Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited

Daniel Perez and Benjamin Livshits, *Imperial College London*

https://www.usenix.org/conference/usenixsecurity21/presentation/perez

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited

Daniel Perez
Imperial College London

Benjamin Livshits
Imperial College London

Abstract

In recent years, 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. While most of the work has focused on detecting *vulnerable* contracts, in this paper, we focus on finding how many of these vulnerable contracts have actually been *exploited*. We survey the 23,327 vulnerable contracts reported by six recent academic projects and find that, despite the amounts at stake, only 1.98% of them have been exploited since deployment. This corresponds to at most 8,487 ETH (~1.7 million USD[1]), or only 0.27% of the 3 million ETH (600 million USD) at stake. We explain these results by demonstrating that the funds are very concentrated in a small number of contracts which are *not exploitable* in practice.

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 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 such an investigation. In this work, we aim to contrast the vulnerabilities reported in smart contracts on the Ethereum [16] blockchain with the actual exploitation of these contracts.

We collect the data shared with us by the authors of six recent papers [24, 31, 32, 35, 39, 51] focusing 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 — 23,327 — represent an excellent study subject.

To make our approach more general, we express six different frequently reported vulnerability classes as Datalog queries computed over relations that represent the state of the Ethereum blockchain. 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 that 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 cryptocurrency exploits such as TheDAO [45] or the Parity wallet [14] bugs.

Contributions. Our contributions are:

- **Datalog formulation.** 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 20 million transactions from the Ethereum blockchain to search for exploits. We highlight that our analyses run *automatically* based on the facts that we extract and the rules defining the vulnerabilities we cover in this paper.
- **Experimental evaluation of exploitation.** We analyze the vulnerabilities reported in six recently published studies and conclude that, although the number of *vulnerable* contracts and the amount of money at risk is very high, the amount of money actually *exploited* is several orders of magnitude lower.

  We discover that out of 23,327 vulnerable contracts worth a total of 3,124,433 ETH, 463 contracts may have been exploited for an amount of 8,487 ETH, which represents only 0.27% of the total amount at stake.
- **Proposed explanations.** We hypothesize that the main reasons for these vast differences is that the amount of *exploitable* Ether is very low compared to the amount of Ether flagged *vulnerable*. Indeed, further analysis of the vulnerable contracts and the Ether they contain suggests that a large majority of Ether is held by only a small number of contracts, and that the vulnerabilities reported on these contracts are either false positives or

[1] We use the exchange rate on 2020-05-16: 1 ETH = 200 USD. For consistency, any monetary amounts denominated in USD are based on this rate.

not exploitable in practice. We also confirm that the set of all contracts on the Ethereum blockchain has a similar distribution of wealth to that in our dataset.

To make many of the discussions in this paper more concrete, we present a thorough investigation of the high-value contracts in Appendix A.

2 Background

The Ethereum [16] 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 [19]. 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) [53]. 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 *gas*. Each executed instruction consumes an agreed upon amount of gas [53]. 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 [40]. 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 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, also called *internal transactions*, do not create new transactions and are therefore not directly recorded on-chain. This means that looking at transactions without executing them does not provide enough information to follow the flow of Ether.

2.1 Smart Contracts Vulnerabilities

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). When a contract "calls" another account, it can choose the amount of gas it allows the called party to use. If the target account is a contract, it will be executed and can use the provided gas budget. If such a contract is malicious and the gas budget is high enough, it can try to call back in the caller — a re-entrant call. If the caller's implementation is not re-entrant, for example because it did not update his internal state containing balances information, the attacker can use this vulnerability to drain funds out of the vulnerable contract [31, 35, 51]. This vulnerability was used in TheDAO exploit [45], essentially causing the Ethereum community to decide to rollback to a previous state using a hard-fork [37]. We provide more details about TheDAO exploit in Section 8

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 is unchecked, the caller continues its execution even if the payment failed, which can easily lead to inconsistencies [15, 31, 35, 48].

Locked Ether (LE). Ethereum smart contracts can, as any account on Ethereum, receive Ether. However, there as several reasons for which the received funds might get locked permanently into the contract.

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. If this was the only way for such a contract to send Ether, it will result in the funds being permanently locked. This is what happened in the Parity Wallet bug in November 2017, locking millions of USD worth of Ether [14]. We provide more details about it in Section 8

There are also cases where the contract will *always* run out of gas when trying to send Ether which could result in locking the contract funds. More details about such issues can be found in [24].

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 enough users to trigger an overflow.

Unrestricted Action (UA). Contracts often perform authorization, by checking the sender of the message, to restrict the type of action that a user can take. Typically, only the owner of a contract should be allowed to destroy the contract or set a new owner. Such an issue can happen not only if the developer

Name	Vulnerabilities						Report month	Citation
	RE	UE	LE	TO	IO	UA		
Oyente	✓	✓		✓	✓		2016-10	[35]
ZEUS	✓	✓	✓	✓	✓		2018-02	[31]
Maian			✓			✓	2018-03	[39]
SmartCheck	✓	✓	✓		✓		2018-05	[48]
Securify	✓	✓	✓	✓		✓	2018-06	[51]
ContractFuzzer	✓	✓					2018-09	[30]
teEther						✓	2018-08	[32]
Vandal	✓	✓					2018-09	[15]
MadMax			✓		✓		2018-10	[24]

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

forgets to perform critical checks but also if an attacker can execute arbitrary code, for example by being able to control the address of a delegated call [32].

2.2 Analysis Tools

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. Given the many common vulnerabilities in smart contracts, some of which we described in the previous section, a large number of tools have been developed to find them automatically [18, 35, 51]. 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 Section 2.1 and give a more detailed description of these tools in Section 8.2.

2.3 Definitions

We give the definitions used in this paper for the terms *vulnerable*, *exploitable* and *exploited*.

vulnerable: A contract is vulnerable if it has been flagged by a static analysis tool as such. As we will see later, this means that some contracts may be *vulnerable* because of a false-positive.

exploitable: A contract is exploitable if it is vulnerable and the vulnerability could be exploited by an external attacker. For example, if the "vulnerability" flagged by a tool is in a function which requires to own the contract, it would be *vulnerable* but not *exploitable*.

Name	Contracts analyzed	Vulnerabilities found	Ether at stake at time of report
Oyente	19,366	7,527	1,287,032
Zeus	1,120	861	671,188
Maian	NA	2,691	15.59
Securify	29,694	9,185	724,306
MadMax	91,800	6,039	1,114,958
teEther	784,344	1,532	1.55

Figure 2: Summary of the contracts in our dataset.

exploited: A contract is exploited if it received a transaction on Ethereum's main network which triggered one of its vulnerabilities. Therefore, a contract can be *vulnerable* or even *exploitable* without having been *exploited*.

3 Dataset

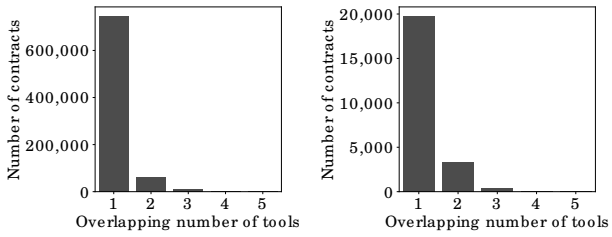
In this paper, we analyze the vulnerable contracts reported by the following six academic papers: [35], [31], [39], [51], [24] and [32]. To collect information about the addresses analyzed and the vulnerabilities found, we reached out to the authors of the different papers.

Oyente [35] data was publicly available [34]. 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 [48], [30] and [15] 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 821,219 contracts, of which 23,327 contracts have been flagged as vulnerable to at least one of the six vulnerabilities described in Section 2. Although we received the data directly from the authors, the numbers of contracts analyzed usually did not match the data reported in the papers, which we show in Figure 1. We believe the two main results for this are: authors improving their tools after the publication and authors not including duplicated contracts in their data they provided us. 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 [11] as-is, we adapt it to fit the vulnerabilities analyzed by the tools in our dataset. We do not cover vulnerabilities not analyzed by at least two of the six tools. We settle on the six types of vulnerabilities described in Section 2: re-entrancy (RE), unhandled exception (UE), locked Ether (LE), transaction order dependency (TO), integer



(a) Overlapping contracts analyzed.

(b) Overlapping vulnerabilities flagged.

Figure 3: Histograms that show the overlap in the contracts analyzed and flagged by examined tools.

Tools	Total	Agreed	Disagreed	% agreement
Oyente/Securify	774	185	589	23.9%
Oyente/Zeus	104	3	101	2.88%
Zeus/Securify	108	2	106	1.85%

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

overflows (IO) and unrestricted actions (UA). As the papers we survey use different terms and slightly different definitions for each of these vulnerabilities, we map the relevant vulnerabilities to one of the six types of vulnerabilities we analyze. We show how we mapped these vulnerabilities in Figure 5.

Overlapping vulnerabilities. In this subsection, we first check how much overlap there is between contracts in our dataset: how many contracts have been analyzed by multiple tools and how many contracts were flagged vulnerable by multiple tools. We note that most papers, except for [35], are written around the same period. We find that 73,627 out of 821,219 contracts have been analyzed by at least two of the tools but only 13,751 by at least three 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 relatively small. We assume one of the reasons is that some tools work on Solidity code [31] while other tools work on EVM bytecode [35, 51], 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 supported by three of the tools we analyze. In Figure 4, we show the agreement between the three tools supporting 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 the different tools' performance, 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 analyses we perform in order to check for exploits of the vulnerabilities described in Section 2.

To check for potential exploits, we 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 the six types of vulnerabilities presented in Section 2.

To perform our analyses, we first retrieve transaction data for all the contracts in our dataset. Next, to perform bytecode-level analysis, we extract the execution traces for the transactions potentially affecting contracts of interest. We use EVM's debug functionality, which gives us the ability to replay transactions while tracing executed instructions. To speed-up the data collection process, we patch the Go Ethereum client [10], 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. To analyze the traces, we encode them into a Datalog representation; Datalog is a language implementing first-order logic with recursion [29], allowing us to concisely express properties about the execution traces. We use the following domains to encode the information about the traces as Datalog facts, noting V as the set of program variables and A is the set of Ethereum addresses. We show an overview of the facts that we collect and the relations we use to check for possible exploits in Figure 7. We highlight that our analyses run *automatically* based on the facts that we extract and the rules that define various violations described in subsequent sections.

4.1 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 [53]. Using this information, we can record $\texttt{call}(a_1, a_2, p)$ facts described in Figure 7a. We note that a contract can also create a new contract using `CREATE` and execute a re-entrancy attack using it [43]. We therefore treat this instruction in a similar way as `CALL`. Using these, we then use the query shown in Figure 7c to retrieve potentially malicious re-entrant calls.

Analysis correctness. Our analysis for re-entrant calls is sound but not complete. As the EVM executes each contract in a single thread, a re-entrant call must come from a recur-

	Oyente	ZEUS	Securify	MadMax	Maian	teEther
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	Ether liquidity	unbounded mass operation wallet griefing	greedy	—
IO	—	integer overflow	—	integer overflows	—	—
UA	—	integer overflow	—	integer overflows	prodigal	exploitable

Figure 5: Mapping of the different vulnerabilities analyzed.

```
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 6: Correctly handled failed `send`.

sive call. For example, given A, B, C and D being functions, a re-entrant call could be generated with a call path such as $A \rightarrow B \rightarrow C \rightarrow A$. Our tool searches for all mutually-recursive calls; it supports an arbitrarily-long calls path by using a recursive Datalog rule, making the analysis sound. However, we have no way of assessing if a re-entrant call is malicious or not, which can lead to false positives.

4.2 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 6. 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 `call_result`(v, n) facts. An important edge case to consider are calls to pre-compiled contracts, which are typically called by the compiler and do not require their return value to be checked, as they are results of computation, where 0 could be a valid value, but could result in false-positives. As pre-compiled contracts have known addresses between 1 and 10, we choose to simply not record `call_result` facts for such calls.

As shown in Figure 6b, the EVM uses the `JUMPI` instruction 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 7c.

Analysis correctness. The analysis we perform to check for unhandled exceptions is complete but not sound. All failed calls in the execution of the program will be recorded, while we accumulate facts about the execution. We then use a recursive Datalog rule to check if the call result is used directly or indirectly in a condition. We could obtain false negatives if the call result is used in a condition but the condition is not enough to prevent an exploit. However, given that the most prevalent pattern for this vulnerability is the result of `send` not being used at all [51], and when the result is used, it is typically done within a `require` or `assert` expression, we hypothesize that such false negatives should be very rare.

4.3 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 [14]. 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 `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 not accessible anymore to callers. 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 collect Datalog facts about all the values of the program counter before and after every `DELEGATECALL` instruction. In particular, we first mark

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
create($a_1 \in A$, $a_2 \in A$, $p \in \mathbb{N}$)	a_1 creates 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
caller($v \in V, a \in A$)	v is the caller with address a
load_data($v \in V$)	v contains transaction call data
restricted_inst($v \in V$)	v is used by a restricted instruction
selfdestruct($v \in V$)	v is used in SELFDESTRUCT

(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($a_1 \in A$, $a_2 \in A$, $p \in \mathbb{Z}$) :- create(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).
condition_flow($v \in V$, $v \in V$) :- in_condition(v). condition_flow(v_1, v_2) :- depends(v_1, v_2), in_condition(v_2).
depends_caller($v \in V$) :- caller(v_2, _), depends(v, v_2).
depends_data($v \in V$) :- load_data(v_2, _), depends(v, v_2).
caller_checked($v \in V$) :- caller(v_2,_), condition_flow(v_2, v_3), $v_3 < v$.

(b) Datalog rule definitions.

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 Excep.	call_result(v, 0), ¬condition_flow(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), $r_1 \neq r_2$
Unrestricted Action	restricted_inst(v), depends_data(v), ¬depends_caller(v), ¬caller_checked(v) ∨ selfdestruct(v), ¬caller_checked(v)

(c) Datalog queries for detecting different vulnerability classes.

Figure 7: Datalog setup.

the program counter value 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 program counter value at which the call returns — call_exit($i_2 \in \mathbb{N}$).

If the called contract does not exist anymore, $i_1 + 1 = i_2$ must hold. Therefore, we can use the Datalog query shown in Figure 7c to retrieve the destructed contracts address.

Analysis correctness. The approach we use to detect locked Ether is sound and complete for the class of locked funds vulnerability we focus on. All vulnerable contracts must have a DELEGATECALL instruction. If the issue is present and the call contract has indeed been destructed, it will always result in a no-op call. Our analysis records all of these calls and systematically check for the program counter before and after the execution, making the analysis sound and complete.

4.4 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 [35] or [51], 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 to the storage, SSTORE. In the EVM, the location of the storage to use for both of these instructions is passed as an argument, and referred to as the storage *key*. This key is available on the stack at the time 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 7c.

Analysis correctness. Our approach to check for transaction order dependencies is sound but not complete. With the definition we use, for a contract to have a transaction order dependency it must have two transactions in the same block, which affect the same key in the storage. We check for all such cases, and therefore no false-negatives can exist. However, finding if there is a transaction order dependency requires more knowledge about how the storage is used and our approach could therefore result in false positives.

4.5 Integer Overflow

The EVM is completely untyped and expresses everything in terms of 256-bits words. Therefore, types are handled entirely 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 the result of an instruction such as `SIGNEXTEND` or `SDIV` 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 7b 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 7c to find instructions which overflow.

Analysis correctness. Our analysis for integer overflow is neither sound nor complete. The types are inferred by using properties of the compiler using a heuristic which should work for most of cases but can fail. For example, if a contract contains code which yields `AND value 0xff` but value is an `uint32`, our type inference algorithm would wrongly infer that this variable is an `uint8`. Such error during type inference could cause both false positives and false negatives. However, this type of issue occurs only when the developer uses bit manipulation with a mask similar to what the Solidity compiler generate. We find that such a pattern is rare enough not to skew our data, and give an estimate the possible number of contracts which could follow such a pattern in Section 5.5.

4.6 Unrestricted Action

Unrestricted actions is a broad class of vulnerability, as it can include the ability to set an owner without being allowed to, destruct a contract without permission or yet execute arbitrary code. As one of our main goal is to check the exploitation of vulnerable contracts, we stay close to the definitions given by previous works [32] and focus on unrestricted Ether transfer using `CALL`, unrestricted writes using and `SSTORE`, and code injection using `DELEGATECALL` or `CALLCODE`.

Contract address	Last transaction	Amount exploited
0xd654bdd32fc99471455e86c2e7f7d7b6437e9179	2016-06-10	5,885
0x675e2c143295b8683b5aed421329c4df85f91b33	2015-12-31	50.49
0xcd3e727275bc2f511822dc9a26bd7b0bbf161784	2017-03-25	10.34

Figure 8: RE: Top contracts victim of re-entrancy attack and ETH amounts exploited

First, we need to remind ourselves that the caller, unlike for example the call data, cannot be forged. Therefore, one of the main insight is that if an action is restricted depending on who is calling, there should be an execution trace before the restricted operation which conditionally jumps, depending on the caller. This is enough for `SELFDESTRUCT` but not for other instructions as it would flag a line such as `balances[msg.sender] = msg.value` to be vulnerable. To model this, we track whether the message sender influences the storage key or the address to call. Finally, for code injection, we check whether the passed data influences the address called by `DELEGATECALL` or `CALLCODE`.

Analysis correctness. Our analysis for unrestricted actions is neither sound nor complete. We take a relatively simple approach of checking whether the message sender influences a condition or not before executing a sensitive instruction. This can result in false negatives because the check could be performed inappropriately, for example not reverting the transaction when needed, making the analysis unsound. Furthermore, there might be some use cases where it is acceptable to allow any sender to write to the storage, but our analysis would flag such as vulnerable, resulting in false positives. We discuss the implications further in Section 5.6.

5 Analysis of Individual Vulnerabilities

As described in Section 3, the combined amount of Ether contained within *all* the vulnerable contracts exceeds 3 million ETH, worth 600 million USD. In this section, we present the results for each vulnerability one by one; our results have been obtained using the methodology described in Section 4; the goal is to show how much of this money is actually at risk.

Methodology. For each vulnerability, we perform our analysis in two steps. First, we fetch the execution traces of all the transactions up to block 10,200,000 affecting the contracts in our dataset, either directly or through internal transactions. We then run our tool to automatically find the total amount of Ether at risk and report this number. This is the amount we use to later give a total upper bound across all vulnerabilities.

In the second step, we manually analyze the contracts at risk to obtain more insight about the exploits and find interesting patterns. As analyzing all the contracts manually would be impractical, for each vulnerability we manually analyze the contracts with the highest amount of Ether at risk to understand better the reasons behind the vulnerabilities. We then present interesting findings as short case studies.

Runtime performance. Our analysis runs in linear time and memory with respect to the number of instructions executed by a given transaction. The number of instructions varies widely between transactions, from a few hundreds to a few hundred thousands, with an average of around 100,000. Our tool takes on average less than 10ms (stddev. 20ms) per transaction with a maximum of less than 2 seconds for the largest transactions, which is below the timeout of 5 seconds which we set for a single transaction.

5.1 RE: Re-Entrancy

There are 4,337 contracts flagged as vulnerable to re-entrancy by [31, 35, 51], with a total of 457,073 transactions. After running the analysis described in Section 4 on all the transactions, we found a total of 116 contracts which contain re-entrant calls. To look for the monetary amount at risk, we compute the sum of the Ether sent between two contracts in transactions containing re-entrant calls. The total amount of Ether exploited using re-entrancy is of 6,076 ETH, which is considerable as it represents more than 1,200,000 USD.

Manual analysis. We manually analyze the top contracts in terms of fund lost and present them in Figure 8. Interestingly, one of these three potential exploits has a substantial amount of Ether at stake: 5,881 ETH, which corresponds to around 1,180,000 USD. This address has already been detected as vulnerable by some recent work focusing on re-entrancy [43]. It appears that the contract, which is part of the Maker DAO [9] platform, was found vulnerable by the authors of the contract, who themselves performed an attack to confirm the risk [2].

Sanity checking. We use two different contracts for sanity checking. First, we look at TheDAO attack, which is the most famous instance of a re-entrancy attack. Our tool detects the following re-entrancy pattern: the malicious account calls TheDAO main account, TheDAO main account calls into the reward account and the reward account sends the reward to the malicious account, allowing it to perform the re-entrant call into TheDAO main account.

As another sanity check, we look at a contract called SpankChain [6], 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.

Finally, we note that our tool finds all the re-entrancy patterns presented by Sereum [43], including delegated and create-based re-entrancy[2].

Contract address	Amount at risk
0x7011f3edc7fa43c81440f9f43a6458174113b162	56.70
0xb336a86e2feb1e87a328fcb7dd4d04de3df254d0	42.27
0xdcabd383a7c497069d0804070e4ba70ab6ecdd51	33.44
0xfd2487cc0e5dce97f08be1bc8ef1dce8d5988b4d	21.60
0x9e15f66b34edc3262796ef5e4d27139c931223f0	10.50

Figure 9: UE: Top contracts affected by unhandled exceptions and ETH amounts at risk

5.2 UE: Unhandled Exceptions

There are 11,427 contracts flagged vulnerable to unhandled exceptions by [31, 35, 51] for a total of more than 3.4 million transactions, which is *an order of magnitude* larger than what we found for re-entrancy issues.

We find a total of 264 contracts where failed calls have not been checked for, which represents roughly 2% of the flagged contracts. The next goal is to find an upper bound on the amount of Ether at risk because of these unhandled exceptions. We define the upper bound on the money at risk to be the minimum value of the balance of the contract at the time of the unhandled exception and the total of Ether which have failed to be sent. We then sum the upper bound of all issues found to obtain a total upper bound. This gives us a total of 271.89 Ether at risk for unhandled exceptions.

Manual analysis. We manually analyze the top contracts and summarize their addresses and the amount at risk in Figure 9. The Solidity code is available for three of these contracts. We confirm that in all cases the issue came from a misuse of a low-level Solidity function such as `send`.

Investigation of the contract at
0x7011f3edc7fa43c81440f9f43a6458174113b162:

The contract 0x7011f3edc7fa43c81440f9f43a6458174113b162 has failed to send a total of 52.90 Ether and currently still holds a balance of 69.3 Ether at the time of writing. After investigation, we find that the contract is an abandoned pyramid scheme [5]. The contract has a total of 21 calls which failed, all trying to send 2.7 Ether, which appears to have been the reward of the pyramid scheme at this point in time. Unfortunately, the code of this contract was not available for further inspection but we conclude that there is a high chance that some of the users in the pyramid scheme did not correctly obtain their reward because of this issue.

[2] https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns

Contract address	First issue	Balance
0x3da71558a40f63b960196cc0679847ff50fad22b	2016-09-06	13,818
0xd79b4c6791784184e2755b2fc1659eaab0f80456	2016-05-03	2,013
0xf45717552f12ef7cb65e95476f217ea008167ae3	2016-03-15	1,064

Figure 10: TOD: Top contracts potentially victim of transaction ordering dependency attack.

5.3 LE: Locked Ether

There are 7,285 contracts flagged vulnerable to locked Ether by [51], [24], [39] and [31]. The contracts hold a total value of more than 1.4 million ETH, which is worth more than 200 million USD. We analyze the transactions of the contracts that could potentially be locked by conducting the analysis described in the previous section. 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. We note that our tool currently only covers dependency on a destructed contract as a reason for locked Ether and patterns such as unbounded mass operation are not yet covered.

Parity wallet. Contracts affected by the Parity wallet (0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4) bug [14] were not flagged by the tools we analyzed, and are therefore not present in our dataset. As this is one of the most famous cases of locked Ether, we test our tool on the contracts affected by this bug. To find the contracts, we simply have to use the Datalog query for locked Ether in Figure 7c 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 [23], with the contract at address 0x3bfc20f0b9afcace800d73d2191166ff16540258 having as much as 306,276 ETH locked.

5.4 TO: Transaction Order Dependency

There are 1,881 contracts flagged vulnerable to transaction ordering dependency by [35] and [31]. We run the analysis described in Section 4 on their 3,002,304 transactions and obtain a total of 54 contracts potentially affected by transaction-order dependency. To estimate the amount of Ether at risk, we sum up the total value of Ether sent, including by internal transactions, during all the flagged transactions, resulting in a total of 297.2 ETH at risk of transaction-order dependency.

Manual analysis. For each contract, we find the block where transaction order dependency could have happened with the highest balance and report top with their balance at the time of the issue in Figure 10. We manually investigated the contracts listed, they all had their source code available. We confirmed that in all the contracts, it was possible for a user to read and write to the same storage location within a single block. We inspected further 0x3da71558a40f63b960196cc0679847ff50fad22b, a contract called `WithDrawChildDAO` and found that the read was simply for users to check their balance, making the issue benign.

5.5 IO: Integer Overflow

There are 2,472 contracts flagged vulnerable to integer overflow, which accounts for a total of more than 1.2 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 Solidity compiler. To ensure that the contracts we analyze were compiled using Solidity, we fetched all the available source codes for contracts flagged vulnerable to integer overflow from Etherscan [7]. Out of 2,492 contracts, 945 had their source code available and all of them were written in Solidity.

Effects of our formulation. As mentioned in Section 4.5, some types of bit manipulation in Solidity contracts which could result in our type inference heuristic failing. We use the source codes we collected here to verify up to what extent this could affect our analysis. We find that bit manipulation by itself is already fairly rare in Solidity, with only 244 out of the 2,492 contracts we collected using any sort of bit manipulation. Furthermore, most of the contracts using bit manipulation were using it to manipulate a variable as a bit array, and only ever retrieved a single bit at a time. Such a pattern does not affect our analysis. We found only 33 contracts which used `0xFF` or similar values, which could actually affect our analysis. This represents about 1.3% of the number of contracts for which the source code was available.

We find a total of 62 contracts with transactions where an integer overflow might have occurred. To find the amount of Ether at stake, we analyze all the transactions which resulted in integer overflows. We approximate the amount by summing the total amount of Ether transferred in and out during a transaction containing an overflow. We find that the total of Ether at stake is 1,842 ETH. This is most likely an over-approximation but we use this value as our upper-bound.

Manual analysis. We 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. We highlight one of such cases in the following investigation.

Investigation of the contract at
0xdcabd383a7c497069d0804070e4ba70ab6ecdd51:

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

Vulnerable				Exploited contracts		Exploited Ether	
Vuln.	**Vulnerable contracts**	**Total Ether at stake**	**Transactions analyzed**	**Contracts exploited**	**% of contracts exploited**	**Exploited Ether**	**% of Ether exploited**
RE	4,337	1,518,067	457,073	116	2.68%	6,076	0.40%
UE	11,427	419,418	3,400,960	264	2.31%	271.9	0.068%
LE	7,285	1,416,086	10,660,066	0	0%	0	0%
TO	1,881	302,679	3,002,304	54	3.72%	297.2	0.091%
IO	2,492	602,980	1,295,913	62	2.49%	1,842	0.31%
UA	5,163	580,927	3,871,770	42	0.813%	0	0%
Total	23,327	3,124,433	20,241,730	463	1.98%	8,487	0.27%

Figure 11: Understanding the exploitation of potentially vulnerable contracts.

5.6 Unrestricted Action

There is a total of 5,163 contracts flagged by [32, 39, 51] as vulnerable to unrestricted actions for a total of 3,871,770 transactions. We use the approach described in Section 4.6 and find a total of 42 contracts having suffered of unrestricted actions, which were all non-restricted self-destructs, but none of them held Ether at the time of the exploit.

Effects of our formulation. As mentioned in Section 4.6, this analysis is not sound, which means we need to be cautious about false positives. A contract could have a check on the message sender which is incorrect and be exploited but not be flagged as such. While we hypothesize that it is an edge case, it cannot be completely excluded. However, having an automation method for such a check requires knowing the intent of the programmer, for example through specifications, which is out-of-scope of this work. We therefore decide to inspect the contracts in our dataset in more details to understand better the level of exploitation.

Manual analysis. The tool teEther flags *exploitable* contracts, as opposed to simply *vulnerable* contracts. Therefore, expect these contracts to be more likely to have been exploited and focus on these for our manual analysis. We fetch all the historical balances of teEther contracts and retrieve the maximum amount held at any point in time and find the total of these to equal 4,921 Ether. However, we find that 4,867 Ether belonged to 48 different contracts with the exact same bytecode, and all had the same transaction pattern, which we describe in the following investigation.

Investigation of the contract at
`0xac54413f686927054a56d35415ba49618634e105`:

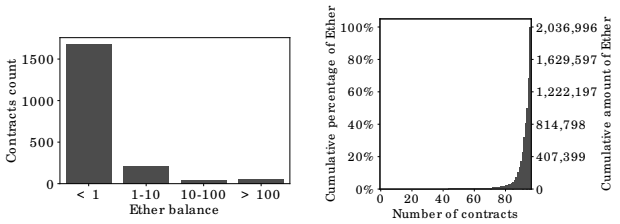
All contracts with a high historical monetary value found by teEther share the same bytecode, creator and transaction pattern as this contract. The contracts are created by `0x15f889d2469d1be0e0699632d8d448f2178a7afe`, receive Ether from Kraken, an exchange, and send the same amount to `0xd1bf1706306c7b667c67ffb5c1f76cc7637685bd` a couple of blocks later. We could not find further information about these addresses. We decompile the contract to understand how the contracts were exploitable and find that during the few blocks they held money, exploiting the contract would have been as simple as sending a transaction with the address to which to transfer the funds as argument. This is a very dangerous situation but because the Ether was usually sent within a minute to another address, an attacker would have needed to be very proactive and use advance tooling to exploit the contract. This illustrates well how a contract can be *exploitable* but have little chance of being *exploited* in practice.

Sanity checking. As a sanity check, in addition to our test suite, we use one of the most famous instance of an unrestricted action: the destructed Parity wallet library contract at address `0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4`. We analyze the transactions and successfully find an unrestricted store instruction in transaction `0x05f71e1b`, which was used to take control of the wallet.

5.7 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 11. The *Contracts exploited* column indicates the number of contracts which are flagged exploited and *% Contracts exploited* is the percentage of this number with respect to the number of contracts flagged vulnerable. The *Exploited Ether* column shows the maximum amount of Ether that 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 *number of contracts exploited* is non negligible, with about 2% to 4% of vulnerable contracts exploited for 4 of the 6 vulnerabilities covered in our study. However, it is important to note that the percentage of Ether exploited is an order of magnitude lower, with at most 0.4% of the Ether at stake exploited for re-entrancy. This indicates that exploited contracts are usually low-value. We will expand on this argument further in Section 7.



(a) Ether held by contracts in our dataset with non-zero balance.

(b) Cumulative Ether held in the 96 contracts in our dataset containing at least 10 ETH.

Figure 12: Ether held in contracts: describing the distribution.

6 Limitations

In this section, we present the different limitations of our system, and describe how we try to mitigate them.

Soundness vs Completeness. As for most tools such as this one, we are faced with the trade-off of soundness against completeness. Whenever possible we choose soundness over completeness — three out of six of our analyses are sound. When we cannot have a sound analysis, we are careful to only leave out cases which are unlikely to generate many false negatives. In other words, we try as much as possible to reduce the number of false negatives, even if this means increasing the number of false positives. Indeed, the main goal of our system is to provide us an upper-bound of the amount of potentially exploited Ether, which make false negatives undesirable. Furthermore, we manually check the high-value contracts flagged as exploited, false-positives will not have an important influence on the final results. As an example of this trade-off, for re-entrancy we flag any contract which was called using a re-entrant call and lost funds in the process. However, in some cases, it could be an expected behavior and the funds could have been transferred to an address belonging to the same entity.

Dataset. We only run our tool on the contracts included in our dataset, which means that we might be missing some exploits which actually occurred. Indeed, we did not have any contract affected by the Parity wallet bug nor had we the contract affected by TheDAO hack in the dataset. However, one of the main goal of this paper is to quantify what fraction of vulnerabilities discovered by analysis tools is exploited in practice and our current approach allows us to do exactly this.

Other types of attacks. Our tool and analysis does not cover every existing attack to smart contracts. There are, for example, attacks targeting ERC-20 tokens [42], or yet some other types of DoS attacks, such as wallet griefing [24]. Furthermore, some detected "exploits" could be the results of Honeypots [50] but our tool does not cover such cases. Although it would be interesting to also cover such cases, we had to make a decision about the scope of the tool. Therefore, we focus on the vulnerabilities which have been the most covered in the literature, which we hypothesise is representative of how common the vulnerabilities are.

7 Discussion

Even considering the limitations of our system, it is clear that the exploitation of smart contracts is vastly lower that what could be expected. In this section, we present some of the factors 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 23,327 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 12a. 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 12b. 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.

To make sure this fact generalizes to the whole Ethereum blockchain and not only our dataset, we fetch the balances of all existing contracts. This gives a total of 15,459,193 contracts. Out of these, only 463,538 contracts have a non-zero balance, which is merely 3% of all the contracts. Out of the contracts with a non-zero balance, the top 10, top 100 and top 1000 account respectively for 54%, 92% and 99% of the total amount of Ether. This shows that our dataset follows the same trend as the whole Ethereum blockchain: a very small amount of contracts hold most of the wealth.

Manual inspection of high value contracts. We decide to manually inspect the top 6 contracts, in terms of balance at the time of writing, marked as vulnerable by any of the tools in our dataset. 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. We give an overview of the findings here and a more in-depth version in Appendix A.

Investigation of the contract at
`0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae`:

The source code for this contract is not available on Etherscan. However, we discovered that it is the multi-signature wallet of the Ethereum foundation [1] and that its source code is available on GitHub [3]. We inspect the code and find that all calls 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.

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

We were able to inspect 4 of the 5 other 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 [4]. We did not find any particular issue, but it does use a delegate pattern which explains the locked Ether vulnerability marked by Zeus.

We present a thorough investigation of the high-value contracts in Appendix A. Overall, all the contracts from Figure 13 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 8 where contracts with high 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 [24, 31].

8 Related Work

Some major smart contracts exploits have been observed on Ethereum in recent years [45]. These attacks have been analyzed and classified [11] and many tools and techniques have emerged to prevent such attacks [21, 26]. Recent literature has also shown how attacks on Ethereum are evolving with time [55]. 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.

8.1 Motivating Large-scale Exploits

TheDAO exploit. TheDAO exploit [45] is one of the most infamous bugs on the Ethereum blockchain. Attackers exploited a re-entrancy vulnerability [11] 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 [14] 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.

8.2 Analyzing and Verifying Smart Contracts

There have 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 and, when relevant, describe how they compare to our work.

Analysis tools can roughly be divided in two categories: static analysis and dynamic analysis tools. Using the term "static" quite loosely, static analysis tools can be defined as tools which catch bugs or vulnerabilities without the need to deploy the smart contracts. Runtime analysis tools try to detect these by executing the deployed contracts. Our tool fits into the second category.

Static analysis tools. Static analysis tools have been the main focus of research. This is understandable, given how critical it is to avoid vulnerabilities in a deployed contract. Most of these tools work by analyzing the bytecode or high-level code of contracts and checking for known vulnerable patterns.

Oyente [35] is one of the first tools which has been developed to analyze smart contracts. It uses symbolic execution in combination with the Z3 SMT solver [20] to check for the following vulnerabilities: transaction ordering dependency, re-entrancy and unhandled exceptions.

ZEUS [31] is a static analysis tool which works on the Solidity smart contract and not on the bytecode, making it appropriate to assist development efforts rather than to analyze deployed contracts, for which Solidity code is typically

not available. Zeus transpiles XACML-styled [46] policies to be enforced and the Solidity contract code into LLVM bitcode [33] and uses constrained Horn clauses [13, 36] over it to check that the policy is respected.

Securify [51] is a static analysis tool which checks security properties of the EVM bytecode of smart contracts. It encodes security properties as patterns written in a Datalog-like [52] domain-specific language, and checks either for compliance or violation. Securify infers semantic facts from the contract and interprets the security patterns to check for their violation or compliance by querying the inferred facts. This approach has many similarities with ours, using Datalog to express vulnerability patterns. The major difference is that Securify works on bytecode while our tool works on execution traces.

MadMax [24] has similarities with Securify, as it also encodes properties of the smart contract into Datalog, but it focuses on vulnerabilities related to gas. It is the first tool to detect "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 [15] and is performant enough to analyze all the contracts of the Ethereum blockchain in only 10 hours.

Several other static analysis tools have been developed, some, such as SmartCheck [48], being quite generic and handling many classes of vulnerabilities, and other being more domain specific, such as Osiris [49] focusing on integer overflows, Maian [39] on unrestricted actions or Gasper [17] on costly gas patterns. More recently, ETHBMC [22] was designed to also support inter-contract relations, cryptographic hash functions and memcopy-style operations.

Finally, there have also been some efforts to formally verify smart contracts. [28] is one of the first efforts in this direction and defines the EVM using Lem [38], which allows to generate definitions for theorem provers such as Coq [12]. [25] presents a complete small-step semantics of EVM bytecode and formalizes it using the F* proof assistant [47]. A similar effort is made in [27] to give an executable formal specification of the EVM using the K Framework [44]. VerX [41] is also a recent work allowing users to write properties about smart contracts which will be formally verified by the tool.

Dynamic analysis tools. Although dynamic analysis tools have been less studied than their static counterpart, some work has emerged in recent years.

One of the first work in this line is ContractFuzzer [30]. As its name indicates, it uses fuzzing to find vulnerabilities in smart contracts and is capable of detecting a wide range of vulnerabilities such as re-entrancy, locked Ether or unhandled exceptions. The tool generates inputs to the contract and checks using an instrumented EVM whether some vulnerabilities have been triggered. An important limitation of this fuzzing approach is that it requires the Application Binary Interface of the contract, which is typically not available for contracts deployed on the main Ethereum network.

Sereum [43] focuses on detecting re-entrancy exploitation at runtime by integrating checks in a modified Go Ethereum client. The tool analyzes runtime traces and uses taint analysis to ensure that no variable accessing the contract storage is used in a re-entrant call. Although there are some similarities with our tool, also analyzing traces at runtime, Sereum focuses on re-entrancy while our tool is more generic, notably because vulnerabilities pattern can easily be expressed using Datalog.

teEther [32] also works at runtime but is different from the previous works presented, as it does not try to protect contracts but rather to actively find an exploit for them. It first analyzes the contract bytecode to look for critical execution paths. Critical paths are execution paths which may result in lost funds, for example by sending money to an arbitrary address or being destructed by anyone. To find these paths, it uses an approach close to Oyente [35], combining symbolic execution and Z3 to solve path constraints.

TXSPECTOR [54], which was published soon after the first version of this paper, uses a very similar approach to ours to detect re-entrancy, unchecked call and suicidal contracts. They also leverage a Datalog approach to detect vulnerabilities but first transforms the transaction traces into a flow graph rather than adding facts about traces directly to the Datalog database. While this does add expressiveness, it makes the analysis significantly more complex, resulting in some analysis timing out on some transactions. Therefore, we believe that their approach could be complementary to ours and used to eliminate potential false-positives of our approach.

Summary. Static analysis tools are typically designed to detect *vulnerable* contracts, while dynamic analysis tools are designed to detect *exploitable* contracts. The only exception is Sereum, which detects contracts *exploited* using re-entrancy. Our work is, to the best of our knowledge, the first attempt to detect contracts *exploited* using a wide range of vulnerabilities. This is mostly orthogonal with other works and can support analysis tool development efforts by helping to understand what type of exploitation is happening in the wild.

9 Conclusion

In this paper, we surveyed the 23,327 vulnerable contracts reported by six 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 20 million transactions executed by these contracts. We found that at most 463 out of 23,327 contracts have been subject to exploits but that at most 8,487 ETH (1.7 million USD), or only 0.27% of the 3 million ETH (600 million USD) potentially at risk, was exploited. Finally, we found that a majority of Ether is held by only a small number of contracts and that the vulnerabilities reported on these are either false positives or not exploitable in practice, thus providing a reasonable explanation for our results.

References

[1] 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/, 2015. [Online; accessed 13-October-2020].

[2] 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/, 2016. [Online; accessed 13-October-2020].

[3] Source code of the Ethereum Foundation Multisig wallet. https://github.com/ethereum/dapp-bin/blob/master/wallet/wallet.sol, 2017. [Online; accessed 13-October-2020].

[4] The DAO Refunds. https://theethereum.wiki/w/index.php/The_DAO_Refunds, 2017. [Online; accessed 13-October-2020].

[5] What's become of the ethereumpyramid? https://www.reddit.com/r/ethtrader/comments/7eimrs/whats_become_of_the_ethereumpyramid/, 2017. [Online; accessed 13-October-2020].

[6] We Got Spanked: What We Know So Far. https://medium.com/spankchain/we-got-spanked-what-we-know-so-far-d5ed3a0f38fe, 2018. [Online; accessed 13-October-2020].

[7] Etherscan — Ethereum (ETH) Blockchain Explorer. https://etherscan.io, 2019. [Online; accessed 13-October-2020].

[8] golem — Computing Power. Shared. https://golem.network/, 2019. [Online; accessed 13-October-2020].

[9] MakerDAO. https://makerdao.com/en/, 2019. [Online; accessed 13-October-2020].

[10] Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum, 2019. [Online; accessed 13-October-2020].

[11] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, 2017.

[12] 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.

[13] Nikolaj Bjørner, Ken Mcmillan, Andrey Rybalchenko, and Technische Universität München. Program verification as satisfiability modulo theories. In *In SMT*, 2012.

[14] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An In-Depth Look at the Parity Multisig Bug. http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/, 2017. [Online; accessed 13-October-2020].

[15] 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. *CoRR*, abs/1809.03981, 2018.

[16] Vitalik Buterin. A next-generation smart contract and decentralized application platform. *Ethereum*, (January), 2014.

[17] 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*, 2017.

[18] ConsenSys. Mythril Classic. https://github.com/ConsenSys/mythril-classic, 2019. [Online; accessed 13-October-2020].

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

[20] 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*. Springer, 2008.

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

[22] Joel Frank, Cornelius Aschermann, and Thorsten Holz. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium*, August 2020.

[23] Max Galka. Multisig wallets affected by the Parity wallet bug. https://github.com/elementus-io/parity-wallet-freeze, 2017. [Online; accessed 13-October-2020].

[24] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, (OOPSLA), October 2018.

[25] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *Principles of Security and Trust*, Cham, 2018.

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

[27] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium*, 2018.

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

[29] Neil Immerman. *Descriptive complexity*. Graduate texts in computer science. Springer, 1999.

[30] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of 25th Annual Network & Distributed System Security Symposium*, 2018.

[32] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium*, August 2018.

[33] 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*. IEEE Computer Society, 2004.

[34] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, and Prateek Saxena. Oyente Benchmarks. https://oyente.tech/benchmarks/, 2016. [Online; accessed 13-October-2020].

[35] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

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

[37] 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), 2019.

[38] 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. ACM, 2014.

[39] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[40] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in EVM. In *Proceedings of 27th Annual Network & Distributed System Security Symposium*, 2020.

[41] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy*, 2020.

[42] R. Rahimian, S. Eskandari, and J. Clark. Resolving the multiple withdrawal attack on erc20 tokens. In *2019 IEEE European Symposium on Security and Privacy Workshops*, June 2019.

[43] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Proceedings of 26th Annual Network & Distributed System Security Symposium*, February 2019.

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

[45] 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.

[46] Remon Sinnema and Erik Wilde. eXtensible Access Control Markup Language (XACML) XML Media Type. https://tools.ietf.org/html/rfc7061, 2013. [Online; accessed 13-October-2020].

[47] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *SIGPLAN Not.*, 46(9):266–278, September 2011.

[48] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, May 2018.

[49] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[50] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *28th USENIX Security Symposium*, August 2019.

[51] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

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

[53] Gavin Wood. Ethereum yellow paper. http://gavwood.com/paper.pdf, 2014. [Online; accessed 13-October-2020].

[54] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium*, August 2020.

[55] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Zhemin Yang, and Yuan Zhang. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In *29th USENIX Security Symposium*, August 2020.

A Investigations

In this appendix, we will give a more in-depth security analysis of the top value contracts we presented in Section 7. In particular, we will focus on the vulnerabilities detected by the different tools and show how it could, or not, affect the contract.

0xde0b295669a9fd93d5f28d9ec85e40f4cb697bae

This contract has been flagged as being vulnerable to reentrancy by Oyente. For a contract to be victim of a reentrancy attack, it must `CALL` another contract, sending it enough gas to be able to perform the re-entrant call. In Solidity terms, this is means that the contract has to invoke `address.call` and not explicitly set the gas limit. By looking at the source code [3], we find 2 such instances: one at line 352 in the `execute` function and another at line 369 in the `confirm function`. The `execute` is protected by the `onlyowner` modifier, which requires the caller to be an owner of the wallet. This means that for a re-entrant call to work, the malicious contract would need to be an owner of the wallet in order to work. The `confirm` function is protected by the `onlymanyowners` modifier, which requires at least n owners to agree on confirming a particular transaction before it is executed, where n is agreed upon at contract creation time. Furthermore, `confirm` will only invoke `address.call` on a transaction previously created in the `execute` function.

0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9

This is the contract for multi-signature wallet of the Golem project [8] and uses a well-known multi-signature implementation. We use the source code available on Etherscan to perform the audit. This contract is marked with locked Ether by MadMax and integer overflow by Zeus.

We first focus on the locked Ether which is due to an unbounded mass operation [24]. An unbounded mass operation is flagged when a loop is bounded by a variable which value could increase, for example the length of an array. This is because if the number of iteration becomes too large the contract would run out of gas every time, which could indeed result in locked funds. Therefore, we check all the loops in the contract. There are 8 loops in the code, at lines 43, 109, 184, 215, 234, 246, 257 and 265. All the loops except the ones at lines 257 and 265 are bound by the total number of owners. As owner can only be added if enough existing owners agree, running out-of-gas when looping on the number of owners cannot happen unless the owners agree. The loops at lines 257 and 265 are in a function called `filterTransactions` and are bounded by the number of transactions. The function `filterTransactions` is only used by two external getters, `getPendingTransactions` and `getExecutedTransactions` and could therefore not result the Ether getting lock. However, as the number of transactions is ever increasing, if the owner submit enough transactions, the `filterTransactions` function could indeed need to loop over too many transactions and end up running out-of-gas on every execution. We estimate the amount of gas used in the loop to be around 50 gas, which means that if the number of transactions reaches 100,000, it would required more than 5,000,000 gas to list the transactions, which would probably make all calls run out of gas. The contract has only received a total of 281 transactions in more than 3 years so it is very unlikely that the number of transactions increase this much. Nevertheless, this is indeed an issue which should be fixed, most likely by limiting the maximum numbers of transactions that can be retrieved by `getPendingTransactions` and `getExecutedTransactions`.

Next, we look for possible integer overflows. All loops discussed above use an `uint` as a loop index. In Solidity, `uint` is a `uint256` which makes it impossible to overflow here, given than neither the number of owners or transactions could ever reach such a number. The only other arithmetic operation performed is `owners.length - 1` in the function `removeOwner` at line 103. This function checks that the owner exists, which means that `owners.length` will always be at least 1 and `owners.length` can therefore never underflow.

0x851b7f3ab81bd8df354f0d7640efcd7288553419

This contract is also a multi-sig wallet, this time owned by Gnosis Ltd.[3] We use the source code available on Etherscan to perform the audit. The contract looks very similar of the one used by `0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9` and has also been marked by MadMax as being vulnerable to locked Ether because of unbounded mass operations. Again, we look at all the loops in the contract and find that as the previous contract, it loops exclusively on owners and transactions. As in the previous contract, we assume looping on the owners is safe and look at the loops over the transactions. This contract has two functions looping over transactions, `getTransactionCount` at line 303 and `getTransactionIds` at line 351. Both functions are getters which are never called from within the contract. Therefore, no Ether could ever be locked because of this. Unlike the previous contract, `getTransactionIds` allows to set the range of transactions to return, therefore making the function safe to unbounded mass operations. However, `getTransactionCount` does loop over all the transactions, and as before, could therefore become unusable at some point, although it is highly unlikely.

0xcafe1a77e84698c83ca8931f54a755176ef75f2c

This contract is again a multi-sig wallet, this time owned by the Aragon project[4]. We use the contract published on Etherscan for the audit. The source code for this contract is exactly the same as `0x851b7f3ab81bd8df354f0d7640efcd7288553419`, except that it misses a contract called `MultiSigWalletWithDailyLimit`. This contract was also flagged as being at risk of unbounded mass operations by MadMax, the conclusions are therefore exactly the same as for the previous contract.

0xbf4ed7b27f1d666546e30d74d50d173d20bca754

This contract is the only one which is very different from the previous ones. It is the `WithdrawDAO` contract, which has been created for users to get their funds back after TheDAO incident [45]. We use the source code from Etherscan to audit the contract. This contract has been flagged with several vulnerabilities: Securify flagged it with transaction order dependency and unhandled exception, while Zeus flagged it with locked ether and integer overflow. The contract has two very short functions: `withdraw` which allows users to convert their TheDAO tokens back to Ether, and the `trusteeWithdraw` which allows to send funds which cannot be withdrawn by regular users to a trusted address. We first look at the transaction order dependency. As any user will only ever be able to receive the amount of tokens he possesses, the order of the transaction should not be an issue in this contract. We then look at unhandled exceptions. There is indeed a call to `send` in the `trusteeWithdraw` which is not checked. Although it is not particularly an issue here, as this does not modify any other state, an error should probably be thrown if the call fails. We then look at locked ether. The contract is flagged with locked ether because of what Zeus classifies as "failed send". This issue was flagged because if the call to `mainDAO.transferFrom` always raised, then the call to `msg.sender.send` would never be reached, indeed preventing from reclaiming funds. However, in this context, `mainDAO` is a trusted contract and it is therefore safe to assume that `mainDAO.transferFrom` will not always fail. Finally, we look at the integer overflow issue. The only place where an overflow could occur is in `trusteeWithdraw` at line 23. This could indeed overflow without some assumptions on the different values. For this particular contract, the following assumptions are made.

```
this.balance+mainDAO.balanceOf(this)≥mainDAO.totalSupply()
mainDAO.totalSupply()>mainDAO.balanceOf(this)
```

As long as these assumptions hold, which was the case when the contract was deployed, this expression will never overflow. Indeed, if we note t the time before the first call to `trusteeWithdraw` and $t+1$ the time after the first call, we have

```
this.balance_{t+1} = this.balance_t - (
  this.balance_t + mainDAO.balanceOf(this)
                 - mainDAO.totalSupply())
 = -mainDAO.balanceOf(this)+mainDAO.totalSupply()
```

meaning that every subsequent call will compute:

```
this.balance_{t+1} + mainDAO.balanceOf(this) -
                       mainDAO.totalSupply()
 = -mainDAO.balanceOf(this)+mainDAO.totalSupply()+
   mainDAO.balanceOf(this) - mainDAO.totalSupply()
 = 0
```

This will always result in sending 0 and will therefore not cause any overflow. If some money is newly received by the contract, the amount received will be transferred the next time `trusteeWithdraw` is called.

[3] https://gnosis.io/
[4] https://aragon.org/