Smart Contract Vulnerabilities: Does Anyone Care?

Daniel Perez
Imperial College London

Benjamin Livshits
Imperial College London

Abstract

In the last year we have seen a great deal of both academic and practical interest in the topic of vulnerabilities in smart contracts, particularly those developed for the Ethereum blockchain. In this paper we survey the 21,270 vulnerable contracts reported by five recent academic projects. Contrary to what might have been believed given the reported number of vulnerable contracts, there has been precious little in terms of *actual exploitation* when it comes to these vulnerabilities. We find that at most 504 out of 21,270 contracts have been subjected to exploits. This corresponds to at most 9,094 ETH (1 million USD [1]), or only 0.30% of the 3 million ETH (350 million USD) claimed in some of the papers. While we are certainly not implying that smart contract vulnerability research is without merit, our results suggest that the potential impact of vulnerable code had been greatly exaggerated.

1 Introduction

When it comes to vulnerability research, especially as it pertains to software security, it is frequently difficult to estimate what fraction of discovered or reported vulnerabilities are exploited in practice. However, public *blockchains*, with their immutability, ease of access, and what amounts to a replayable execution log for smart contracts present an excellent opportunity for just such an investigation. In this work we aim to contrast the vulnerabilities that are reported in smart contracts on the Ethereum [24] blockchain with the actual exploitation of these contracts.

We collect the data shared with us by the authors of five recent papers [33, 40, 42, 46, 52] that focus on finding smart contract vulnerabilities. These academic datasets are significantly bigger in scale than reports we can find in the wild and because of the sheer number of affected contracts — 21,270 — represent an excellent study subject.

[1] We use the exchange rate on 2019-01-15: 1 ETH = 115 USD. For consistency, any monetary amounts denominated in USD are based on this rate.

To make our approach more general, we express five different frequently reported vulnerability classes as Datalog queries computed over relations that represent the state of the Ethereum blockchain, both current and historic. The Datalog-based exploit discovery approach gives more scalability to our process; also, while others have used Datalog for static analysis formulation, we are not aware of it being used to capture the dynamic state of the blockchain over time.

We discover that the amount of smart contract exploitation which occurs in the wild is notably lower than what might be believed, given what is suggested by the sometimes sensational nature of some of the famous crypto-currency exploits such as TheDAO [49] or the Parity wallet [22] bugs.

Contributions. Our contributions are:

- This paper presents the first broadly scoped analysis of the real-life prominence of security exploits against smart contracts.
- We propose a Datalog-based formulation for performing analysis over Ethereum Virtual Machine (EVM) execution traces. We use this highly scalable approach to analyze a total of more than 16 million transactions from the Ethereum blockchain to search for exploits.
- We analyze the vulnerabilities reported in five recently published studies and conclude that, although the number of contracts and the amount of money supposedly at risk is very high, the amount of money which has actually been exploited is several orders of magnitude lower.
- We discover out of 21,270 vulnerable contracts worth a total of 3,088,102 ETH, merely 49 contracts containing Ether may have been exploited for an amount of 9,094 ETH, which represents as little as 0.30% of the total amount at stake.
- We hypothesize that the reasons for these vast differences are multifold: lack of appetite for exploitation, the sheer difficulty of executing some exploits, fear of attribution, other more attractive exploitation options, etc. Further analysis of the vulnerable contracts and the Ether they contain suggests that a large majority of Ether is

Name	Contracts analyzed	Issues found	Vulnerabilities RE	UE	LE	TO	IO	Report month	Citation
Oyente	19K	8.8K	✓	✓		✓		2016-10	[42]
ZEUS	22.4K	21K	✓	✓	✓	✓	✓	2018-02	[40]
Maian	34K	3.7K			✓			2018-03	[46]
SmartCheck	4.6K	4.6K	✓	✓	✓		✓	2018-05	[51]
Securify	25K	5K	✓	✓	✓	✓		2018-06	[52]
ContractFuzzer	7K	460	✓	✓				2018-09	[39]
Vandal	141K	85K	✓	✓				2018-09	[23]
MadMax	92K	6K			✓		✓	2018-10	[33]

Figure 1: A summary of smart contract analysis tools presented in prior work.

held by only a small number of contracts, and that the vulnerabilities reported on these contracts are either false positives or not applicable in practice, making exploitation significantly less attractive as a goal.

2 Background

The Ethereum [24] platform allows its users to run "smart contracts" on its distributed infrastructure. Ethereum *smart contracts* are programs which define a set of rules for the governing of associated funds, typically written in a Turing-complete programming language called Solidity [30]. Solidity is similar to JavaScript, yet some notable differences are that it is strongly-typed and has built-in constructs to interact with the Ethereum platform. Programs written in Solidity are compiled into low-level untyped bytecode to be executed on the Ethereum platform by the Ethereum Virtual Machine (EVM). It is important to note that it is also possible to write EVM contracts without using Solidity.

To execute a smart contract, a sender has to send a transaction to the contract and pay a fee which is derived from the contract's computational cost, measured in units of so-called *gas*. Consumed gas is credited to the miner of the block containing the transaction, while any unused gas is refunded to the sender. In order to avoid system failure stemming from never-terminating programs, transactions specify a gas limit for contract execution. An out-of-gas exception is thrown once this limit has been reached.

Smart contracts themselves have the capability to "call" another account present on the Ethereum blockchain. This functionality is somehow overloaded, as it is used both to call a function in another contract and to send Ether (ETH), the underlying currency in Ethereum, to an account. A particularity of how this works in Ethereum is that calls from within a contract do not create any new transactions and are therefore not directly recorded on-chain. This means that merely looking at the transactions without executing them does not provide enough information to follow the flow of Ether.

2.1 Current Industrial Practice

Smart contracts are generally designed to manipulate and *hold* funds denominated in Ether. This makes them very tempting attack targets, as a successful attack may allow the attacker to directly steal funds from the contract. Hence, in order to ensure a sufficient degree of smart contract security, a wide variety of practices that operate at different stages of the development life-cycle have been adopted in the industry.

Analysis tools. A large number of tools have been developed to analyze smart contracts [28, 42, 52]. Most of these tools analyze either the contract source code or its compiled EVM bytecode and look for known security issues, such as re-entrancy or transaction order dependency vulnerabilities. We present a summary of these different works in Figure 1. The second and third columns respectively present the reported number of contracts analyzed and contracts flagged vulnerable in each paper. The "vulnerabilities" columns show the type of vulnerabilities that each tool can check for. We present these vulnerabilities in Subsection 2.2 and give a more detailed description of these tools in Section 7.

Testing. Like any piece of software, smart contracts benefit from automated testing and some efforts have therefore been made to make the testing experience more straightforward. Truffle [27] is a popular framework for developing smart contracts, which allows to write both unit and integration tests for smart contracts in JavaScript. One difficulty of testing on the Ethereum platform is that the EVM does not have a single main entry point and bytecode is executed when fulfilling a transaction.

There are mainly two methods used to work around this. The first is to use a private Ethereum network, or a test-net, where it is easy to control the state. The smart contracts are deployed and executed on the private network in the same way they would be deployed on the main Ethereum network. The other approach is to use a standalone implementation of the EVM. Ganache [29] is one of the most popular such standalone implementation of the EVM built for development purposes and is developed by the same authors as Truffle. Although this provides a more lightweight way to run tests, it also requires the implementation to perfectly mimic the original one, which is error-prone.

Auditing. As smarts contracts can have a high monetary value, *auditing* contracts for vulnerabilities is a common industrial practice. Audit should preferably be performed while contracts are still in testing phase but given the relatively high cost of auditing (usually around 30,000 to 40,000 USD [14]) some companies choose to perform audit later in their development cycle. In addition to checking for common vulnerabilities and implementation issues such as gas consuming operations, audits also usually check for divergences from the whitepaper and other high-level logic errors, which are impossible for current automatic tools to detect.

Bounty programs. Another common practice for develop-

ers to improve the security of their smart contracts is to run bounty programs. While auditing is usually a one-time process, bounty programs remain ongoing throughout a contract's lifetime and allow community members to be rewarded for reporting vulnerabilities. Companies or projects running bounty programs can either choose to reward the contributors by paying them in a fiat currency such as US dollars, a cryptocurrency — typically Bitcoin or Ether — or their own crypto token. Some bounty programs, such as the one run by the 0x project [1], offer bounties as high as 100,000 USD for critical vulnerabilities.

Contract upgrades. In Ethereum, smart contracts are by nature immutable. Once a contract has been deployed on the blockchain, its code cannot be modified. This creates a challenge during the deployment of smart contracts, as upgrading the code requires to work around this limitation. There are several approaches to deploy a new version of a smart contract [7].

The first approach is to use a *registry contract* which returns the address of the latest version of a smart contract. When deploying a contract, the contract with the updated version of the code is deployed and the address of the latest version stored in the registry is updated. Although this leaves a lot of flexibility to the developers, it forces the users of the smart contracts to always query the registry before being able to interact with the contract. To avoid adding overhead to the user of the contract, an alternative approach is to use a *facade contract*. In this approach, a contract with a fixed address is deployed but delegates all the calls to another contract, the address of which can be updated [25]. The end-user of the contract can therefore always transact with the same contract, while the developers are able to update the behavior of the contract by deploying a new contract and updating the facade to delegate to the newly deployed code.

There are two main drawbacks with this approach. One of the drawbacks of this approach is that developers cannot modify the contract interface, as the facade code does not change. The other is that there is a gas cost overhead, as the facade contract uses gas to call the backend contract.

2.2 Vulnerability types

In this subsection, we briefly review some of the most common vulnerability types that have been researched and reported for EVM-based smart contracts. We provide a two-letter abbreviation for each vulnerability, which we shall use throughout the remainder of this paper.

Re-entrancy (RE). The vulnerability is exploited when a contract tries to send Ether before having updated its internal state. If the destination address is another contract, it will be executed and can therefore call the function to request Ether again and again [40, 42, 52]. This vulnerability has been used in TheDAO exploit [49], essentially causing the Ethereum community to decide to rollback to a previous state using a so-called hard-fork [44].

Unhandled exceptions (UE). Some low-level operations in Solidity such as `send`, which is used to send Ether, do not throw an exception on failure, but rather report the status by returning a boolean. If this return value were to be unchecked, a contract would continue its execution even if the payment failed, which could be easily lead to inconsistencies [23, 40, 42, 51].

Locked Ether (LE). Ethereum smart contracts can have a function marked as `payable` which allows it to receive Ether, increasing the balance of the contract. Most of the times, the contract will also have a function which sends Ether. For example, a contract might have a `payable` function called `deposit`, which receives Ether, and a function called `withdraw`, which sends Ether. However, there are several reasons for which the `withdraw` function may become unable to send funds anymore.

One reason is that the contract may depend on another contract, which has been destructed using the `SELFDESTRUCT` instruction of the EVM — i.e. its code has been removed and its funds transferred. For example, the `withdraw` function may require an external contract to send Ether. However, if the contract it relies on has been destructed, the `withdraw` function would not be able to actually send the Ether, effectively locking the funds of the contract. This is what happened in the Parity Wallet bug in November 2017, locking millions of USD worth of Ether [22].

There are also cases where the contract will *always* run out of gas when trying to send Ether, locking the contract funds. More details about such issues can be found in [33].

Transaction Order Dependency (TO). In Ethereum, multiple transactions are included in a single block, which means that the state of a contract can be updated multiple times in the same block. If the order of two transactions calling the same smart contract changes the final outcome, an attacker could exploit this property. For example, given a contract which expects participant to submit the solution to a puzzle in exchange for a reward, a malicious contract owner could reduce the amount of the reward when the transaction is submitted.

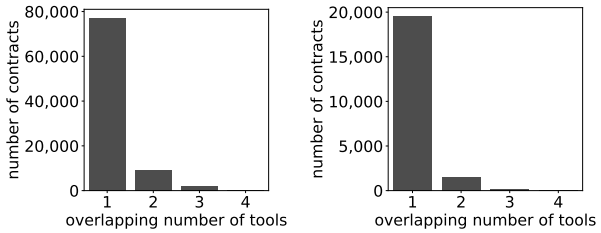
Integer overflow (IO). Integer overflow and underflow is a common type of bug in many programming languages but in the context of Ethereum it can have very severe consequences. For example, if a loop counter were to overflow, creating an infinite loop, the funds of a contract could become completely frozen. This can be exploited by an attacker if he has a way of incrementing the number of iterations of the loop, for example, by registering new users of the contract.

3 Dataset

In this paper, we analyze the vulnerable contracts reported by the following five academic papers: [42], [40], [46], [52]

Name	Contracts analyzed	Vulnerabilities found	Ether at stake at time of report
Oyente	19,366	7,527	1,289,177
Zeus	1,120	855	729,376
Maian	NA	2,691	14.13
Securify	29,694	9,185	719,567
MadMax	91,800	6,039	1,114,692

Figure 2: Summary of the contracts in our dataset.



(a) Overlapping contracts analyzed.

(b) Overlapping vulnerabilities flagged.

Figure 3: Histograms that show the overlap in the contracts analyzed and flagged by different papers.

and [33]. To collect information about the addresses analyzed and the vulnerabilities found, we reached out to the authors of the different papers.

Oyente [42] data was publicly available [13]. The authors of the other papers were kind enough to provide us with their dataset. We received all the replies within less than a week of contacting the authors.

We also reached out to the authors of [51], [39] and [23] but could not obtain their dataset, which is why we left these papers out of our analysis.

Our dataset is comprised of a total of 110,177 contracts analyzed, of which 21,270 contracts have been flagged as vulnerable to at least one of the five vulnerabilities described in Section 2. For most papers, the numbers of contracts analyzed and the number of vulnerable contracts found did not match the numbers reported in the papers, which we reported in Figure 1. Therefore, we present the numbers in our dataset, as well as the Ether at stake for vulnerable contracts in Figure 2. The Ether at stake is computed by summing the balance of all the contracts flagged vulnerable. We use the balance at the time at which each paper was published rather than the current one, as it gives a better sense of the amount of Ether which could potentially have been exploited.

Taxonomy. Rather than reusing existing smart contracts vulnerabilities taxonomies [19] as-is, we adapt it to fit the vulnerabilities analyzed by the different tools in our dataset. We do not cover any vulnerability which is not analyzed by at least two of the five tools we analyze. We settle on the five types of vulnerabilities described in Section 2: re-entrancy (RE), unhandled exception (UE), locked Ether (LE), transaction order dependency (TO) and integer overflows (IO). As the papers we analyze use different terms and slightly different definitions for each of these vulnerabilities, we map the relevant vulnerability to one of the five types of vulnerabilities we analyze. We show how we mapped these vulnerabilities in Figure 5.

Tools	Total	Agreed	Disagreed	% agreement
Oyente/Securify	774	185	589	23.9%
Oyente/Zeus	104	3	101	0.029%
Zeus/Securify	108	2	106	0.019%

Figure 4: Agreement among tools for re-entrancy analysis.

Overlapping vulnerabilities. In this subsection, we analyze our dataset to see how much the contracts analyzed by the different tools and the vulnerabilities found overlap. Although most papers, except for [42], are written around the same period, we find that only 13,751 contracts out of the total of 110,177 have been analyzed by at least two of the tools. In Figure 3a, we show a histogram of how many different tools analyze a single contract. In Figure 3b, we show the number of tools which flag a single contract as vulnerable to any of the analyzed vulnerability. The overlap for both the analyzed and the vulnerable contracts is clearly very small. We assume one of the reasons is that some tools work on Solidity code [40] while other tools work on EVM bytecode [42, 52], making the population of contracts available different among tools.

We also find a lot of contradiction in the analysis of the different tools. We choose re-entrancy to illustrate this point, as it is unambiguous and is supported by three of the tools we analyze. In Figure 4, we show the agreement between the three tools which support re-entrancy detection. The *Total* column shows the total number of contracts analyzed by both tools in the *Tools* column and flagged by at least one of them as vulnerable to re-entrancy. Oyente and Securify agree on only 23% of the contracts, while Zeus does not seem to agree with any of the other tools. This reflects the difficulty of building static analysis tools targeted at the EVM. While we are not trying to evaluate or compare the performance of the different tools, this gives us yet another motivation to find out the impact of the reported vulnerabilities.

4 Methodology

In this section, we describe in details the different analysis we perform in order to check for exploits of the vulnerabilities described in Section 2.

To check for potential exploits, we mainly perform bytecode-level transaction analysis, whereby we look at the code executed by the contract when carrying out a particular transaction. We use this type of analysis to detect re-entrancy (RE), unhandled exceptions (UE) and integer overflows (IO). We also use transaction pattern analysis, where we analyze the flow of the transactions both from and to the contract, either to filter or to refine our results. We use this type of

	Oyente	ZEUS	Maian	Securify	MadMax
RE	re-entrancy	re-entrancy	—	no writes after call	—
UE	callstack	unchecked send	—	handled exceptions	—
TO	concurrency	tx order dependency	—	transaction ordering dependency	—
LE	—	failed send	greedy contracts	Ether liquidity	unbounded mass operation wallet griefing
IO	—	integer overflow	—	—	integer overflows

Figure 5: Mapping of the different vulnerabilities analyzed.

```
[
    {"op": "EQ", "pc": 7,
    "depth": 1, "stack": ["2b", "a3"]},
    {"op": "ISZERO", "pc": 8, "depth": 1,
    "stack": ["00"]}
]
```

Figure 6: Sample execution trace information.

analysis to refine the results of locked Ether (LE) and to filter transactions when looking for transaction order dependency (TO) exploits.

To perform both analysis, we first retrieve transaction data for all the contracts in our dataset. To simplify the retrieval process, we use data provided by Etherscan [8], a well-known Ethereum blockchain explorer service, rather than scanning the entire Ethereum blockchain ourselves.

Next, to perform bytecode-level analysis, we extract the execution traces for the transactions which may have affected contracts of interest. We use EVM's debug functionality, which gives us the ability to replay transactions and to trace all the executed instructions. To speed-up the data collection process, we patch the Go Ethereum client [12], opposed to relying on the Remote Procedure Call (RPC) functionality provided by the default Ethereum client.

The extracted traces contain a list of executed instructions, as well as the state of the stack at each instruction. We show a truncated sample of the extracted traces in Figure 6 for illustration. The `op` key is the current instruction, `pc` is the program counter, `depth` is the current level of call nesting, and finally, `stack` contains the current state of the stack. We use single-byte values in the example, but the actual values are 32 bytes (256 bits).

To analyze the traces, we encode them into a Datalog representation; Datalog is a language implementing first-order logic with recursion [38], which has been used extensively by the programming language community. We use the following domains to encode the information about the traces as Datalog facts:

- V is the set of program variables;
- A is the set of Ethereum addresses;
- $\mathbb{N}$ is the set of natural numbers, $\mathbb{Z}$ is the set of integers.

We show an overview of the facts that we collect and the relations that we use to check for possible exploits in Figure 8.

```
if (!addr.send(100)) { throw; }
```

(a) Failure handling in Solidity.

```
; preparing call
(0x65) CALL
; call result pushed on the stack
(0x69) PUSH1 0x73
(0x71) JUMPI ; jump to 0x73 if call was successful
(0x72) REVERT
(0x73) JUMPDEST
```

(b) EVM instructions for failure handling.
Figure 7: Correctly handled failed `send`.

Re-entrancy. In the EVM, as transactions are executed independently, re-entrancy issues can only occur *within* a single transaction. Therefore, for re-entrancy to be exploited, there must be a call to an external contract, which invokes, directly or indirectly, a re-entrant callback to the calling contract. We therefore start by looking for `CALL` instructions in the execution traces, while keeping track of the contract currently being executed.

When `CALL` is executed, the address of the contract to be called as well as the value to be sent can be retrieved by inspecting the values on the stack [54]. Using this information, we can record $\texttt{direct\_call}(a_1, a_2, p)$ facts described in Figure 8a. Using these, we then use the query shown in Figure 8c to retrieve potentially malicious re-entrant calls.

Unhandled exceptions. When Solidity compiles contracts, methods to send Ether, such as `send`, are compiled into the EVM `CALL` instructions. We show an example of such a call and its instructions counterpart in Figure 7. If the address passed to `CALL` is an address, the EVM executes the code of the contract, otherwise it executes the necessary instructions to transfer Ether to the address. When the EVM is done executing, it pushes either 1 on the stack, if the `CALL` succeeded, or 0 otherwise.

To retrieve information about call results, we can therefore check for `CALL` instructions and use the value pushed on the stack after the call execution. The end of the call execution can be easily found by checking when the `depth` of the trace turns back to the value it had when the `CALL` instruction was executed; we save this information as $\texttt{call\_result}(v, n)$ facts.

As shown in Figure 7b, the EVM uses the `JUMPI` instruc-

Fact	Description
`is_output`($v_1 \in V$, $v_2 \in V$)	v_1 is an output of v_2
`size`($v \in V$, $n \in \mathbb{N}$)	v has n bits
`is_signed`($v \in V$)	v is signed
`in_condition`($v \in V$)	v is used in a condition
`call`($a_1 \in A$, $a_2 \in A$, $p \in \mathbb{N}$)	a_1 calls a_2 with p Ether
`expected_result`($v \in V$, $r \in \mathbb{Z}$)	v's expected result is r
`actual_result`($v \in V$, $r \in \mathbb{Z}$)	v's actual result is r
`call_result`($v \in V$, $n \in \mathbb{N}$)	v is the result of a call and has a value of n
`call_entry`($i \in \mathbb{N}$, $a \in A$)	contract a is called when program counter is i
`call_exit`($i \in \mathbb{N}$)	program counter is i when exiting a call to a contract
`tx_sstore`($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$)	storage key k is written in transaction i of block b
`tx_sload`($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$)	storage key k is read in transaction i of block b

(a) Datalog facts.

Datalog rules
`depends`($v_1 \in V$, $v_2 \in V$) `:-` `is_output`(v_1, v_2). `depends`(v_1, v_2) `:-` `is_output`(v_1, v_3), `depends`(v_3, v_2).
`call_flow`($a_1 \in A$, $a_2 \in A$, $p \in \mathbb{Z}$) `:-` `call`(a_1, a_2, p). `call_flow`(a_1, a_2, p) `:-` `call`(a_1, a_3, p), `call_flow`(a_3, a_2, _).
`inferred_size`($v \in V$, $n \in \mathbb{N}$) `:-` `size`(v, n). `inferred_size`(v, n) `:-` `depends`(v, v_2), `size`(v_2, n).
`inferred_signed`($v \in V$) `:-` `is_signed`(v). `inferred_signed`(v) `:-` `depends`(v, v_2), `is_signed`(v_2).
`influences_condition`($v \in V$) `:-` `in_condition`(v). `influences_condition`(v) `:-` `depends`(v_2, v), `in_condition`(v_2).

(b) Basic Datalog relation definitions.

Type of vulnerability	Query
Re-entrancy	`call_flow`(a_1, a_2, p_1), `call_flow`(a_2, a_1, p_2), $a_1 \neq a_2$
Unhandled exceptions	`call_result`(v, 0), $\neg$`influences_condition`(v)
Transaction Order Dependency	`tx_sstore`(b, t_1, i), `tx_sload`(b, t_2, i), $t_1 \neq t_2$
Locked Ether	`call_entry`(i_1, a), `call_exit`(i_2), $i_1 + 1 = i_2$
Integer overflow	`actual_result`(v, r_1), `expected_result`(v, r_2), $r_1 \neq r_2$

(c) Datalog queries for detecting different vulnerability classes.

Figure 8: Datalog setup.

tion to perform conditional jumps. At the time of writing, this is the only instruction available to execute conditional control flow. We therefore mark all the values used as a condition in `JUMPI` as `in_condition`. We can then check for the unhandled exceptions by looking for call results, which never influence a condition using the query shown in Figure 8c.

Locked Ether. Although there are several reasons for funds locked in a contract, we focus on the case where the contract relies on an external contract, which does not exist anymore, as this is the pattern which had the largest financial impact on Ethereum [22]. Such a case can occur when a contract uses another contract as a library to perform some actions on its behalf. To use a contract in this way, the `DELEGATECALL` instruction is used instead of the `CALL`, as the latter does not preserve call data, such as the sender or the value.

The next important part is the behavior of the EVM when trying to call a contract which does not exist anymore. When a contract is destructed, it is not completely removed per-se, but its code is removed. When a contract tries to call a contract which has been destructed, the call is a no-op rather than a failure, which means that the next instruction will be executed and the call will be marked as successful. To find such patterns, we first mark the index at which the call is executed — `call_entry`($i_1 \in \mathbb{N}$, $a \in A$). Then, using the same approach as for Unhandled exceptions, we skip the content of the call and mark the index at which the call returns — `call_exit`($i_2 \in \mathbb{N}$).

If the contract called does not exist anymore, $i_1 + 1 = i_2$ must hold. Therefore, we can use the query shown in Figure 8c to retrieve the address of the destructed contracts, if any.

Transaction Order Dependency. The first insight to check for exploitation of transaction ordering dependency is that at least two transactions to the same contract must be included in the same block for such an attack to be successful. Furthermore, as shown in [42] or [52], exploiting a transaction ordering dependency vulnerability requires manipulation of the contract's storage.

The EVM has only one instruction to read from the storage, `SLOAD`, and one instruction to write the storage, `SSTORE`. The key to which the EVM reads or writes to is available on the stack when the instruction is called. We go through all the transactions of the contracts and each time we encounter one of these instructions, we record either `tx_sload`($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$) or `tx_sstore`($b \in \mathbb{N}, i \in \mathbb{N}, k \in \mathbb{N}$) where in each case b is the block number, i is the index of the transaction in the block and k is the storage key being accessed.

The essence of the rule to check for transaction order dependency issues is then to look for patterns where at least two transactions are included in the same block with one of the transactions writing a key in the storage and another transaction reading the same key. We show the actual rule in Figure 8c.

Instruction	Description
SIGNEXTEND	Increase the number of bits
SLT	Signed lower than
SGT	Signed greater than
SDIV	Signed division
SMOD	Signed modulo

Figure 9: EVM instructions that operate on signed operands.

Integer overflow. The EVM is completely untyped and expresses everything in terms of 256-bits words. Therefore, types are handled completely at the compilation level, and there is no explicit information about the original types in any execution traces.

To check for integer overflow, we accumulate facts over two passes. In the first pass, we try to recover the sign and size of the different values on the stack. To do so, we use known invariants about the Solidity compilation process. First, any value which is used as an operand of one of the instructions shown in Figure 9 can be marked to be signed with `is_signed`(v). Furthermore, `SIGNEXTEND` being the usual sign extension operation for two's complement, it is passed both the value to extend and the number of bits of the value. This allows to retrieve the size of the signed value. We assume any value not explicitly marked as signed to be unsigned. To retrieve the size of unsigned values, we use another behavior of the Solidity compiler.

To work around the lack of type in the EVM, the Solidity compiler inserts an `AND` instruction to "cast" unsigned integers to their correct value. For example, to emulate an `uint8`, the compiler inserts `AND value 0xff`. In the case of a "cast", the second operand m will always be of the form $m = 16^n - 1$, $n \in \mathbb{N}$, $n = 2^p$, $p \in [1,6]$. We use this observation to mark values with the according type: `uintN` where $N = n \times 4$. Variables size are stored as `size`(v, n) facts.

During the second phase, we use the `inferred_signed`(v) and `inferred_size`(v, n) rules shown in Figure 8b to retrieve information about the current variable. When no information about the size can be inferred, we over-approximate it to 256 bits, the size of an EVM word. Using this information, we compute the expected value for all arithmetic instructions (e.g. `ADD`, `MUL`), as well as the actual result computed by the EVM and store them as Datalog facts. Finally, we use the query shown in Figure 8c to find instructions which overflow.

5 Analysis of Individual Vulnerabilities

As described in Section 3, the combined amount of Ether contained within *all* the flagged contracts exceeds 3 million ETH, worth 345 million USD. In this section, we present the results for each vulnerability, one by one; our results are have been obtained using the methodology described in Section 4; the goal is to show how much of this money is actually at risk.

Contract address	Last transaction	Amount exploited
0xd654bdd32fc99471455e86c2e7f7d7b6437e9179	2016-06-10	5,885
0x675e2c143295b8683b5aed421329c4df85f91b33	2015-12-31	50.49
0x02b2101903eb6a51518e63e84b785180859fda9d	2016-04-10	43.41
0x8dd7693ead649aa369bc188049112fd7486a3b6b	2015-11-09	28.59
0x233820087a752349ee20daab1c18e0b7c546d3f6	2016-05-15	13.38
0xcd3e727275bc2f511822dc9a26bd7b0bbf161784	2017-03-25	10.34

Figure 10: RE: Contracts victim of re-entrancy attack.

5.1 RE: Re-entrancy

There is a total of 4,336 contracts flagged as vulnerable to re-entrancy by [40, 42, 52], with a total of 411,604 transactions.

However, a common pattern when exploiting re-entrancy is to first send a transaction to an attacker contract, which will in turn call the vulnerable contract. In such a case, the vulnerable contract is called in a transaction sent to another contract and there is no direct call to the vulnerable contract recorded on the blockchain. Therefore, we separately retrieve all the transactions which indirectly call the contracts to avoid missing any re-entrant call. This results in 34,742 more transactions, for a total of 446,346 transactions to analyze.

After running the analysis described in Section 4 on all the transactions, we found a total of 113 contracts, which contain re-entrant calls. Most of these calls are performed on toy contracts and send almost no Ether at all. To reduce the noise, we looked for contracts where at least 10 ETH has been lost, which represents 1,150 USD. To find the amount of Ether lost, we compute the sum of the Ether sent between two contracts in transactions containing re-entrant calls.

This gives us a total of *only* 6 potential contracts at risk, as shown in Figure 10. Interestingly, one of these five potential attacks has a substantial amount of Ether at stake: 5,881 ETH, which corresponds to around 670,000 USD. This address has already been detected as vulnerable by some recent work focusing on re-entrancy [47]. It appears that the contract, which is part of the Maker DAO [10] platform, was found vulnerable by the authors of the contract, who themselves performed an attack to confirm the risk [4].

Sanity checking. As a sanity check, we also add to our set of analyzed contracts the address of a contract called SpankChain [18], which is known to recently have been compromised by a re-entrancy attack. We confirm that our approach successfully marks this contract as having been the victim of a re-entrancy attack and correctly identifies the attacker contract.

5.2 UE: Unhandled exceptions

There is a total of 11,426 contracts flagged vulnerable to unhandled exceptions by [40, 42, 52]. This is a total of more than 3 million transactions, which is *an order of magnitude* larger than what we found for re-entrancy issues.

Contract address	Balance
0xfb6916095ca1df60bb79ce92ce3ea74c37c5d359	3,270
0x7011f3edc7fa43c81440f9f43a6458174113b162	69.30
0x192e16ba79baa5a043a301c6b22174f6049202c6	65.28
0xa9e4e3b1da2462752aea980698c335e70e9ab26c	36.98
0xfd2487cc0e5dce97f08be1bc8ef1dce8d5988b4d	32.44
0x2faa316fc4624ec39adc2ef7b5301124cfb68777	10.21

Figure 11: UE: Contracts affected by unhandled exceptions.

We find a total of 268 contracts where failed calls have not been checked for, which represent roughly 2% of the flagged contracts. We then analyze how many contracts with unhandled exceptions still hold Ether. We find a total of 6 contracts which currently hold more than 10 ETH. We summarize their addresses and balances in Figure 11.

Investigation of the contract at
0xfb6916095ca1df60bb79ce92ce3ea74c37c5d359:

The contract 0xfb6916095ca1df60bb79ce92ce3ea74c37c5d359 contains 3,270 ETH. As this is a fairly large amount, we manually inspect the issue further. While manually inspecting the traces, we find that the contract is trying to execute the CALL instruction by sending 0 ETH to the address 0x0000000000000000000000000000000000000004. We also inspected the Solidity code of the contract, but could not find what was causing the behavior seen at the EVM level.

We then look at the contract's historical balances — the balance at the block of the first transaction with unhandled exceptions.

Investigation of the contract at
0x2faa316fc4624ec39adc2ef7b5301124cfb68777:

During this investigation, we found that the last contract of Figure 11 at address 0x2faa316fc4624ec39adc2ef7b5301124cfb68777 used to hold more than 2,000 ETH. We find that the contract is affiliated with Etherdice [5] and while we do not find any issue directly related to the unhandled exception, the contract did have gas issues which caused Ether to be locked for some period of time [6]. However, none of the tools we analyzed marked this contract as vulnerable to locked Ether.

Finally, we look at the amount of Ether that has failed to be sent without being noticed. We find a total of about 169 ETH which has most likely been lost as a result of unhandled exceptions. We assume that any call with an Ether value greater than 10 times the balance of the contract is probably a programming error and therefore discard these.

Contract address	Last send	Balance
0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9	2017-11-15	369,023
0x4a412fe6e60949016457897f9170bb00078b89a3	2018-01-25	4,000
0x90420b8aef42f856a0afb4ffbfaa57405fb190f3	2018-01-08	636.7
0x49edf201c1e139282643d5e7c6fb0c7219ad1db7	2017-05-20	44.23
0x40a05d4ce308bf600cb275d7a3e9113518f59c54	2017-09-11	39.48
0xf0a924661b0263e5ce12756d07f45b8668c53380	2017-07-27	37.51
0x709c7134053510fce03b464982eab6e3d89728a5	2017-11-09	31.72
0xba69e7c96e9541863f009e713caf26d4ad2241a0	2016-05-11	30.95
0x2550d5989f10f1a732c9ffe48f97fe2b178ec2df	2018-01-24	26.00
0x446d1696a5527018453cda3d67aa4c2cd189b9f6	2016-04-21	22.86
0xf9a5e0436c01cc15708449e480fa5f2d6a5b55ca	2017-07-13	21.88
0x5a5eff38da95b0d58b6c616f2699168b480953c9	NA	16.20
0x29aa20fb9b23421e310bdb8a7cfb81d7fbb4a1b3	2017-08-24	15.46
0xe63dc0b48fd13c888661bfb30d7069823f967f03	NA	14.2895
0x2ab10625428aaaee3c0092767c0fa635dd68c57c	2018-06-28	11.47

Figure 12: LE: Inactive contracts flagged with locked Ether funds (The send date is *from* the contract).

5.3 LE: Locked Ether

There is a total of 7,271 contracts flagged vulnerable to locked Ether by [52], [33], [46] and [40]. The 7,271 contracts hold a total value of more than 1 million ETH, which is worth more than 115 million USD. We first analyze the transactions of the contracts that could potentially be locked by conducting analysis at the bytecode-level. Our tool shows than *none* of the contracts are actually affected by the pattern we check for — i.e., dependency on a contract which had been destructed.

It is worth pointing out that some tools, such as MadMax [33], check for other types of issues, which could also lock Ether. To try to check for such issues ourselves, we search for contracts with high monetary value, which have been inactive for a notably long period of time to see whether Ether is indeed locked.

We find a total of 15 contracts, which follow this pattern and summarize them in Figure 12. We manually inspect the top three contracts, which contain a substantial amount of Ether, as well as the contracts, which have never *send* any Ether. These top three contracts are all implementing multisig wallets, which are typically used to store Ether for long periods of time, thus explaining the inactivity. After further manual inspection, we concluded that none of the contracts had been exploited, nor were exploitable.

The first contract, which never sent any Ether, at address 0x5a5eff38da95b0d58b6c616f2699168b480953c9 has its code publicly available. After inspection, it seems to be a "lifelog" and the fact it is not sending Ether seems to be there by-design; in other words, the funds are not locked. Although we were not able to inspect the other contract because its code was not available, we did not find any vulnerability report for this address.

Parity wallet. Despite having been known for more than a year, contracts affected by the Parity wallet bug [22] were not flagged by the tools that we analyzed. As this is one of the

Contract address	First issue	Balance
0xbf4ed7b27f1d666546e30d74d50d173d20bca754	2016-07-21	5,774,963
0x755cdba6ae4f479f7164792b318b2a06c759833b	2016-10-21	344,206
0x3da71558a40f63b960196cc0679847ff50fad22b	2016-09-06	13,818
0xd79b4c6791784184e2755b2fc1659eaab0f80456	2016-05-03	2,013
0xf45717552f12ef7cb65e95476f217ea008167ae3	2016-03-15	1,064
0x9af4f9b67b601967f2f84e9509bb1b411f195ac9	2015-09-18	211.7
0x7011f3edc7fa43c81440f9f43a6458174113b162	2016-05-31	151.3
0x7d56485e026d5d3881f778e99969d2b1f90c50af	2016-04-18	112.1
0xe82719202e5965cf5d9b6673b7503a3b92de20be	2016-04-14	47.80
0x5f0281910af44bfb5fc7e86a404d0304b0e042f1	2017-09-04	40.47
0x7d4c7c61f98d653d5b49b695a01839791e002393	2016-01-30	26.26
0x019d7e5ae8d2ba9a292244311dc7355058ab1b08	2017-02-08	13.86
0x1aa7cd179908adb9c257e7501c1eb32531e66616	2017-03-14	11.66
0xcd3e727275bc2f511822dc9a26bd7b0bbf161784	2017-02-17	10.39

Figure 13: TOD: Contracts potentially victim of transaction ordering dependency attack.

most famous cases of locked Ether, we use the contracts as a sanity check to make sure our tool is detecting this issue correctly.

To find the contracts, we simply have to use the Datalog query for locked Ether in Figure 8c and insert the value of the Parity wallet address as argument *a*. Our results for contracts affected by the Parity bug indeed matches what others had found in the past [11], with the contract at address 0x3bfc20f0b9afcace800d73d2191166ff16540258 having as much as 306,276 ETH locked.

5.4 TO: Transaction Order Dependency

There is a total of 1,877 contracts flagged vulnerable to transaction ordering dependency by [42] and [40]. However, as 642 of these contracts have no transactions except for the contract creation transaction, we only analyze the remaining 1,235 contracts.

For this vulnerability type, we first filter the contracts by simply looking at the transactions, as many transactions are unlikely to affect the result of the analysis. Hence, we filter out all contracts that do not have *at least two transactions* within the same block, as transaction order dependency could not possibly have been exploited in such a case.

This already reduces the number of candidate contracts to only 229, which shows that more than 80% of the contracts which have been flagged were mostly inactive contracts. We then filter out all the contracts where all the transactions in the same block had no input — i.e. called the contract fallback function — as these are most likely benign calls which ended up in the same block.

This leaves 153 contracts. Finally, we filter out contracts where all the transactions in the same block were sent by the same sender. After applying this filter, we are left with a total of 57 contracts to analyze using byte-code level analysis.

At this point, we run the analysis that we described in Section 4 on the remaining contracts. After this step, there

Contract address	First issue	Balance
0x2935aa0a2d2fbb791622c29eb1c117b65b7a9085	2015-10-08	3,197
0x709c7134053510fce03b464982eab6e3d89728a5	2016-12-20	195.6
0x6dfaa563d04a77aff4c4ad2b17cf4c64d2983dc8	2016-06-24	178.6
0x0096800019bf4eca0bed5a714a553ecf844884c5	2016-06-21	174.0
0x9f53cdb4ea5b205a2fcd079f54af7196b1288938	2016-06-29	106.5
0x995020804986274763df9deb0296b754f2659ca1	2018-02-02	89.60
0xaf0f6a53269fc9dbbd9da9f11c368d36b7a60006	2016-08-09	56.50
0x7fd022cc8b6e019260627fafa1d9c3afece18cef	2016-06-11	44.26
0x0f1b27153a56cf318b060838eb527ff441e6ea11	2016-06-10	40.39
0xc3fddcc48edc1284cb1898da2e3dbf5937ea46a0	2018-06-11	37.18

Figure 14: IO: Contracts potentially victim of integer overflows.

are 48 contracts left, which shows that our initial filtering had done a good job at keeping only contracts that might have been affected.

We finally look for contracts that held Ether at the time the transactions were made. For each contract, we find the block where transaction order dependency could have happened with the highest balance and select all contracts with a balance higher than 10 ETH.

This gives us a total of 14 contracts. We show the addresses of these contracts in Figure 13. The *First issue* column is the date on which the transaction order issue had been seen for the first time. The *Balance* column is the balance of the contract at the time the first issue occurred.

Looking at the date, it is clear that this issue has not been seen in the wild for a long time, at least within the range of the contracts flagged by the tools we analyzed. We manually investigated the top five contracts in the list but did not find *any* evidence of transaction order-related exploits.

5.5 IO: Integer overflow

There is a total of 2,472 contracts flagged vulnerable to integer overflow, which accounts for a total of more than 1 million transactions. We run the approach we described in Section 4 to search for actual occurrences of integer overflows.

It is worth noting that for integer overflow analysis we rely on properties of the compiler, which might not always hold. Our analysis is therefore not complete and may contain some false positives.

We find a total of 141 contracts with transactions where an integer overflow might indeed have occurred. We first inspect some of the results we obtained a little further to get a better sense of what kind of cases lead to overflows. We find that a very frequent cause of overflow is rather underflow of unsigned values.

Investigation of the contract at
0xdcabd383a7c497069d0804070e4ba70ab6ecdd51:

This contract was flagged positive to both unhandled exceptions and integer overflow by our tool. After inspection, it

Vulnerable				Exploited contracts			Exploited Ether	
Vuln.	**Vulnerable contracts**	**Total Ether at stake**	**Transactions analyzed**	**Contracts exploited**	**Contracts w/Ether exploited**	**% w/Ether exploited**	**Exploited Ether**	**% of Ether exploited**
RE	4,336	1,027,585	411,604	113	6	0.14%	6,075	0.59%
UE	11,426	208,528	3,063,510	268	6	0.053%	168.5	0.081%
LE	7,271	1,135,313	8,550,292	0	0	0%	0	0%
TO	1,877	207,926	2,817,260	57	14	0.75%	189.0	0.091%
IO	2,472	508,750	1,214,206	141	23	0.93%	2,661	0.52%
Total	21,270	3,088,102	16,056,872	504	49	0.0023%	9,094	0.30%

Figure 15: Understanding the exploitation of potentially vulnerable contracts.

seems that at block height 1,364,860, the owner tried to reduce the fees but the unsigned value of the fees overflowed and became a huge number. Because of this issue, the contract was then trying to send large amount of Ether. This resulted in failed calls which happened not to be checked, hence the flag for unhandled exceptions.

Next, we look for contracts which had at least 10 ETH at the time of the overflow. We find a total of 23 contracts with this condition. We show the 10 contracts with the highest balance at the time of the overflow in Figure 14. The top contract in this list had a large balance at the time of the issue but it had already been self-destructed and we could therefore not inspect further. We did not, however, find any report mentioning this contract.

Finally, we analyze all the transactions which resulted in integer overflows and find the amount of Ether which it may have influenced overall. Although it is hard to evaluate exactly how these overflows may have affected the Ether balances held by the contracts, the total amounts of Ether transferred in and out during a transaction containing an overflow gives us a decent proxy on the upper bound of Ether at stake.

5.6 Summary

We summarize all our findings, including the number of contracts originally flagged, the amount of Ether at stake, and the amount *actually exploited* in Figure 15. The *Contracts exploited* column indicates the number of contracts which we believe to have been exploited and *Contracts with Ether exploited* refers to the number of contracts which contained at least 10 ETH at the time of exploitation. The *Exploited Ether* column shows the maximum amount of Ether that potentially could have been exploited and the next column shows the percentage this amount represents compared to the total amount at stake. The *Total* row accounts for contracts flagged with more than one vulnerability only once.

Overall, we find that the situation is *many orders of magnitude* less critical than most papers we review in Section 2 make it sound. The amount of money which was actually exploited is *not even remotely close* to the 5 billion USD [33] or even 500 million [40] USD, which were supposedly at risk.

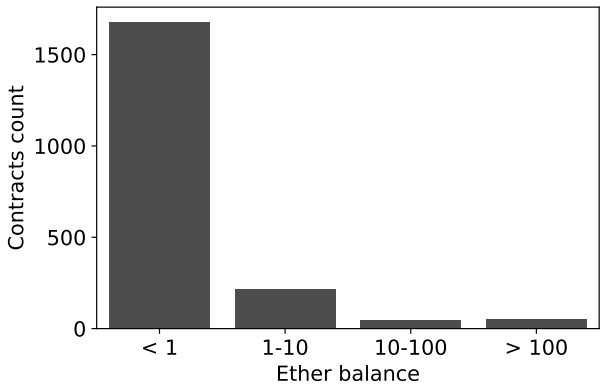
Below, we summarize the main takeaways regarding each vulnerability we examined in this paper.

Re-entrancy. This is by far the most dangerous issue of all the ones we have analyzed, accounting for more than 65% of the total exploitations we observed. Although some proposals have been made to add a protection against this in the Solidity compiler [15], we think that this issue should instead be handled at the interpreter level. Sereum [47] is an attempt to do this, and we think that such an addition would help make the Ethereum smart contracts ecosystem considerably more secure.

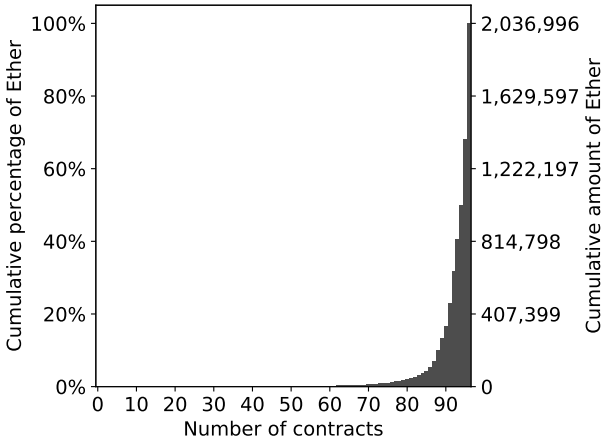
Unhandled exceptions. As we can see in Figure 15, the amount of Ether *actually exploited* is very low compared to other vulnerabilities. Although unhandled exceptions used to be a real issue a few years ago, the Solidity compiler has now made a lot of progress and such a pattern would generate a warning at compile time. Therefore, we think that this issue has already been given enough attention and is handled well enough by the current generation of development tools, at least as it pertains to EVM contracts developed in Solidity.

Locked Ether. In this work, we mainly cover locked Ether caused by self-destructed library contracts, such as the one seen by the Parity wallet bug [22]. This particular issue has generated much attention by the community because of the amount of money involved. However, we believe that the pattern of delegating to a library is a common pattern when working with smart contracts, and that such contracts should not be treated as "vulnerable". Indeed, we show that this issue did not happen even *a single time* in our dataset. We believe that the focus should lie on keeping library contracts safe opposed to not using them at all.

Transaction Order Dependency. While this vulnerability has received a lot of focus in the academic community [42,52] it has rarely been observed in reality. Our data confirms that this is very rarely exploited in practice. One of the reason is that this vulnerability is simply quite hard to exploit: in order to reliably arrange the order of the transactions, the attacker needs to be a miner. Given that almost 85% of Ether is mined by mining pools [2], it would require the mining pool operator to be dishonest. Pragmatically speaking, there is generally



(a) Ether held in analyzed contracts with non-zero balance.



(b) Cumulative Ether held in the 96 analyzed contracts holding more than 10 ETH.

Figure 16: Ether held in contracts: describing the distribution.

not enough financial incentives for mining pools to perform such an attack, in part because more lucrative alternative opportunities may exist for them if they are dishonest.

Integer overflow. While this remains a very common issue with smart contracts, it is both difficult to automatically detect such issues and to evaluate the impact that they may actually have. The Solidity compiler now emits warning or errors for cases working directly on integral literals but does not check anything else than that. A case as simple as `uint8 n = 255; n++;` would not get any warnings or errors. We believe that this is a place where static analysis tools such as [40] or [33] can be very valuable to avoid smart contracts that fail in unexpected ways.

6 Discussion

In this section, we discuss some of the factors we think might be impacting the actual exploitation of smart contracts.

We believe that a major reason for the difference between the number of vulnerable contracts reported and the number of contracts exploited is the distribution of Ether among contracts. Indeed, only about 2,000 out of the 21,270 contracts in our dataset contain Ether, and most of these contracts have a balance lower than 1 ETH. We show the balance distribution of the contracts containing Ether in our dataset in Figure 16a. Furthermore, the top 10 contracts hold about 95% of the total Ether. We show the cumulative distribution of Ether within the contracts containing more than 10 ETH in Figure 16b. This shows that, as long as the top contracts cannot be exploited, the total amount of Ether that is actually at stake will be nowhere close to the upper bound of "vulnerable" Ether.

We decide to manually inspect the top 6 contracts — i.e contracts with the highest balances at the time of writing — marked as vulnerable by any of the tools in our set. We focused on the top 6 because it happened to be the number of contracts which currently hold more than 100,000 ETH. These contracts hold a total of 1,695,240 ETH, or 83% of the total of 2,037,521 ETH currently held by all the contracts in our dataset.

Investigation of the contract at
`0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae`:

The source code for this contract is not directly available. However, we discovered that this is the multi-signature wallet of the Ethereum foundation [3] and that its source code is available on GitHub [16]. We inspect the code and find that the only calls taking place require the sender of the message to be an owner. This by itself is enough to prevent any re-entrant call, as the malicious contract would have to be an owner, which does not make sense. Furthermore, although the version of Oyente used in the paper reported the re-entrancy, more recent versions of the tool did not report this vulnerability anymore. Therefore, we safely conclude that the re-entrancy issue was a false alert.

We were able to inspect 4 of the 5 remaining contracts. The contract at address `0x07ee55aa48bb72dcc6e9d78256648910de513eca` is the only one for which we were unable to find any information. The second, third and fifth contracts in the list were also multi-signature wallets and exploitation would require a majority owner to be malicious. For example, for Ether to get locked, the owners would have to agree on adding enough extra owners so that all the loops over the owners result in an out-of-gas exception. The contract at address `0xbf4ed7b27f1d666546e30d74d50d173d20bca754` is a contract known as `WithDrawDAO` [17]. We did not find any particular issue, but it does use a delegate pattern which explains the locked ether vulnerability marked by Zeus.

Overall, all the contracts from Figure 17 that we could analyze seemed quite secure and the vulnerabilities flagged were definitely not exploitable. Although there are some very rare cases that we present in Section 7 where contracts with high

Address	Ether balance	Deployment date	Flagged vulnerabilities
0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae	649,493	2015-08-08	Oyente: RE
0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9	369,023	2016-11-10	MadMax: LE, Zeus: IO
0x851b7f3ab81bd8df354f0d7640efcd7288553419	189,232	2017-04-18	MadMax: LE
0x07ee55aa48bb72dcc6e9d78256648910de513eca	182,524	2016-08-08	Securify: RE
0xcafe1a77e84698c83ca8931f54a755176ef75f2c	180,300	2017-06-04	MadMax: LE
0xbf4ed7b27f1d666546e30d74d50d173d20bca754	124,668	2016-07-16	Securify: TO, UE; Zeus: LE, IO

Figure 17: Top six most valuable contracts flagged as vulnerable by at least one tool.

Ether balances are being stolen, these remain exceptions. The facts we presented up to now help us confirm that the amount of Ether at risk on the Ethereum blockchain is nowhere as close as what is claimed [33, 40].

7 Related work

Some major smart contracts exploits have been observed on Ethereum in recent years [49]. These attacks have been analyzed and classified [19] and many tools and techniques have emerged to prevent such attacks [32, 35]. In this section, we will first provide details about two of the most prominent historic exploits and then present existing work aimed at increasing smart contract security.

7.1 Motivating Large-scale Exploits

TheDAO exploit. TheDAO exploit [49] is one of the most infamous bugs on the Ethereum blockchain. Attackers exploited a re-entrancy vulnerability [19] of the contract which allowed for the draining of the contract's funds. The attacker contract could call the function to withdraw funds in a re-entrant manner before its balance on TheDAO was reduced, making it indeed possible to freely drain funds. A total of more than 3.5 million Ether were drained. Given the severity of the attack, the Ethereum community finally agreed on hard-forking.

Parity wallet bug. The Parity Wallet bug [22] is another prominent vulnerability on the Ethereum blockchain which caused 280 million USD worth of Ethereum to be frozen on the Parity wallet account. It was due to a very simple vulnerability: a library contract used by the parity wallet was not initialized correctly and could be destructed by anyone. Once the library was destructed, any call to the Parity wallet would then fail, effectively locking all funds.

7.2 Analyzing and Verifying Smart Contracts

There has been a lot of efforts in order to prevent such attacks and to make smart contracts more secure in general. We will here present some of the tools and techniques which have been presented in the literature.

Oyente. Oyente [42] is one of the first tools which has been developed to analyze smart contracts. It uses symbolic execution to check for the following vulnerabilities: transaction ordering dependency, re-entrancy and unhandled exceptions. The tool takes as input the bytecode of a smart contract and a state of the Ethereum blockchain. It emulates the EVM and explores the different paths of the contracts. It then uses the Z3 SMT solver [31] to decide the satisfiability of conditions which would make the program vulnerable in the current path.

ZEUS. ZEUS [40] is a static analysis tool which can check for a vast range of vulnerabilities such as re-entrancy, unhandled exceptions, integer overflows, transaction order dependency and others. Unlike Oyente, it operates on the high-level representation of the smart contract written in Solidity and not on the bytecode. It first generates a XACML-styled [9] policy from the Solidity abstract syntax tree (AST) which can be further customized by the user. A policy could for example enforce that the amount to send to a user is always smaller or equal to his balance. It then transpiles the policy and the Solidity contract code to LLVM bitcode [41] and finally uses constrained Horn clauses [21, 43] over the LLVM bitcode to check that the policy is respected.

Maian. Maian [46] is also a tool to analyze contracts but instead of using static analysis to find bugs in the contract, it tries to find vulnerabilities across long sequences of invocations of a contract. It focuses mainly on finding three types of vulnerabilities: contracts that can be removed from the blockchain by anyone, contracts which can lock funds by being unable to send Ether, and contracts which can "leak" Ether to a user they have never interacted with. The tool performs symbolic analysis across multiple executions of the contract in order to find traces that violate the security properties being checked.

SmartCheck. [51] is a tool which, as ZEUS, analyzes the high-level solidity source code of the smart contract to find vulnerabilities. It is also able to find a wide range of vulnerabilities such as re-entrancy, unhandled exceptions, locked Ether, integer overflows and many more. Not unlike ZEUS, SmartCheck transforms the solidity contract in an intermediate representation (IR) but uses an XML-based. It then uses XPaths patterns to check for security properties in the con-

tract IR. This simple approach makes the system efficient but loses precision for vulnerabilities which cannot naturally be expressed as XPaths, such as re-entrancy.

Securify. Securify [52] is a static analysis tool which checks security properties of the EVM bytecode of smart contracts. The security properties are encoded as patterns written in a domain-specific language, and checked either for compliance or violation. To analyze the contract, Securify first transforms the EVM bytecode into a stackless static-single assignment form. It then infers semantic facts from the contract such as data and control-flow dependencies which it encodes in stratified Datalog [53]. It finally interprets the security patterns to check for their violation or compliance by querying the inferred facts.

ContactFuzzer. Unlike the previous tools, ContractFuzzer [39] uses dynamic analysis and more particularly fuzzing to find vulnerabilities in smart contracts. It is capable of detecting a wide range of vulnerabilities such as re-entrancy, locked Ether or unhandled exceptions. To operate, ContractFuzzer generates inputs for the contracts by looking at their Application Binary Interface (ABI). It uses an instrumented EVM to run the fuzzed contracts and records the executed instructions during fuzzing and, analyzes them later on to find vulnerabilities in the contract. Due to the dynamic analysis nature of the detection, this tool has a substantially higher true positive rate than the static analysis tools previously presented.

Vandal. Vandal [23] is a static analysis tool which is in many ways similar to Securify [52]. Vandal also analyzes the EVM bytecode by decompiling it and encodes properties of the smart contract into Datalog. The tool is able to detect vulnerabilities, such as re-entrancy and unhandled exceptions, and can easily be extended by writing queries to check for other types of vulnerabilities in Datalog.

MadMax. MadMax [33] also statically analyzes smart contracts but focuses mainly on vulnerabilities related to gas. It is the first tool to detect so-called unbounded mass operations where a loop is bounded by a dynamic property such as the number of users, causing the contract to always run out of gas passed a certain number of users. MadMax is built on top of the decompiler implemented by Vandal and also encodes properties of the smart contract into Datalog. It is performant enough to analyze all the contracts of the Ethereum blockchain in only 10 hours.

Gasper. Gasper [26] is also a static analysis tool focused on gas but instead of looking for vulnerabilities it searches for patterns which might be costly to the contract owner in terms of gas. Gasper builds a control flow graph from the EVM bytecode and uses symbolic execution backed by an SMT solver to explore the different paths that might be taken. Gasper looks for patterns such as dead code or expensive operations in loops to help contract developers reduce gas cost.

Sereum. Sereum [47] focuses on detecting and preventing exploitation at runtime rather than trying to detect vulnerabilities beforehand. It proposes a modification to the Go Ethereum client which is able to detect and reject re-entrancy exploits and could also handle other exploits. It first performs taint analysis to infer properties about variables in the contract and then checks that tainted variables are not used in a way which would violate security properties. For re-entrancy, it uses this technique to check that no variable accessing the contract storage is used in a reentrant call.

Formal verification. There has also been some efforts to formally verify smart contracts. [37] is one of the first efforts in this direction and defines the EVM using Lem [45], which allows to generate definitions for theorem provers such as Coq [20]. [34] presents a complete small-step semantics of EVM bytecode and formalizes it using the F* proof assistant [50]. A similar effort is made in [36] to give an executable formal specification of the EVM using the K Framework [48].

8 Conclusion

In this paper, we surveyed the 21,270 vulnerable contracts reported by five recent academic projects. We proposed a Datalog-based formulation for performing analysis over EVM execution traces and used it to analyze a total of more than 16 million transactions executed by these contracts. We found that at most 504 out of 21,270 contracts have been subjected to exploits. This corresponds to at most 9,094 ETH (1 million USD), or only 0.30% of the 3 million ETH (350 million USD) claimed to be at risk.

While we are certainly not implying that smart contract vulnerability research is without merit, our results suggest that the potential impact of vulnerable code had been greatly exaggerated. These results are likely to be controversial, yet, we are reluctant to over-claim as far as the underlying reasons.

We hypothesize that the reasons for these vast differences are multifold: lack of appetite for exploitation, the sheer difficulty of executing some exploits, fear of attribution, other more attractive exploitation options, etc. From what we can find by analyzing the blockchain, a majority of Ether is held by only a small number of contracts. Further, the vulnerabilities reported on these contracts are either false positives or not applicable in practice, making exploitation significantly less attractive.

References

[1] 0x Bug bounty. https://0x.org/wiki#Deployed-Addresses. [Online; accessed 12-February-2019].

[2] Are Miners Centralized? A Look into Mining Pools. https://media.consensys.net/are-miners-centralized-a-look-into-mining-

pools-b594425411dc. [Online; accessed 11-February-2019].

[3] contract with 11,901,464 ether? What does it do? https://www.reddit.com/r/ethereum/comments/3gi0qn/contract_with_11901464_ether_what_does_it_do/. [Online; accessed 12-February-2019].

[4] Critical ether token wrapper vulnerability - eth tokens salvaged from potential attacks. https://www.reddit.com/r/MakerDAO/comments/4niu10/critical_ether_token_wrapper_vulnerability_eth/. [Online; accessed 9-February-2019].

[5] Etherdice. https://www.etherdice.io. [Online; accessed 10-February-2019].

[6] Etherdice is down for maintenance. https://www.reddit.com/r/ethereum/comments/47f028/etherdice_is_down_for_maintenance_we_are_having/. [Online; accessed 10-February-2019].

[7] Ethereum Smart Contract Best Practices. [Online; accessed 21-January-2019].

[8] Etherscan — Ethereum (ETH) Blockchain Explorer. https://etherscan.io. [Online; accessed 21-January-2019].

[9] eXtensible Access Control Markup Language (XACML) XML Media Type. https://tools.ietf.org/html/rfc7061.

[10] MakerDAO. https://makerdao.com/en/. [Online; accessed 9-February-2019].

[11] Multisig wallets affected by the Parity wallet bug. https://github.com/elementus-io/parity-wallet-freeze. [Online; accessed 21-January-2019].

[12] Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum. [Online; accessed 21-January-2019].

[13] Oyente Benchmarks. https://oyente.tech/benchmarks/. [Online; accessed 19-November-2018].

[14] Solidified. https://solidified.io/faq. [Online; accessed 11-February-2019].

[15] Solidity 'noreentrancy' flag. https://github.com/ethereum/solidity/issues/2619. [Online; accessed 11-February-2019].

[16] Source code of the Ethereum Foundation Multisig wallet. https://github.com/ethereum/dapp-bin/blob/master/wallet/wallet.sol. [Online; accessed 12-February-2019].

[17] The DAO Refunds. https://theethereum.wiki/w/index.php/The_DAO_Refunds. [Online; accessed 12-February-2019].

[18] We Got Spanked: What We Know So Far. https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe. [Online; accessed 9-February-2019].

[19] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *POST*, 2017.

[20] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[21] Nikolaj Bjørner, Kenneth L McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories.

[22] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An In-Depth Look at the Parity Multisig Bug.

[23] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A Scalable Security Analysis Framework for Smart Contracts. Technical report, 2018.

[24] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Ethereum*, (January):1–36, 2014.

[25] Vitalik Buterin. EIP 7. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-7.md, 2015. [Online; accessed 21-January-2019].

[26] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 442–446, 2017.

[27] ConSensys. A development framework for Ethereum. https://github.com/trufflesuite/truffle. [Online; accessed 21-January-2019].

[28] ConsenSys. Mythril Classic. https://github.com/ConsenSys/mythril-classic. [Online; accessed 21-January-2019].

[29] ConSensys. Personal blockchain for Ethereum development. https://github.com/trufflesuite/ganache. [Online; accessed 21-January-2019].

[30] Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, CA, USA, 1st edition, 2017.

[31] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[32] Ardit Dika. Ethereum Smart Contracts : Security Vulnerabilities and Security Tools. (December), 2017.

[33] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *SPLASH 2018 Oopsla*, 2(October), 2018.

[34] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A Semantic Framework for the Security Analysis of Ethereum smart contracts. 2018.

[35] Dominik Harz and William Knottenbelt. Towards Safer Smart Contracts: A Survey of Languages and Verification Methods. *arXiv preprint arXiv:1809.09805*, 2018.

[36] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *Proceedings - IEEE Computer Security Foundations Symposium*, 2018.

[37] Yoichi Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Workshop on Trusted Smart Contracts*, 2017.

[38] Neil Immerman. Descriptive complexity, 1999.

[39] Bo Jiang and Ye Liu. ContractFuzzer : Fuzzing Smart Contracts for Vulnerability Detection. *Ase*, (July):1–11, 2018.

[40] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts.

[41] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[42] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *CCS*, 2016.

[43] Kenneth L McMillan. Interpolants and symbolic model checking. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 89–90. Springer, 2007.

[44] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Journal of Cases on Information Technology (JCIT)*, 21(1):19–32, 2019.

[45] Dominic P Mulligan, Scott Owens, Kathryn E Gray, Tom Ridge, and Peter Sewell. Lem: reusable engineering of real-world semantics. In *ACM SIGPLAN Notices*, volume 49, pages 175–188. ACM, 2014.

[46] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. 2018.

[47] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum : Protecting Existing Smart Contracts Against Re-Entrancy Attacks. (February), 2019.

[48] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[49] Us Securities and Exchange Commission. Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO. Technical report, 2017.

[50] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming*, pages 266–278. ACM, 2011.

[51] Sergei Tikhomirov, Ekaterina Voskresenskaya, and Ivan Ivanitskiy. SmartCheck : Static Analysis of Ethereum Smart Contracts. (October):9–16, 2017.

[52] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. (July), 2018.

[53] Jeffrey D Ullman. *Principles of database systems*. Galgotia publications, 1984.

[54] Gavin Wood. Ethereum yellow paper, 2014.