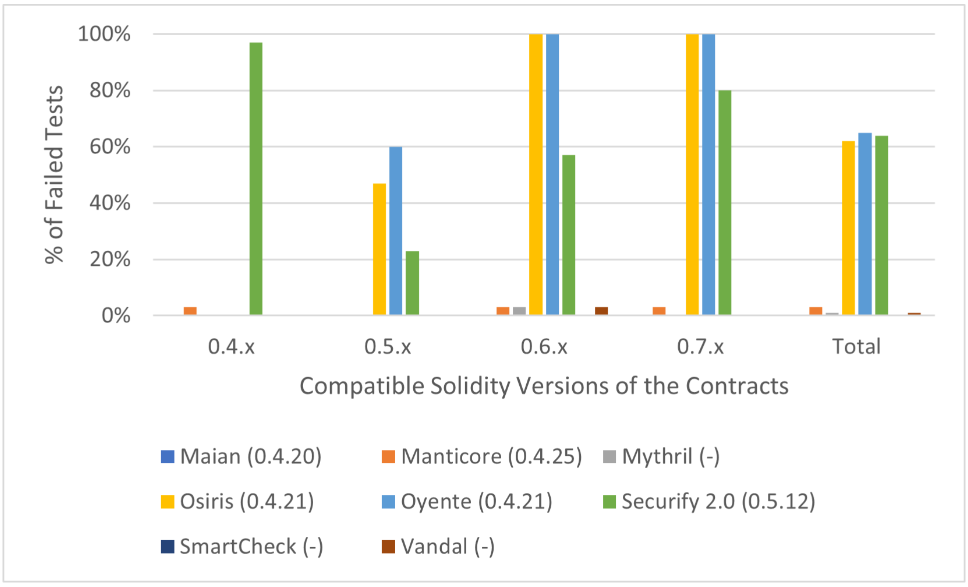


Figure 6.1: The major Solidity versions with which the tools fail. The value in brackets is the version of the Solidity compiler deployed with the tool.

of 3.9 GHz per logical processor. Memories and CPU are integrated on a MSI 970A SLI Krait motherboard. The VM was allowed to use 10 GB of our memory and two of the logical processors. We tested the tools subsequently so that the resource consumption of a tool does not affect the execution time of the other tools. Our overall testing procedure took about 75 hours.

6.3 Failed Tests

In Figure 6.1, we distinguish our contracts based on the different major Solidity version they are written for and show how often our tools broke in each category. As stated in Section 5.1, some of the tools are deployed in a container with a pre-installed Solidity compiler. Since these tools are most likely well-tested with this compiler version, it is especially interesting what happens when they are forced to use another one.

Erroneous Tools. As we can see, Osiris and Oyente performed quite similar, which was expectable, since Osiris extends Oyente [11]. They were mostly error-free on contracts with the same major Solidity version as their embedded compiler, namely version `0.4.21`. Otherwise, they often failed, for contracts with a version higher than `0.6.0` even every single time. This result is somewhat counter-intuitive since they analyse bytecode internally. A possible reason is that some parameters of the compiler output, like the contract's AST, occasionally change with major version updates [33, 34, 35].
Securify 2.0 uses this AST, which could explain why it failed a lot when it was forced to use another major version than the one it is embedded with. However, the AST only changes with version `0.5.0` [33] and `0.7.0` [35], therefore it cannot justify why the tool also broke for contracts written for the major Solidity versions `0.5.x` and `0.6.x`. Yet, Securify 2.0 analyses Solidity source code directly [13] and therefore, we assume that it failed because it is unfamiliar with some language constructs.

Mostly Faultless Tools. The other tools fail quite less frequently. Manticore broke four times; each time it experienced problems with Zlib [103], a "data compression library" it depends on.
Mythril compiled the single failing contract with the Solidity compiler `0.6.12`, which failed due to some imprecisely defined license identifiers in the contract. The other tools either

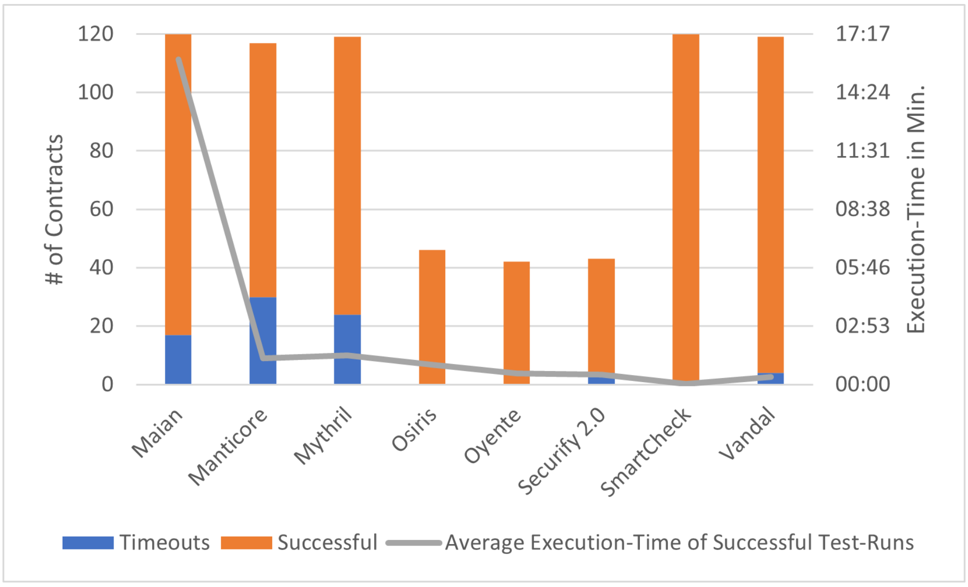


Figure 6.2: The timeouts and the average execution time of the tools.

did not compile the contract or used another compiler version.
Vandal only accepts bytecode and therefore the testbed compiles the Solidity contracts directly before passing them to Vandal. Hence, it was no surprise that Vandal never encountered problems except for once, when it somehow could not identify the given contract file.
SmartCheck and Maian did both not encounter any problem at all.

6.4 Execution Time

In Figure 6.2 we illustrate how often each tools exceeded the timeout of 30 minutes and how long the average execution time of a successful test was. With successful, we mean that the tool neither encountered an error nor timed out. Maian, Manticore and Mythril all validate their findings on a private blockchain (cf. Section 3.1), which presumptively justifies their large number of timeouts and their higher mean execution time.
Symbolic execution can take a lot of time for reasons we have already pointed out in Section 3.2. This possibly explains why the single tool not using it, namely SmartCheck, analyses the average contract in about 2 seconds. This is almost eight times quicker than the second fastest tool, Vandal, with about 17 seconds.

6.5 The Discovered Security Issues

In Table 6.2, we narrow on eight security issues. For each one of them we record how often it was found by our testbed or the tools. We also give the percentage of flagged contracts compared to the successfully analysed contracts.
In some cases Table 6.2 contradicts with Table 3.1 on whether a tool looks for a certain security issue. The reason is that we obtained the most information for Table 3.1 from a paper published one and a half years ago while we take the current data for Table 6.2. In the meantime, some of the tools have changed the vulnerabilities they detect [104].

The Total Number of Flagged Contracts. In total, our testbed detected one of the listed potential weaknesses in 111 contracts. The most frequently reported one was the reentrancy bug, with 64 occurrences, while the testbed did not find any contract with an unprotected selfdestruct instruction.

Security Issues:	Reentrancy	Over- or Underflow	Unchecked Call	Call-Stack Depth	Locked Ether	Unprotected Selfdestruct	TOD	Timestamp	Total
Testbed	**64** (53%)	**35** (29%)	**50** (41%)	**4** (3%)	**21** (17%)	**0**	**61** (50%)	**8** (6%)	**111** (92%)
Maian	✗	✗	✗	✗	0	0	✗	✗	**0**
Manticore	1 (1%)	3 (3%)	✗	✗	✗	0	0	3 (3%)	**7** (8%)
Mythril	2 (2%)	2 (2%)	0	✗	✗	0	✗	✗	**3** (3%)
Osiris	✗	17 (37%)	✗	✗	✗	✗	✗	✗	**17** (37%)
Oyente	0	30 (71%)	✗	4 (9%)	✗	✗	3 (7%)	0	**32** (76%)
Securify 2.0	0	✗	15 (38%)	✗	2 (5%)	0	4 (10%)	5 (12%)	**19** (49%)
SmartCheck	✗	0	33 (27%)	✗	20 (16%)	✗	57 (47%)	0	**88** (73%)
Vandal	64 (55%)	✗	30 (26%)	✗	✗	0	✗	✗	**66** (57%)

Table 6.2: The number of contracts where the tools or the testbed found a certain security issue. The value in the brackets shows their percentage of all the contracts successfully analysed by the appropriate tool. ✗ indicates that the tool does not look for the corresponding security issue.

As we can see, the results of the individual tools vary wildly. Maian, for example, discovered none of the issues in any contract. These results partly confirm to the analyses in Maian's paper. There, the authors could only flag one and a half in thousand contracts as containing an unprotected selfdestruct. However, in contrast to our evaluation, they assumed a locked Ether bug in 3,2% of the contracts in their test set [8].
Manticore and Mythril, too, reported only a few contracts as vulnerable. They merely assume 8 or 3 percentage of all successfully analysed contracts as risky. This is comparably low, since the fourth least frequently reporting tool, Osiris, already flagged 37% of all its successfully analysed contracts. Oyente, which is most likely to mark a contract, even suspects more than three third of its successfully tested contracts to be problematic.
A possible reason for this phenomenon could, of course, be that the various tools look for different security issues and thus do not report the same contracts.

Disagreement between the Tools. However, the explanation suggested above contradicts to our data. Table 6.3 shows how frequently two tools disagree on whether a contract has an Over- or Underflow or a Reentrancy bug. As we can see for the Over- or Underflow vulnerability, Oyente and Mythril, for example, dissent on 81% of the contracts successfully analysed by both tools. The same peculiarity applies between Oyente and Osiris, even though Osiris extends Oyente. For the same security issue, these tools oppose themselves in more than 40% of the contracts they both analysed successfully.
A similar situation can be observed between Manticore and Vandal. For the reentrancy bug, their results conflict on 62% of the contracts successfully analysed by both tools.

Tools	Security Issue	Manticore	Mythril	Osiris	Oyente	Securify 2.0	SmartCheck	Vandal
Manticore	Over- or Underflow	-	2/74 (**3**%)	14/35 (**40**%)	23/31 (**74**%)	✗	0/87	✗
	Reentrancy	-	2/74 (**3**%)	0/35	0/31	0/27	✗	51/82 (**62**%)
Mythril	Over- or Underflow	2/74 (**3**%)	-	12/35 (**34**%)	25/31 (**81**%)	✗	2/95 (**2**%)	✗
	Reentrancy	2/74 (**3**%)	-	0/35	0/31	2/28 (**7**%)	✗	50/90 (**56**%)
Osiris	Over- or Underflow	14/35 (**40**%)	12/35 (**34**%)	-	17/42 (**40**%)	✗	17/46 (**37**%)	✗
	Reentrancy	✗	✗	-	✗	✗	✗	✗
Oyente	Over- or Underflow	23/31 (**74**%)	25/31 (**81**%)	17/42 (**40**%)	-	✗	30/42 (**71**%)	✗
	Reentrancy	0/31	0/31	0/42	-	0/12	✗	22/42 (**52**%)
Securify 2.0	Over- or Underflow	✗	✗	✗	✗	-	✗	✗
	Reentrancy	0/27	2/28 (**7**%)	0/14	0/12	-	✗	14/39 (**36**%)
SmartCheck	Over- or Underflow	0/87	2/95 (**2**%)	17/46 (**37**%)	30/42 (**71**%)	✗	-	✗
	Reentrancy	✗	✗	✗	✗	✗	-	✗
Vandal	Over- or Underflow	✗	✗	✗	✗	✗	✗	-
	Reentrancy	51/82 (**62**%)	50/90 (**56**%)	26/46 (**57**%)	22/42 (**52**%)	14/39 (**36**%)	✗	-

Table 6.3: The value before the slash shows how often two tools disagree for a certain security issues. The value after the slash gives the number of contracts successfully analysed by both tools. The percentage sets these values in proportion. Sometimes one of the compared tools does not look for a security issue. In this case, we indicate this with ✗.

Since the multiple tools each have their own way to search for a certain security issue, their dissent must have its origin from diverse false positives and negatives. Osiris, for instance, uses taint analysis to reduce its false positive rate [11, 12]. Oyente does not take this step and therefore Osiris possibly produces far less false positives than Oyente. This assumption is further reinforced, since independent analyses diagnosed a high false positive rate for Oyente [105]. Moreover, we assume that it is no coincidence that the three tools additionally validating their findings on a private blockchain report the fewest contracts.

Consequences. As a consequence of our evaluation, we can see that testing a contract with multiple tools is substantial. This has various reasons. First of all, using only a single or a few tools might not give any results since they might all encounter an error during their execution. Secondly, more tools usually cover more security vulnerabilities. And, last but not least, two tools looking for the same security issue often disagree on whether a contract contains that issue or not. Testing with a single tool will therefore produce more false negatives.

7. Conclusion

The contributions of our thesis are manifold. We present a testbed which combines eight security analysing tools. With our two user interfaces, we simplify the analysis of a smart contract with all these tools into a single test run. While our command line interface reduces their execution to an intuitive command, our self-explaining web pages completely supersede the necessity of reading any documentation. Moreover, we even give our users a clear overview to easily compare the outcomes of the various tools.
Additionally, we evaluate the eight tools and compare their results with the testbed running all of the tools together. We show that analysing a contract with various tools does not only cover a wider range of security issues. Instead, even when several tools look for the same security issue, testing with multiple tools often flags a lot more contracts than any of the investigated tools individually does. This demonstrates the necessity of testing a contract with a number of tools since otherwise many false negatives would probably be missed.

Future Work. In the future we could obviously integrate additional tools into the testbed. Luckily, our testbed's object-oriented and modular design enables other programmers to conveniently extend it.
Every one of the built-in tools presents a variety of possibilities to customize their analysis. We, on the other side, always use default settings to keep our testbed as user-friendly as possible. Yet, the testbed could additionally offer advanced options so that the experienced user can fully leverage the potential of the underlying tools.
Currently, our server uses a naive approach to prevent from overloading. It just refuses any requests as soon as it operates a fixed number of active test runs. A more sophisticated strategy could adjust the quantity of permitted test runs with the server's workload while also terminating long test runs on traffic peaks.
In addition, the web interface could improve the user experience with an overhauled design. Moreover, the `results` web page could automatically retrieve the results of the testing procedure which would omit the necessity to reload the page manually.
Additionally, we could extend our database by classifying the security issues by their severity level. The user could then choose to only see issues of a higher level and our web page could additionally highlight the different categories in terms of color.
Some of the tools also report the lines of the code where they found a certain vulnerability. On the web server, we could display the tested contract and then use this information to mark the problematic positions. Colors could furthermore indicate the vulnerabilities' severity and the number of tools which flagged these issues. Eventually, we could even

enable the user to directly edit these lines so that he can instantly submit the modified contract for another test run.

Acronyms

API	application programming interface
AST	abstract syntax tree
CFG	control flow graph
CPU	central processing unit
CSS	Cascading Style Sheet
DAO	Decentralized Autonomous Organization
DoS	Denial of Service
EOA	externally owned account
EVM	Ethereum Virtual Machine
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ILF	Imitation Learning based Fuzzer
IP	Internet Protocol
IR	intermediate representation
JSON	JacaScript Object Notation
LTS	Long-Term-Support
NPM	Node Package Manager
ORM	object-relational mapping
PoW	Proof of Work
SSA	single static assignment
SWC	Smart Weakness Classification
TOD	Transaction Order Dependency
URL	Uniform Resource Locator
VM	virtual machine
XML	Extensible Markup Language

Bibliography

[1] P. D. H. Liam Kelly, Sarah Rothrie, “Who’s Afraid of Ethereum? The Top 12 Smart Contract Platforms,” accessed 14.11.2020. [Online]. Available: https://cryptobriefing.com/whos-afraid-ethereum-top-12-smart-contract-platforms/

[2] “Today’s Cryptocurrency Prices by Market Cap,” accessed 14.11.2020. [Online]. Available: https://coinmarketcap.com/

[3] N. Atzei, M. Bartoletti, and T. Cimoli, “A Survey of Attacks on Ethereum Smart Contracts (SoK),” *Principles of Security and Trust*, 2017.

[4] M. D. Angelo and G. Salzer, “A Survey of Tools for Analyzing Ethereum Smart Contracts,” *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, 2019.

[5] “The Parity Wallet Hack Explained,” accessed 14.11.2020. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/

[6] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” 2008, White Paper. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[7] V. Buterin, “Ethereum White Paper - A next generation smart contract decentralized application platform,” 2013. [Online]. Available: https://ethereum.org/en/whitepaper/

[8] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding The Greedy, Prodigal, and Suicidal Contracts at Scale,” *34th Annual Computer Security Applications Conference*, 2018.

[9] trail of bits, “Manticore,” accessed 14.11.2020. [Online]. Available: https://github.com/trailofbits/manticore

[10] B. Mueller, “Smashing Ethereum Smart Contracts for Fun and Real Profit,” *HITB SECCONF Amsterdam*, 2018.

[11] C. F. Torres, J. Schütte, and R. State, “Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts,” *34th Annual Computer Security Applications Conference*, 2018.

[12] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making Smart Contracts Smarter,” *2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[13] E. S. Lab, “Securify 2.0,” accessed 14.11.2020. [Online]. Available: https://github.com/eth-sri/securify2/tree/71c22dd3d6fc74fb87ed4c4118710642a0d6707e

[14] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “SmartCheck: Static Analysis of Ethereum Smart Contracts,” *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018.

[15] L. B. et al., “Vandal: A scalable security analysis framework for smart contracts,” *CoRR.*

[16] Z. Koticha, “Introduction to Blockchain through Cryptoeconomics - Part 1: Bitcoin,” accessed 14.11.2020. [Online]. Available: https://medium.com/blockchain-at-berkeley/introduction-to-blockchain-through-cryptoeconomics-part-1-bitcoin-369f245067f9

[17] S. Seth, “Public, Private, Permissioned Blockchains Compared,” accessed 14.11.2020. [Online]. Available: https://www.investopedia.com/news/public-private-permissioned-blockchains-compared

[18] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak, “Proofs of Space,” *Advances in Cryptology – CRYPTO 2015*, 2015.

[19] “Proof of Stake (PoS),” accessed 14.11.2020. [Online]. Available: https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake/

[20] S. Nadeem, “How Bitcoin mining really works,” accessed: 14.11.2020. [Online]. Available: https://www.freecodecamp.org/news/how-bitcoin-mining-really-works-38563ec38c87/

[21] V. Buterin, “On Slow and Fast Block Times,” accessed 14.11.2020. [Online]. Available: https://blog.ethereum.org/2015/09/14/on-slow-and-fast-block-times/

[22] J. Martinez, “Understanding Proof of Stake: The Nothing at Stake Theory,” accessed 14.11.2020. [Online]. Available: https://medium.com/coinmonks/understanding-proof-of-stake-the-nothing-at-stake-theory-1f0d71bc027

[23] Z. Koticha, “Introduction to Blockchain through Cryptoeconomics, Part 2: Proof of Work and Nakamoto Consensus,” accessed 14.11.2020. [Online]. Available: https://medium.com/mechanism-labs/introduction-to-bitcoin-through-cryptoeconomics-part-2-proof-of-work-and\-nakamoto-consensus-1252f6a6c012

[24] “How Blockchain Changes the Nature of Trust,” accessed 14.11.2020. [Online]. Available: https://www.linuxfoundation.org/blog/2019/01/how-blockchain-changes-the-nature-of-trust

[25] P. Kasireddy, “How does Ethereum work, anyway?” accessed 1.1.2020. [Online]. Available: https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369

[26] A. T. Don Tapscott, *Blockchain Revolution: How the technology behind bitcoin is changing money, buisiness and the world*, 2016.

[27] D. Heaven, “A house has been bought on the blockchain for the first time,” accessed 14.11.2020. [Online]. Available: https://www.newscientist.com/article/mg23631474-500-a-house-has-been-bought-on-the-blockchain-for-the-first-time/

[28] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi, “A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics,” *IEEE Access*, 2019.

[29] “Ethereum accounts,” accessed 14.11.2020. [Online]. Available: https://ethereum.org/en/developers/docs/accounts/

[30] “Ethereum Yellow Paper,” accessed 14.11.2020. [Online]. Available: https://github.com/ethereum/yellowpaper/tree/3e2c0890ddc3200b8e03c1f4154789fd5b44c76b

[31] D. Mihov, "These are the top 10 programming languages in blockchain," accessed 14.11.2020. [Online]. Available: https://thenextweb.com/hardfork/2019/05/24/javascript-programming-java-cryptocurrency/

[32] "Solidity," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.7.4/index.html

[33] "Solidity v0.5.0 Breaking Changes," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.7.4/050-breaking-changes.html

[34] "Solidity v0.6.0 Breaking Changes," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.7.4/060-breaking-changes.html

[35] "Solidity v0.7.0 Breaking Changes," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.7.4/070-breaking-changes.html

[36] "Layout of a Solidity Source File," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.4.0/layout-of-source-files.html

[37] "Expressions and Control Structures," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.4.0/control-structures.html

[38] "Contract ABI Specification," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.4.0/abi-spec.html

[39] "Frequently Asked Questions," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.4.0/frequently-asked-questions.html

[40] "Units and Globally Available Variables," accessed 14.11.2020. [Online]. Available: https://solidity.readthedocs.io/en/v0.4.0/units-and-global-variables.html

[41] "Understanding The DAO Attack," accessed 25.06.2016. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[42] "SWC Registry - Smart Contract Weakness Classification and Test Cases," accessed 14.11.2020. [Online]. Available: https://swcregistry.io/

[43] N. Lu, B. Wang, Y. Zhang, W. Shi, and C. Esposito, "NeuCheck: A more practical Ethereum smart contract security analysis tool," *Software: Practice and Experience*, 2019.

[44] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," *2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[45] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework For Smart Contracts," *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019.

[46] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection," *33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[47] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts," *29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[48] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts," *2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[49] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts." [Online]. Available: https://arxiv.org/abs/2004.08563

[50] "Modules," accessed 14.11.2020. [Online]. Available: https://github.com/ConsenSys/mythril/tree/0e827d1aa25d4007e3710a6a6c577e43565503e5/mythril/analysis/module/modules

[51] "Ethereum Detectors," accessed 14.11.2020. [Online]. Available: https://github.com/trailofbits/manticore/wiki/Ethereum-Detectors

[52] T. S. S. Analyzer, "Slither," accessed 14.11.2020. [Online]. Available: https://github.com/crytic/slither/tree/858677464cff0f61d1553ae2ae642963ee074153

[53] F. H. Vogel, "IRSRI - An Intermediate Representation for Smart Contracts," 2019. [Online]. Available: https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/343039/Vogel_Frederic.pdf?sequence=1&isAllowed=y

[54] "Datalog," accessed 01.06.2020. [Online]. Available: https://souffle-lang.github.io/datalog

[55] "Souffle - Logic Defined Static Analysis." accessed 14.11.2020. [Online]. Available: https://souffle-lang.github.io/

[56] "Static Single Assignment Form - Harvard SEAS," accessed 14.11.2020. [Online]. Available: www.seas.harvard.edu%2Fcourses%2Fcs252%2F2011sp%2Fslides%2FLec04-SSA.pdf&usg=AOvVaw0zpKFq2ZA811-ktgc0kbHQ

[57] P. Tsankov, "How can I test a byte-code contract file with Securify? 14," accessed 14.11.2020. [Online]. Available: https://github.com/eth-sri/securify2/issues/14#issuecomment-704333360

[58] "Echidna: A Fast Smart Contract Fuzzer," accessed 14.11.2020. [Online]. Available: https://github.com/crytic/echidna/tree/673aece0382e4da1a5ad1ac880b8a3bc9f231b92

[59] "ILF: AI-based Fuzzer for Ethereum Smart Contracts," accessed 14.11.2020. [Online]. Available: https://github.com/eth-sri/ilf/tree/9e8e3015a48783634658c8e748f113d2da2628c7

[60] A. Canoole, "Simple Fuzz," accessed 14.11.2020. [Online]. Available: https://github.com/foreni-packages/sfuzz/tree/6f6fd94279d4e9355832c0a321b589e6b9c6814e

[61] "[DEPRECATED] Securify," accessed 14.11.2020. [Online]. Available: https://github.com/eth-sri/securify/tree/51ba1240e60332b4a1e6ad02090da6fe57676354

[62] "NeuCheck," accessed 14.11.2020. [Online]. Available: https://github.com/Northeastern-University-Blockchain/NeuCheck/tree/8fa56662ca173d7ca7a748053a8bfec7e484ac5a

[63] "ContractFuzzer," accessed 14.11.2020. [Online]. Available: https://github.com/gongbell/ContractFuzzer/tree/5e5ad684d56599cbab85bfebe579f8e842e0f342

[64] "Lack of documentation on how to execute bytecode-only ethereum contracts," accessed 14.11.2020. [Online]. Available: https://github.com/trailofbits/manticore/issues/841

[65] "Manticore," accessed 14.11.2020. [Online]. Available: https://github.com/trailofbits/manticore/tree/2152023f390a7dc45cf16cd5cc283d0cfa93b8ee

[66] "Mythril," accessed 14.11.2020. [Online]. Available: https://github.com/ConsenSys/mythril/tree/0e827d1aa25d4007e3710a6a6c577e43565503e5

[67] "Osiris," accessed 14.11.2020. [Online]. Available: https://github.com/christoftorres/Osiris/tree/f08c3c1772f4638b1ff3401c43a323c61e5325f6

[68] "Oyente," accessed 14.11.2020. [Online]. Available: https://github.com/melonproject/oyente/tree/1e0a00274ecf618206e2fec0636900017e3568c9

[69] "cryptomental/maian-augur-ci:latest," accessed 14.11.2020. [Online]. Available: https://hub.docker.com/layers/cryptomental/maian-augur-ci/latest/images/sha256-e838a693f05fded4d8c57758307632090d48d1d0b097057cebaf182479a95aa0

[70] "Vandal," accessed 14.11.2020. [Online]. Available: https://github.com/usyd-blockchain/vandal/tree/d2b004326fee33920c313e64d0970410b1933990

[71] "Install Soufflé," accessed 14.11.2020. [Online]. Available: https://souffle-lang.github.io/install

[72] "SmartCheck," accessed 14.11.2020. [Online]. Available: https://github.com/smartdec/smartcheck/tree/eb3a5b9106160e29e0eb98c5f28edfa3bf6a4a30

[73] P. Tsankov, "solidity_ast_compiler.py," accessed 14.11.2020. [Online]. Available: https://github.com/eth-sri/securify2/blob/71c22dd3d6fc74fb87ed4c4118710642a0d6707e/securify/solidity/solidity_ast_compiler.py

[74] "Releases," accessed 14.11.2020. [Online]. Available: https://github.com/ethereum/solidity/releases

[75] "Etherscan," accessed 14.11.2020. [Online]. Available: https://etherscan.io/

[76] "trailofbits/manticore:0.3.4," accessed 14.11.2020. [Online]. Available: https://hub.docker.com/layers/trailofbits/manticore/0.3.4/images/sha256-f3bc748fc8d09677f7ce7673240dc2a51cb0218cff4e9c8f0d27ca194666432c

[77] "mythril/myth:0.22.4," accessed 14.11.2020. [Online]. Available: https://hub.docker.com/layers/mythril/myth/0.22.4/images/sha256-d4ff43213812b49e275fc5ddd2bf83d882a91bd2ed220d52f169ca93b44d6452

[78] "christoftorres/osiris:latest," accessed 14.11.2020. [Online]. Available: https://hub.docker.com/layers/christoftorres/osiris/latest/images/sha256-d80ab2b4ed96c1e55fe68dc30c1a61b85698628409889f9370f0b9712da785b7

[79] "luongnguyen/oyente:latest," accessed 14.11.2020. [Online]. Available: https://hub.docker.com/layers/luongnguyen/oyente/latest/images/sha256-16c74d21e997cffe9a7480cb140ee6c05b7ba608e1b878df90f585b4710e41b0

[80] "SmartCheck," accessed 14.11.2020. [Online]. Available: https://www.npmjs.com/package/@smartdec/smartcheck/v/2.0.1

[81] "Object Relational Tutorial," accessed 14.11.2020. [Online]. Available: https://docs.sqlalchemy.org/en/13/orm/tutorial.html

[82] "SQLite," accessed 14.11.2020. [Online]. Available: https://sqlite.org/index.html

[83] "Basic Relationship Patterns," accessed 14.11.2020. [Online]. Available: https://docs.sqlalchemy.org/en/13/orm/basic_relationships.html

[84] "Inheritance Configuration," accessed 14.11.2020. [Online]. Available: https://docs.sqlalchemy.org/en/13/orm/extensions/declarative/inheritance.html

[85] "Constructors and Object Initialization," accessed 14.11.2020. [Online]. Available: https://docs.sqlalchemy.org/en/13/orm/constructors.html

[86] "Mythril Detection Capabilities," accessed 14.11.2020. [Online]. Available: https://github.com/ConsenSys/mythril/wiki/Mythril-Detection-Capabilities

[87] "demo_analyses.dl," accessed 14.11.2020. [Online]. Available: https://github.com/usyd-blockchain/vandal/blob/d2b004326fee33920c313e64d0970410b1933990/datalog/demo_analyses.dl

[88] "rule_descriptions," accessed 14.11.2020. [Online]. Available: https://github.com/smartdec/smartcheck/tree/3a47fb3b550b996e6beb2be51dbcde1d23f44a92/rule_descriptions

[89] "patterns," accessed: 14.11.2020. [Online]. Available: https://github.com/eth-sri/securify2/tree/master/securify/staticanalysis/souffle_analysis/patterns

[90] "Bulk analyser," accessed 14.11.2020. [Online]. Available: https://github.com/usyd-blockchain/vandal/blob/d2b004326fee33920c313e64d0970410b1933990/tools/bulk_analyser/README.md

[91] "5. The import system," accessed 14.11.2020. [Online]. Available: https://docs.python.org/3/reference/import.html

[92] Private and Protected, "Python - public," accessed 14.11.2020. [Online]. Available: https://www.tutorialsteacher.com/python/private-and-protected-access-modifiers-in-python

[93] "Maian," accessed 14.11.2020. [Online]. Available: https://github.com/ivicanikolicsg/MAIAN/tree/ab387e171bf99969676e60a0e5c59de80af15010

[94] "docker run," accessed 14.11.2020. [Online]. Available: https://docs.docker.com/engine/reference/commandline/run/

[95] "bash," accessed 14.11.2020. [Online]. Available: https://manpages.ubuntu.com/manpages/xenial/man1/bash.1.html

[96] "Quickstart," accessed 14.11.2020. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/quickstart/

[97] "API," accessed 14.11.2020. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/api/

[98] "Templates," accessed 14.11.2020. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/tutorial/templates/

[99] "form," accessed 14.11.2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form

[100] "argparse — Parser for command-line options, arguments and sub-commands," accessed 14.11.2020. [Online]. Available: https://docs.python.org/3.8/library/argparse.html

[101] "Ethereum Developer APIs," accessed 14.11.2020. [Online]. Available: https://etherscan.io/apis

[102] "difflib — Helpers for computing deltas," accessed 14.11.2020. [Online]. Available: https://docs.python.org/3/library/difflib.html

[103] "Zlib," accessed 14.11.2020. [Online]. Available: https://www.zlib.net/

[104] "Which of the rules in the rule description cover a reentrancy vulnerability?#34," accessed 14.11.2020. [Online]. Available: https://github.com/smartdec/smartcheck/issues/34#issuecomment-712954516

[105] R. M. Parizi, A. Dehghantanha, K. K. R. Choo, and A. Singh, "Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains," *28th Annual International Conference on Computer Science and Software Engineering (CASCON18)*, 2018.

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Augsburg, 16. November 2020

. . . .
(Lukas Denk)