eTainter: Detecting Gas-Related Vulnerabilities in Smart Contracts

Asem Ghaleb
University of British Columbia
Vancouver, Canada
aghaleb@alumni.ubc.ca

Julia Rubin
University of British Columbia
Vancouver, Canada
mjulia@ece.ubc.ca

Karthik Pattabiraman
University of British Columbia
Vancouver, Canada
karthikp@ece.ubc.ca

ABSTRACT

The execution of smart contracts on the Ethereum blockchain consumes gas paid for by users submitting contracts' invocation requests. A contract execution proceeds as long as the users dedicate enough gas, within the limit set by Ethereum. If insufficient gas is provided, the contract execution halts and changes made during execution get reverted. Unfortunately, contracts may contain code patterns that increase execution cost, causing the contracts to run out of gas. These patterns can be manipulated by malicious attackers to induce unwanted behavior in the targeted victim contracts, e.g., Denial-of-Service (DoS) attacks. We call these gas-related vulnerabilities. We propose *eTainter*, a static analyzer for detecting gas-related vulnerabilities based on taint tracking in the bytecode of smart contracts. We evaluate *eTainter* by comparing it with the prior work, MadMax, on a dataset of annotated contracts. The results show that *eTainter* outperforms MadMax in both precision and recall, and that *eTainter* has a precision of 90% based on manual inspection. We also use *eTainter* to perform large-scale analysis of 60,612 real-world contracts on the Ethereum blockchain. We find that gas-related vulnerabilities exist in 2,763 of these contracts, and that *eTainter* analyzes a contract in eight seconds, on average.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*;

KEYWORDS

Ethereum, Solidity, security, taint analysis

ACM Reference Format:
Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: Detecting Gas-Related Vulnerabilities in Smart Contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534378

1 INTRODUCTION

Smart contracts supported by the Ethereum blockchain have witnessed a dramatic rise in interest. Similar to traditional programs, smart contracts are defined by a set of functions, each consisting of a sequence of instructions. The contract's functions are invoked through calls called transactions. Ethereum smart contracts are written using high-level programming languages (e.g., Solidity) [12], and compiled to Ethereum Virtual Machine (EVM) bytecode that is deployed and stored in the blockchain accounts [37].

The execution of a smart contract requires fees, called gas, paid by the user who submits the transaction to execute the contract. The gas is paid by users upon transaction submission. Running out of gas during a transaction's execution results in exceptions and abortion of the transaction, thereby leading to unwanted behaviors. Further, Ethereum imposes an upper bound on the amount of gas spent to execute transactions in each block (i.e., block gas limit). Therefore, when the block gas limit is exceeded during the execution of a transaction within a block, the transaction fails, and its execution gets reverted (i.e., Ethereum rolls back all changes made before transaction failure). This prevents functions that exceed the gas block limit from completing successfully. The problem gets worse if these functions are responsible for transferring Ether out of the contract, as this would prevent the contract owners/users from ever accessing their money (Ether), e.g., in the Governmental contract [31], where $2.5 million worth of Ether got locked out.

Smart contracts may contain vulnerable code patterns that can be exploited by malicious users to force contracts to run out of gas [33]. We refer to these as *gas-related vulnerabilities*. Discovering gas-related vulnerabilities in smart contracts by developers is not straightforward as the gas cost of a contract is based on (1) the cost of the EVM low-level instructions rather than its source code, and (2) the global state of the contract, i.e., data in the contract storage. A recent study [39] found that it is time-consuming to manually identify gas-related vulnerabilities in smart contracts. Further, once a smart contract is deployed, it is difficult to modify it. Therefore, we need automated tools to analyze smart contracts for gas-related vulnerabilities *before deployment* on the blockchain.

There has been little prior work to find gas-related vulnerabilities in smart contracts. Grech et al. proposed a tool called MadMax that statically analyzes smart contracts for gas-related vulnerabilities [17]. Other work [3, 5] has focused on finding inefficient gas code patterns rather than vulnerabilities. However, these tools use pre-specified code templates and rules, and are hence brittle as even small variations in the patterns can make the tools fail, and also result in high false-positives (as we show later in the paper).

We propose an efficient *static-analysis*-based approach, *eTainter* (EVM Tainter), to find gas-related vulnerabilities in smart contracts. Our main insight is that gas-related vulnerabilities are caused by the dependency of contract code on data items that are either provided or manipulated by the contract users. We formulate the detection of gas-related vulnerabilities as a taint analysis problem [25]. We then use static *taint tracking* to find gas-based vulnerabilities, without

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534378

any pre-existing code patterns and rules. The main challenge in static taint analysis for smart contracts is that contracts have certain peculiarities (e.g., taint propagation via contract's storage, multiple entry points, access control), as well as patterns of use (e.g., loops for processing strings and bytes), which must be taken into account to avoid false-positive and false-negative results.

To the best of our knowledge, eTainter *is the first technique to target gas-related vulnerabilities without requiring pre-specified code patterns, via static taint analysis. Our contributions are as follows.*

1. Proposing an inter-procedural static taint-analysis approach for detecting multiple classes of gas-related vulnerabilities, *eTainter*, which tracks taints through the contract's persistent storage and the flow of taints from the contract's multiple entry points, in addition to using domain-specific optimizations to reduce false-positives.

2. Implementing the proposed approach as an automated tool that works on the smart contract's EVM bytecode. Our implementation is publicly available at [16].

3. Evaluating the efficacy of *eTainter* on a custom dataset of 28 smart contracts with hand-annotated gas-related vulnerabilities; a set of 60,612 unique smart contracts deployed on Ethereum; and a set of the 3,000 most frequently used contracts in Ethereum.

The results of evaluating *eTainter* show that *eTainter* is able to find gas-related vulnerabilities with a precision and recall of over 90%. Our comparison with the prior work, MadMax, shows an F1 score of 92% for *eTainter* compared with 69% for MadMax for the same set of contracts, with both precision and recall being higher for *eTainter*. Further, *eTainter* has an analysis time of about 8 seconds on average per smart contract, and it analyzed successfully 88% of the Ethereum contracts without timing out; 97% of these contracts were analyzed within one minute each by *eTainter*. Finally, we find that gas-related vulnerabilities are present in about 5% of the contracts in Ethereum, and in about 2.5% of the most frequently-used contracts.

2 BACKGROUND AND RELATED WORK

2.1 Ethereum Smart Contracts

Ethereum is a distributed computing platform running programs called smart contracts. Smart contracts run in a stack-based Ethereum virtual machine (EVM), which maintains a stack of 256-bit words. Further, in EVM, each contract has access to persistent private a key-value storage of 256-bit keys and 256-bit values, in which data is maintained over multiple executions. We refer to each key-value field in the storage as a storage slot. The smart contract also has access to volatile memory that is initialized at the beginning of each execution. The EVM bytecode deployed on the blockchain gets executed through transactions upon user requests. The bytecode is executed by miners, which are a network of mutually untrusted nodes, and governed by the consensus protocol of the Ethereum blockchain. Miners receive execution fees called gas for running a transaction, paid by the submitting user.

2.2 EVM Bytecode

Unlike traditional programs that have a main entry point, EVM bytecode starts with a function selector, which provides EVM bytecode with multiple entry points. The function selector is like an entry gate for the contract and routes execution to the called external or public function based on matching the static signature of the called function. When calling a contract, if the provided function signature does not match any of the contract's public/external functions, the execution is directed to the fallback function, if any (defined as `function()`). This function is typically used for receiving Ether.

In EVM bytecode, instructions work on data from either the stack, memory, or storage. As the EVM is stack-based, each instruction in the bytecode that takes arguments, pops its arguments from the stack (except for the PUSH instructions that take immediate arguments). In addition to the stack, EVM uses memory as an input and output buffer for a few instructions, such as `SHA3` that computes the keccak256 hash of data in memory and pushes the result to the stack, where the memory start location and size are read from the stack. Further, EVM uses two instructions (`SLOAD` and `SSTORE`) for managing persistent data in the contract storage. Both instructions get the corresponding storage addresses from the stack, and `SLOAD` pushes the loaded data to the stack.

EVM also has instructions for managing data in memory (`MLOAD` and `MSTORE`), querying blockchain state (e.g., `TIMESTASMP`), writing events to the blockchain (e.g., `LOG1`), and obtaining execution environment information (e.g., the caller of the contract (`CALLER`)). Further, EVM involves a set of arithmetic (e.g., `ADD`, `MUL`) and logic (e.g., `GT`, `EQ`) instructions, control transfer instructions (e.g., conditional jump (`JUMPI`)), other contracts' call instructions (e.g., `CALL`), and instructions that end the execution (e.g., `STOP`) or revert changes made to the contract state during execution (`REVERT`). EVM does not have method invocation and return instructions to perform intra-contract function calls. Instead, the EVM pushes the return address to the stack and performs a direct jump to the target address of the method in the bytecode.

2.3 Taint Analysis

Taint analysis is used in *information-flow based security* to determine if there is a data-flow from low-integrity data (i.e., *sources*) to high-integrity data (i.e., *sinks*) [25]. The sources are typically data that can be manipulated by the user (e.g., user input data) and the sinks are typically security-sensitive operations (e.g., writing to a database). A flow of data from sources to sinks represents a vulnerability.

There are generally two forms of tainting: (1) explicit, and (2) implicit. Explicit tainting considers only direct data-flow from sources to sinks, without considering control-flow. Implicit tainting also considers indirect tainting via control-flow and is needed for sound analysis. However, tracking implicit taints is typically more expensive and hence most tainting techniques perform explicit tainting.

Finally, taint analysis can be performed either statically or dynamically. Static taint analysis techniques may be imprecise (i.e., has false-positives), but are typically sound (i.e., cover all potential taint flows in the program and hence have no false-negatives). Dynamic taint analysis typically is precise (i.e., has no false positives), but its coverage is limited by the inputs provided for executing the program, making it unsound (i.e., has false negatives).

2.4 Related Static Analysis Tools

There has been limited prior work on finding gas-related vulnerabilities in smart contracts. One of the first tools to identify gas-related vulnerabilities, MadMax [17], statically analyzes smart contracts for three types of vulnerabilities: (1) unbounded loops, (2) DoS with

Failed Call (wallet griefing), and (3) Induction Variable overflows. MadMax constructs an intermediate representation (IR) from the EVM bytecode of the contract, and uses Datalog-based inference rules for extracting contract properties. However, MadMax has three drawbacks. First, it mainly searches for specific vulnerability patterns rather that reasoning on the causes of the vulnerabilities being detected. Thus, even a slight variation in the code syntax of the gas-related vulnerabilities would not be covered by these rules (Section 3). Second, MadMax coarsely over-approximates many of the vulnerability cases, e.g., considering a loop that references the size of any storage array in its condition as an unbounded loop. This over-approximation results in high false positives (Section 7.2). Finally, MadMax only partially models the contract's storage and memory, and hence overlooks vulnerabilities that depend on memory data items and storage complex structures, leading to false-negatives.

GasReducer [5] is a static analysis tool that performs bytecode optimization by finding unoptimized sequences of EVM instructions and replacing them with optimized sequences. GASPER [4], and its extension GasChecker [3], focus on optimizing gas consumption in smart contracts by detecting gas-inefficient code patterns. However, these tools do not consider gas-related vulnerabilities, and are limited to dead code elimination and simple loop optimizations.

There has been a vast body of work on statically analyzing smart contracts for other classes of security bugs than gas-related vulnerabilities [21], [19], [23], [38], [22], [29], [30] , [2], [27]. Many of these techniques use symbolic execution. As a result, they suffer from the path explosion problem, which makes them unable to explore all transaction sequences [11, 28]. Unfortunately, the detection of gas-related vulnerabilities often requires a complex sequence of transactions to reach the vulnerable state.

GasFuzzer [1] uses fuzzing to generate inputs for smart contracts, and create transactions leading to high gas consumption values. It then analyzes the execution logs for exception disorder vulnerabilities (unhandled exceptions including those caused by insufficient gas). However, GasFuzzer fuzzes individual functions in the smart contracts through single transactions, and many gas-related vulnerabilities only arise when calling multiple functions.

Finally, taint analysis has been used by some tools for finding bugs in smart contracts. However, none of the taint analysis tools consider gas-related issues. Osiris [29] uses static taint analysis to validate arithmetic bugs detected through symbolic execution. Its taint analysis is built on top of symbolic execution and performed on the instructions of the counter examples generated by the symbolic execution, which makes its approach insufficient for detecting gas-related issues as the taint propagation will be limited to the generated counter examples. Sereum [24] uses dynamic taint tracking to protect against Reentrancy attacks by extending the Ethereum client to perform runtime monitoring. However, like all dynamic analysis techniques, this approach has the drawback of finding only the vulnerabilities that are exercised by the provided inputs. Ethainter [2] uses Datalog rules to perform information flow analysis to detect composite vulnerabilities. Unlike *eTainter*, which works on tracking taints in EVM bytecode, Ethainter designed an abstract input language to capture information flow semantics in smart contracts, customized to detect composite vulnerabilities.

A recent paper, SMARTIAN [6], uses fuzzing guided by dynamic and static dataflow analyses to detect a set of smart contract bugs. SMARTIAN performs dynamic dataflow analysis to guide the fuzzing engine and to implement bug oracles. The use of static taint analysis by SMARTIAN is limited to initial seed construction, to find the set of functions in the contract that include sender-checker routines. Further, SMARTIAN does not target gas-related bugs, and extending SMARTIAN to accurately detect gas-related bugs would be challenging. Dynamically detecting unbounded loops, for instance, requires executing a massive number of transactions to reach the vulnerable state and trigger the bug.

3 MOTIVATING EXAMPLE

We use a real-world contract deployed on the Ethereum blockchain to illustrate the challenges of finding gas-related vulnerabilities. We also use this as a running example in Section 4 and Section 5.

```
1 contract PIPOT {
2  uint public fee = 20;
3  mapping(uint => uint) jackpot;
4 struct order{
5   address player;
6   uint betPrice;}
7 mapping(uint => order[]) orders;
8
9  function buyTicket(uint betPrice)public payable{
10     orders[game].push(order(msg.sender , betPrice));
11     uint distribute = msg.value * fee / 100;
12     jackpot[game] += (msg.value - distribute);
13   }
14  function start(uint winPrice)public onlyOwner(){
15     if (orders[game].length > 0) {
16        pickTheWinner(winPrice);}
17     startGame();
18   }
19  function pickTheWinner(uint winPrice) internal {
20    uint toPlayer = jackpot[game]/orders[game].length;
21    for(uint i=0; i<orders[game].length;i++){
22       if (orders[game][i].betPrice == winPrice){
23          orders[game][i].player.transfer(toPlayer);}
24     }
25  }
```

Figure 1: Example of smart contract with two classes of gas-related vulnerabilities.

The code in Figure 1 is simplified from the PIPOT contract on Etherscan [10], an explorer for Ethereum. Due to space constraints, we show only the relevant parts of the code. The contract implements a lottery/game where the users of the contract can participate in the game by buying tickets and guessing a bet price. The winners of the game are the participants who correctly guess the bet price specified by the owner of the contract. Users can participate in the game by calling the function *buyTicket* at line 9. The function *buyTicket* stores the address of each participant and the bet price in the mapping *orders* (line 10), which is a data structure in Solidity similar to a hash table. It also deducts 20% of the bet price as a fee and adds the remaining 80% to the mapping *jackpot* (lines 11-12).

The owner of the contract can start a new game by calling the function *start* (line 14), and specifying the winning price for the current active game (parameter *winPrice*). The current game will be deactivated when the owner starts a new game, and this leads to the *pickTheWinner* function (line 19) getting called by the *start* function (line 16). The *pickTheWinner* function distributes the jackpot evenly to the winners of the game, which are the participants who correctly guessed the bet price picked by the owner of the contract.

The *pickTheWinner* function sends the jackpot share to each winner by iterating through the mapping *orders* in a for-loop (line 21), which has vulnerabilities of different classes: (1) unbounded loop and (2) DoS with Failed Call. The first vulnerability is that the loop is iterating through the dynamic mapping *orders* that grows over time. When a large number of players participate in the game, the loop fails, as the cost of executing the loop exceeds the block gas limit, and none of the winners will receive the prize money. This also locks out the jackpot money in the contract forever as there are no other functions in the contract to transfer the money out. The second vulnerability is that the function *pickTheWinner* is sending money for the winners within the loop. If one of the winners refuses the money or fails to receive it by throwing an exception, the whole loop will fail, and none of the winners will receive the jackpot. The jackpot money gets locked out in the contract forever. Both vulnerabilities have serious consequences, as they destroy the promise of the game and the trust in the contract.

When we analyzed this contract with the MadMax tool [13], we found that it failed to detect either of the vulnerabilities (though both vulnerabilities are within the scope of MadMax's detection capabilities). To understand why, we looked into the source code of MadMax (available on Github). We found that to check for unbounded loops, MadMax checks if a loop iterates through an array, bounded by the array's size variable (e.g., *a.length*). In other words, MadMax checks if any of the variables used in the loop represents the array's size. Concretely, in EVM, the first element of a storage array is used to store the length of the array, and it is located at a *constant address* in the contract's storage. This constant address is used in the bytecode to calculate the addresses of the array elements using a `SHA3` hash instruction. Therefore, MadMax detects that a variable used in the loop's condition represents the array's size, if and only if the variable is used in an `SHA3` instruction. Similarly, to check for DoS with failed call, MadMax checks if the address of a call within a loop is an array element. To do so, it checks if the element is part of an array whose size is used in `SHA3` instruction. Unfortunately, these rules does not work for nested structures (e.g., multi-dimensional arrays) such as the one in this example. This is because the address of the variable used to store the length of a nested structure is a hash address that is resolved at run-time, rather than a constant address. Therefore, the pre-specified rules in MadMax are unable to reason about the structure used in the loop, and hence MadMax does not detect these vulnerabilities.

This example illustrates the challenge of using pre-specified rules to identify gas-related vulnerabilities (as done by MadMax). Even if the rules are modified to handle this particular case, there are many other variations that are difficult to express as pre-specified rules.

4 GAS-RELATED VULNERABILITIES AND DETECTION VIA TAINT ANALYSIS

eTainter uses *taint tracking* to find gas-related vulnerabilities without using predefined rules. In what follows, we discuss the taint sources in our analysis, describe each vulnerability, and then explain how taint analysis can detect it.

4.1 Taint Sources

In our analysis, the taint sources are the EVM instructions that introduce user data. These instructions read either the arguments of the invoked contract's function or the transaction-related data (e.g., sender and Ether value).

The first row of Table 1 shows the EVM instructions that are defined as taint sources. The EVM instructions `CALLDATALOAD` and `CALLDATACOPY` read data passed as arguments when calling a contract's function, `CALLER` and `ORIGIN` return the sender and the origin of the transaction, respectively, and `CALLVALUE` returns the Ether value of the transaction. In our running example, the parameter *betPrice* (line 9) of the function *buyTicket* is read by the EVM instruction `CALLDATALOAD` in the bytecode. The global built-in variable *msg.sender* (used in line 10) corresponds to the EVM instruction `CALLER`, which returns the address of the contract's caller (transaction sender) in the bytecode.

Further, in the bytecode, the EVM instruction, `SLOAD`, loads data from the contract's storage. In line 21 of our running example, the length of the array read by *orders[game].length* is loaded from the storage, and this pattern is translated to `SLOAD` instruction in the bytecode. Our analysis initially considers the data read from a storage slot as a tainted data; thus, the EVM instruction `SLOAD` is defined as a source of taints. However, because not all data in the storage is actually controlled by the user, our analysis performs further checks to validate if storage sources are controlled by the contract users as we discuss in Section 6.1.

4.2 Gas-Related Vulnerabilities

We follow the definition of gas-related vulnerabilities from prior work, MadMax [17], excluding the Induction Variable Overflow vulnerability class since the issue causing this vulnerability (casting induction variables) is statically detected by newer versions of the Solidity compiler [18]. Below, we define the remaining vulnerability classes and cast each as an instance of the taint analysis problem.

4.2.1 *Unbounded Loops.*

This class of vulnerabilities occurs when a loop iteration is determined based on user input. The most common form of this vulnerability are loops that iterate through a dynamic data structure (e.g., array, mapping) that grow over time, and can be manipulated by the contract's users [8, 35], as in line 21 of the example in Figure 1.

Another example of this vulnerability class are implicit loops generated due to Solidity's programming patterns. Such loops iterate over all items of dynamic arrays, e.g., the code pattern "*arrayName = new dataType[](0)*" used by developers to clear arrays. Such code pattern, "*creditorAddresses = new address[](0);*" from the GovernMental contract, was behind the vulnerability responsible for locking out about 1100 Ether (worth about $2.5 million) [31].

The code above compiles to a loop that iterates over the elements of the array *creditorAddresses* and sets them to zero, effectively deleting them, as shown by the following pseudocode: **foreach** $a \in$ array A **do** $a \leftarrow 0$; When the number of creditors in *creditorAddresses* is large, the gas cost of executing the loop exceeded the block gas limit. This resulted in the execution of the enclosing function being reverted and prevented it from doing its job.

Table 1: EVM instructions defined as sources and sinks by *eTainter*. NA = Not applicable (i.e., No constraints).

Type	EVM Instruction(s)	Additional Constraints
Sources	`CALLDATALOAD`, `CALLDATACOPY`, `CALLER`, `ORIGIN`, `CALLVALUE`, `SLOAD`	NA
Sink	`CALL` < gas, **addr**, value, dataOffset, dataLength>	(1) `CALL` is in a loop's body, and (2) The `CALL`'s return is the condition of a revert
Sink	`JUMPI` <destination, **condition**>	NA
Sink	`ISZERO` <**value**>	NA
Sink	`SSTORE` < key, **value**>	NA

Detection: Taint analysis can find that a loop is unbounded by defining the condition of the loop as a sink and then checking if the tainting sources (data loaded from storage slots written or manipulated by contract users) reach the defined sink. For the example in Figure 1, taint analysis checks if contract users can change the length of the array *orders[game]* used as a bound in the loop (line 21). The condition in this loop ($i < orders[game].length$) is the sink, and *msg.sender* is the source. Taint analysis finds data flows in the *buyTicket* function (line 10) in which the taint source reaches the array *orders[]*, causing an increase in the array size. Loops that iterate over user inputs (e.g., function parameters) are commonly used, and reporting all these loops would result in high false-positives. Therefore, our analysis only detects the loops that can result in a Denial of Service (DoS) due to being bound by storage data items.

In our analysis, sinks in the EVM bytecode are defined by first locating the basic blocks forming loops' exit conditions. Then the arguments of the logical instructions (e.g., `ISZERO`) along with the condition arguments of conditional jump instructions (e.g., `JUMPI`), used to direct execution of the loop, are defined as sinks. The condition in the example (*i< orders[game].length*) forms a basic block with the last instruction as `JUMPI <destination, condition>` that directs the execution to the loop body. The condition argument of the `JUMPI` instruction is defined as a sink.

4.2.2 *DoS with Failed Call.*

This vulnerability occurs when external calls are performed in the body of a loop (e.g., to pay users by sending Ether to several addresses) [36]. In smart contracts, it is not advisable to group external calls in a single transaction, i.e., within a loop, and calls should rather be isolated to their own transaction, e.g., each user should withdraw their own Ether. That is because the common practice is to check that each call succeeds, and revert the execution if not. When grouping transactions, if one of the target call recipients has an error (e.g., with a gas-expensive fallback function), the failed call throws an exception, the whole execution gets reverted, and the loop never completes (i.e., no one gets paid). The loop in Figure 1 (line 21) has this vulnerability as *transfer* (line 23) is designed to revert on failures, which an attacker can exploit if they are able to make a malicious contract (e.g., one with an expensive fallback function in terms of gas) a target of the external call executed in the loop's body.

Detection: Taint analysis can be used to detect loops having this vulnerability by defining the target address of a call executed in the loop's body as a sink if the return of the call is the condition of a revert statement in the loop's body. Then taint analysis checks if tainting sources (user-defined data loaded from storage) can reach the sink. In this example, the target address (*orders[][].player*) of the *transfer* statement at line 23 is defined as a sink for the taint analysis. Then, taint analysis finds data flows in the buyTicket function (line 10) in which the tainting source *msg.sender* is reaching the array *orders[]*, storing the target address of the transfer. In the EVM bytecode, the external calls are translated into `CALL` instructions of the form `CALL <gas,address,value,..>`. Our analysis defines as sinks the `address` arguments of the `CALL` instructions found in the loop body that are followed by revert instructions.

4.3 Vulnerability Protection Mechanisms

In some cases, developers follow specific programming practices to mitigate the risk of the discussed vulnerabilities and protect the contracts from being exploited [34]. For example, developers may mitigate the risk of an unbounded loop by splitting the loop over multiple transactions safely [35]. Another example is implementing access control to prevent exploiting a vulnerability through a public function by restricting the call of the function to only the owner of the contract. We refer to the code patterns used to implement such mechanisms as protective patterns. Our analysis does not report cases with protective patterns as vulnerabilities (see Section 6.2).

5 OUR APPROACH: ETAINTER

This section presents *eTainter*, our taint analysis approach for detecting gas-related vulnerabilities in smart contracts. At a high-level, *eTainter* takes as input the EVM bytecode of the smart contract being analyzed; it builds the Control Flow Graph (CFG) and identifies the EVM instructions that introduce user data in the bytecode (taint sources) and the EVM instructions that can form gas-related vulnerabilities (sinks). It then extracts the CFG paths leading to the defined sinks and performs taint analysis on these paths. Finally, *eTainter* searches for possible protective patterns and excludes protected vulnerabilities. It reports the found vulnerabilities to the users, along with the corresponding vulnerable functions causing the vulnerabilities. In the following subsections, we explain our approach using the running example in Figure 1.

5.1 CFG Construction

eTainter constructs a context-sensitive, inter-procedural CFG, i.e., constructed for the entire contract rather than for individual functions. The CFG is built for the EVM runtime bytecode of the contract, i.e., the code deployed on the Ethereum blockchain, and represents all functions of the contract and the interaction between them. *eTainter* performs an inter-procedural analysis, with any public/external function or fallback function in the CFG used as an entry point for the analysis (Section 2).

5.2 Extracting Vulnerable Paths

eTainter identifies paths leading to gas-related vulnerabilities as specified in Algorithm 1. The algorithm takes as input the sinks for each class of gas-related vulnerabilities (as discussed in Section 4). It uses the algorithm proposed by Krupp et al. [19] to compute a backward slice (over the created CFG) for each sink instruction (line 2). This is done to reduce the number of paths analyzed for taint flows to the defined sink. If the slice contains any of the instructions defined as taint sources, *eTainter* traverses the CFG and finds paths leading to the sink under analysis, from which all instructions of the slice can be reached, to perform taint analysis.

Then for each extracted path, starting at the root, the algorithm propagates taints through the stack, memory, and storage based on the semantics of the EVM instructions (line 4). *eTainter* uses `taintAnalysis` function (lines 22-27), which returns a tuple `<tsink,src>`, where `tsink` defines whether the sink being analyzed is tainted in the current path, and `src` is a set of the taint sources reaching the analyzed sink. When taints reaching the sink are loaded from slots in the contract's storage, the algorithm checks if the corresponding storage slots are also tainted (lines 5-16). Finally, *eTainter* checks for protective patterns in lines 17 and 20.

To check if a storage slot is tainted, *eTainter* first searches for storage write (`SSTORE`) instructions in the contract and defines them as sinks (line 7). Then *eTainter* traverses the CFG to find paths leading to the defined storage sink instructions for taint analysis (line 9), and checks for taint flow in each path (line 10) – we discuss this further in Section 6.1. If a taint flow is found, the address of the storage slot written to by `SSTORE` is added to list of tainted slots (lines 11-14), and the storage slot is confirmed as tainted. When the taints flow from another storage slot (storage-to-storage taint flow) not marked as tainted, the algorithm repeats the process to check if this new storage-slot source is also tainted (lines 15-16).

To propagate taints, for instructions defined as sources, the procedure `propagateTaint` introduces taints based on the semantics of the source instructions. Otherwise, `propagateTaint` propagates taints according to the taint propagation rules (line 25). *eTainter* uses three main taint propagation rules, summarized in Table 2:
Rule-1: For instructions that take one or more operands and derive a value, if any operand is tainted, the derived value is tainted.
Rule-2: For instructions that query the state of blockchain, such as `TIMESTAMP`, no taint propagation happens since blockchain state information (e.g., block numbers, timestamp) is not manipulated by the contract users. Similarly, no taint propagation happens for instructions that write to the blockchain (do not include `SSTORE` instruction that modify contract's storage), such as `LOGn` instructions that archive logs generated by the contract events and `CREATE` instruction that creates other contracts. The changes made by such instructions neither influence the contract execution nor change the contract state (e.g., the logs written to the blockchain are not accessible from the contract code).
Rule-3: For instructions that read their input from memory (e.g., `SHA3`), if the data read from memory is tainted, the result value is tainted. The same holds for loading data from storage using `SLOAD` instruction. *eTainter* considers taintedness of the loaded data rather than addresses. For example, when propagating taints for `V=SLOAD(`s_i`)` instruction that loads data from the storage address s_i, *eTainter* will mark `V` tainted only if the data stored at location s_i is tainted. To do so, *eTainter* propagates taints through the contract storage.

Table 2: Taint propagation rules. INS = Instruction.

Rule-1	$V = INS(x_1, ..., x_n), \exists x_i \in Tainted \implies V = tainted$
Rule-2	$BlockChain = INS \parallel INS = BlockChain \;\not\Longrightarrow$ Propagate
Rule-3	$V = INS(m_i/s_i), m_i/s_i \in Tainted \implies V = tainted$

Propagating taints through the contract storage is challenging for two main reasons. First, the data stored in the contract storage is persistent (i.e., maintained over transactions). Second, EVM uses an unconventional method to access arrays and mappings stored in the contract storage. We discuss how we address these challenges next in Section 6.1.

Algorithm 1: Extracting vulnerable paths

```
   Input: sources,sinks,cfg
   Output: {Pv,Pt} ▹ Vulnerable & tainting paths
1  begin
2      slices ⟵ BackwardSlicing(sinks,cfg)
3      foreach Path p ∈ ExtrPaths(slices,sinks,cfg) do
4          <tsink,src> ⟵ TaintAnalysis(p,sources,sink)
5          if tsink & src ∈ Storage then
6              tSlots ⟵ {} ▹ list of tainted storage slots
7              sSinks ⟵ find(sstore)
8              sSlices ⟵ BackwardSlicing(sSinks,cfg)
9              foreach path ps ∈
                ExtrPaths(sSlices,sSinks,cfg) do
10                 <tslot,src> ⟵
                    TaintAnalysis(ps,sources,sSink)
11                 if tslot & (src ∉ Storage | src ∈ tSlots)
                    then
12                     if src ∉ Storage then
                           ▹ Add address of sstore sink to the list of tainted slots
13                         tSlots ⟵ tSlots ∪ sSink.addr
14                     break
15                 else if src ∈ Storage then
16                     go to 7
17             if tslot & not IsProtected(p,ps) then
                   ▹ No protective patterns
18                 {Pv,Pt} ⟵ {Pv,Pt} ∪ {p,ps}
19         else if tsink then
20             if not IsProtected(p) then
21                 {Pv,Pt} ⟵ {Pv,Pt} ∪ {p,p}
22 Procedure TaintAnalysis(P, Sources, sink)
23     foreach instr ∈ P do
24         if instr ∈ Sources | instr ∉ sink then
25             <tMem′,tStorage′,tStack′> ⟵
                PropagateTaint(tMem,tStorage,tStack)
26         else if instr ∈ sink & sink ∈ Tainted then
27             return {true, taint sources reaching sink}
```

5.3 Implementation

We have implemented *eTainter* in an automated tool that works on the EVM bytecode of smart contracts. *eTainter* generates the bytecode by compiling the input source code using the *solc* Solidity compiler. *eTainter* uses a modified version (modified in this work) of the code used in *teEther* [19], a symbolic analysis framework for smart contracts, to generate the control flow graph (CFG) and compute backward slices. Further, *eTainter* uses *rattle* [14], a framework for recovering the Static Single Assignment (SSA) [7] form of the bytecode as the current implementation of *eTainter* uses SSA form for performing taint analysis to handle data overwrites.

6 DESIGN DECISIONS

In this section, we discuss the design decisions we made while implementing our approach to address challenges of (1) propagating taints through storage and memory, (2) checking for protective patterns, and (3) identifying loops in the EVM bytecode.

6.1 Handling Storage and Memory Taints

Unlike in traditional languages, taint analysis in smart contracts has to deal with the contract's storage, used to store persistent data, besides volatile memory, used to store transient data. In this section, we discuss the challenges encountered in propagating taints in storage and memory, and how we address them.

6.1.1 Validating Storage Taints. When the taint analysis module finds a flow of tainted data loaded from the storage to a specific sink, the sink will be flagged along with the storage slot(s) from which the data is loaded. However, considering all data loaded from the storage as tainted data will increase the number of false positive cases, as not all storage slots can be manipulated by the contract's users. For example, in the loop at line 21 of the running example, the taint analysis module finds that the value of "*orders[game].length*" (marked as tainted since it is loaded from the storage) reaches the loop condition "*i<orders[game].length*" defined as a sink. *eTainter* needs to check whether users of the contract can indeed change the value of "*orders[game].length*" to mark it as tainted, as only then can the array grow over time, and the vulnerability be exploited.

As specified in lines 5-16 of Algorithm 1, *eTainter* addresses this challenge by recursively checking for possible taint flows that could reach these storage slots. *eTainter* first marks these storage slots as sinks (*orders[game].length* in the example), and then performs taint analysis to check if tainted data is written to these storage slots. If so, then the storage slots under analysis are confirmed as taint sources. However, in some cases, *eTainter* cannot locate the SSTORE instructions writing to these specific storage slots to mark them as sinks (some storage addresses used in SSTORE instructions are resolved through constant propagation during taint analysis as we will discuss in the following section 6.1.2). *eTainter* gets around this limitation by extracting all paths leading to all SSTORE instructions in the contract's bytecode. Later, during taint analysis, the storage addresses are resolved, and *eTainter* is able to check the flow of tainted data to only the storage slots under analysis.

Handling paths leading to all SSTORE instructions would be rather expensive. However, extracting paths leading to SSTORE based on slices containing taint sources would reduce the number of the extracted paths. Further, when the taint analyzer finds a vulnerable path, it will stop the analysis of the remaining paths. We found the performance to be quite reasonable in our experiments.

6.1.2 Dealing with Hashed Addresses. Hashed addresses used to reference individual items within arrays and mappings pose two main challenges in propagating taints through the contract storage: (1) difficulty to know the parent array or mapping of an item from its hashed address; and (2) dealing with the cases when the calculation of a hash address depends on user inputs.

To explain the first challenge, in our running example, the variable "fee" is stored in the storage at address 0 and the array "orders[]" is stored starting, say, at address 2. The first slot of the array (address 2 in this example) is used to store the length of the array. However, the addresses of the array's elements are calculated using the EVM instruction SHA3, which computes the *Keccak-256* hash of the array length (stored at address 2). Then the element's index is added to the calculated hash to obtain the element's address. For instance, the array element $orders[\,][1]$ will be stored in the address calculated as $SHA3(orders[\,].length)+1$. The loop at line 21 is unbounded by the dynamic array "*orders[].length*" that grows over time in the public function *buyTicket* (line 10). When the function *buyTicket* is called, an item will be added to the array and stored in a new storage slot, for example at address 'X', and no tainted data will propagate to the storage slot storing "*orders[].length*", as this slot will be incremented by a *constant* value 1. Therefore, it is not straightforward for *eTainter* to decide that this address is part of the array "*orders[]*", and that tainting the slot at 'X' will also taint the slot storing "*orders[].length*".

To address this challenge, *eTainter* propagates the taint to the slot that stores the length of an array/mapping when one of its slots is tainted. To do this, *eTainter* keeps track of the addresses of the base slots used for calculating storage hash addresses (e.g., the base slot for the hash address calculated through "$SHA3(1) + 2$" is 1) when propagating taints. Thus, when *eTainter* finds that the storage slot at address $SHA3(1) + 2$ is being tainted, it will also mark the storage slots of the array with the base address "1" as tainted.

As for the second challenge, when propagating taints through storage arrays and mappings, not all the addresses used by storage instructions (SLOAD and SSTORE) are constants (stated in the bytecode). To handle these cases, *eTainter* resolves the address used in the SLOAD or SSTORE instruction by propagating constants through the other EVM instructions that derive the address. When *eTainter* cannot resolve the address due to a dependency on user data inputs, which makes precise modeling infeasible, it over-approximates and considers the whole array or mapping as tainted. In our analysis, we preferred to over-approximate for two reasons. First, adding elements to an array results in increasing the array's size, and most of the *unbounded loops* occur due to being bounded by the array sizes of dynamic arrays. Hence, ignoring these cases when addresses are unresolvable will result in several false negatives. Second, the "*DoS with failed call*" vulnerabilities can result from the attacker tainting a single element of the array/mapping that stores the target addresses of external calls performed within loops. Hence, we over-approximate to detect these cases when we are not able to resolve the address for an individual element of an array/mapping.

In EVM, the base addresses of arrays and mappings are always constant, and as mentioned above, *eTainter* keeps track of the arrays and mappings base addresses. This enables *eTainter* to identify the arrays or mappings to which an item belongs even when the item index/key is not a constant.

6.1.3 Handling Memory Taints. *eTainter* propagates taints through the contract's memory since few instructions use memory as input/output buffer. The challenge faced for modeling memory taints propagation statically in EVM bytecode is that not all offsets used in memory instructions are constant values. In our analysis, *eTainter* implements memory modeling that favors precision. *eTainter* resolves offsets by propagating constants through code instructions that derive memory offsets for memory-based instructions, and it handles memory locations accessed by instructions with unresolved offsets as untainted. This modeling might be incomplete, but it covers the needs of our analysis.

Previous work [30] showed that 80% of offsets in memory writing instructions (`MSTORE`) and 85% in memory reading istructions (`MLOAD`) are statically resolved. Further, most of the unresolved offsets are in instructions that use memory to call other contracts (`CALL` and `DELEGATECALL`), log events (`LOG1`), and halt (`RETURN`) or revert (`REVERT`) contract execution [20]. *eTainter* performs intra-contract analysis and is hence not affected by calling other contracts, and the other instructions either end (`RETURN` and `REVERT`) or do not influence contract execution (`LOG1`).

6.2 Checking for Protective Patterns

To enhance the precision of *eTainter* in reporting true vulnerabilities, the taint analysis excludes vulnerable paths that implement the following protective patterns.

6.2.1 Use of Access Control. One of the common practices in smart contracts is the use of function modifiers (Solidity's special construct) to restrict the call of public functions to only the contract's owner or specific addresses. A function modifier checks a condition before the execution of the function. In addition to the use of modifiers, developers could implement checks on the function's caller within the function code itself, such as "*require(msg.sender == owner)*" that halt the execution of the function, and reverts changes when the function is called by any address except the owner. *eTainter* excludes such vulnerabilities that can be exploited only by the contract's owner or authorized users.

6.2.2 Resumable Loops. To protect against DoS attacks when executing unbounded loops, developers may write loops that are split across multiple transactions. The common practice is to check the available gas in each iteration, store the last successful point of the loop in a storage variable, and resume the loop from this point in the next run. *eTainter* excludes vulnerable paths leading to loops implementing these patterns by looking for the code patterns that check available gas either in the loop header or in the loop body. Gas checks might have other uses as well; however, gas checks within loops are usually implemented to prevent DoS attacks, and hence *eTainter* does not report these loops.

6.3 Deriving Bytecode's Loops

To define sinks for the unbounded loops, *eTainter* needs to derive loops from the bytecode. However, smart contracts' bytecode often contains several benign loops generated by the compiler that iterate through dynamic arrays, which do not correspond to any iterative pattern in the source code. These loops are used for cases such as handling operations over data items of type strings and bytes (in EVM, data items of type strings and bytes are represented as dynamic arrays). Identifying these loops as vulnerable by *eTainter* would result in several false-positives. Hence, *eTainter* uses static-analysis-based filters (during the definition of the sinks) to filter out these loops. The filters rely mainly on the observation that these loops have a simple structure (exactly two basic blocks; a header and body), and have unique instruction patterns. These patterns are unlikely to overlap with user-defined loops because of the simple functionality of these loops (e.g., initialize a string data item).

For example, in the code: **function** *set(string _n) {name = _n;}* the compiler forms a loop to assign the string received as a parameter (*_n*) to the storage variable (*name*). The simplified body of this loop (does not involve stack instructions) matches the pattern (`MLOAD SSTORE ADD ADD JUMPI`), and is preceded by a code pattern that pushes the length of the string (name) on the stack. Therefore, *eTainter* filters this loop out.

7 EVALUATION

Our evaluation aims to answer three research questions (RQs):
RQ1. What is the effectiveness of *eTainter* in detecting gas-related vulnerabilities compared with MadMax?
RQ2. What is the performance of *eTainter* in terms of its analysis time and memory consumption?
RQ3. What is the prevalence of the gas-related vulnerabilities addressed by *eTainter* in real-world Ethereum contracts?

7.1 Experimental Setup

7.1.1 Datasets. To conduct our experiments, we used three datasets as shown in Table 3. The first is an annotated dataset consisting of 28 unique smart contracts, and we use it to compare *eTainter* with MadMax [17]. Eight of these contracts had been previously used to evaluate the precision of MadMax [17], which we obtained from the authors - we call these *MadMax contracts*. MadMax had originally been evaluated on 13 contracts, but we excluded five of the contracts that are vulnerable to loop overflows. This is because loop overflows due to casting are statically detected by the current versions of Solidity compilers, i.e., they result in compilation errors. The other 20 contracts are from those deployed on the Ethereum blockchain, selected randomly from among the contracts flagged by MadMax as vulnerable. We call these the *random contracts*.

We have manually inspected all the 28 contracts and annotated all the vulnerabilities that belong to the two classes: (1) unbounded loops, and (2) DoS with failed call. The inspection process is done by two researchers to avoid bias, one of whom is this paper's co-author. Each researcher separately annotated vulnerabilities, and finally, only the vulnerabilities agreed on by both the researchers are annotated in the contracts. We call this the *"Annotated dataset"*.

The second dataset is obtained by downloading a snapshot of the Ethereum blockchain, and extracting the contracts from it using Ethereum Etl [15], an open-source tool. We have extracted 856, 121 contracts from the snapshot taken on Jan 30, 2021. We removed the duplicates through running the *md5sum* checksum (as done by Duriex et al. [9]) on the contracts' binary code, we ended up with 60,612 unique contracts. We refer to this as the *"Ethereum dataset"*.

Finally, we extracted 3,000 contracts deployed on Ethereum blockchain that have the largest number of transactions. These

Table 3: Datasets used in our evaluation.

Dataset	Number of unique contracts
Annotated dataset	28
Ethereum dataset	60,612
Popular-contracts dataset	3,000

have 89,000 transactions on average, while the average transactions overall is only 129. We call this the "Popular-contracts dataset".

7.1.2 Methods and Metrics. We run all experiments on ten Intel Xeon 2.5 GHz machines, allocating one core and 48 GB of RAM for each run on each machine. In our experiments, we set a timeout value of 5 minutes per smart contract.

To answer **RQ1**, *eTainter* is compared with the MadMax tool in terms of its effectiveness in detecting gas-based vulnerabilities on the Annotated dataset. We inspect the reported vulnerabilities by MadMax and *eTainter* to determine if these reported vulnerabilities are true-positives or false-positives, based on the annotations of the vulnerabilities in the contracts. We use the inspection results to estimate the precision, recall, and the F1 score (harmonic mean of the precision and recall) for both tools. Further, to remove any bias due to the choice of contracts, we estimate the precision, recall, and the F1 score separately based on the inspection results of (1) MadMax contracts and (2) the random contracts. In our comparison, we use the current version of MadMax [13], along with MadMax's online deployment [32] as it prints analysis logs in a readable format.

To answer **RQ2**, we use *eTainter* to analyze each contract in the Ethereum dataset, and calculate the average time for the contracts that were analyzed successfully. If the analysis exceeded the timeout value, we terminate it, and consider *eTainter* as unable to analyze the contract. We also measure its memory usage.

To answer **RQ3**, we ran *eTainter* on the Ethereum dataset to determine how prevalent are gas-related vulnerabilities in real-world contracts. For each vulnerability class, we count the contracts that contain at least one instance of the class. We also run *eTainter* on the Popular-contracts dataset to determine how prevalent are gas-related vulnerabilities in widely-used real-world contracts.

We made the artifact of the paper, including the annotated dataset, publicly available at [16].

7.2 Results

RQ1 (Comparison with MadMax). Table 4 shows a summary of the comparison of *eTainter* with MadMax on the Annotated dataset. The first part shows the comparison results using the MadMax contracts, while the second part shows the comparison results using the random contracts. In each part, the column (Annotated) shows the number of vulnerabilities in the contracts for each class. The other columns show the vulnerabilities reported by both tools (#N), the true-positives (TPs), and the false-positives (FPs). The table shows the precision, recall, and F1 score for each tool.

eTainter *reported 36 of the 37 true vulnerabilities reported by MadMax, and 11 additional true vulnerabilities.* In all, *eTainter* reported *47 out of the 50 vulnerabilities* annotated in the 28 contracts (we later discuss the reason for the three false-negatives). Thus, *eTainter* has an overall **precision** of **90.4%**, a **recall** of **94%**, and an **F1 score** of **92.2%**. In comparison, MadMax had a precision of 64.9% and a recall of 74%, which leads to an F1 score of 69.2%.

Examining the results by vulnerability class, for *unbounded loops*, MadMax exhibits low precision (60.5%) and recall (70.2%). In contrast, *eTainter* exhibits high precision (87.5 %) and recall (94.6%) for this class. For *DoS with failed call*, *eTainter* exhibits perfect precision (100%) and a high recall (92.3%), while the precision and recall of MadMax for this class are 78.5% and 84.6%, respectively.

Table 4 shows the results of the tools for both MadMax contracts and the random contracts separately to study whether the choice of the random contracts biases the results. Madmax performed slightly better on the random contracts than on the MadMax contracts (F1 score of 70.2% compared to 61.5%). However, *eTainter* performed similarly on both sets (F1 score of 90.9% on the Madmax contracts and 92.3% on the random contracts). Thus, we believe that the choice of the random contracts did not introduce bias.

Note that in our comparison with MadMax on the MadMax contracts, we could not reproduce the results reported in MadMax's paper using the current version of MadMax available on GitHub (commit#6e9a6e99c6) [13]. Specifically, the number of reported false-positives by the current version is higher (MadMax's paper reported only false-positives). When we contacted MadMax's authors about this, they clarified that the main difference is that the current version of MadMax considers that DoS attacks by owners of a smart contract do not count as a vulnerability. However, we do not believe that this was the reason for the difference in MadMax's results, as we did not consider these as true bugs in our annotation of the contracts (and neither does *eTainter*).

To understand the false negative cases of Madmax for all annotated contracts, we find that MadMax did not detect five unbounded loops (reported by *eTainter*) controlled by dynamic items defined within complex structures. Further, all unbounded loops that are controlled by data values stored in memory that reference storage data items are not detected by MadMax - these resulted in three undetected cases. Finally, MadMax's inference rules do not detect DoS with failed call if the Ether transfer is performed by internal functions, called inside the loop, rather than directly within the loop body. An example is shown in Figure 2. The loop at line 7 calls the internal function *send()* at line 9, to send Ether. However, this is not detected by MadMax. In contrast, *eTainter* tracks data flow throughout the contract, and can hence detect these vulnerabilities.

```
 1 // Safely sends the ETH by the passed parameters
 2 function send(address _receiver, uint _amount) internal {
 3    if (_amount > 0 && _receiver != address(0)) {
 4        _receiver.transfer(_amount);}
 5    }
 6 function placeBets() internal {
 7    for (uint i=currentIndex; i < bets.length; i++) {
 8         Bet memory bet = bets[i];
 9         send(bet.player, payout);}
10    }
```

Figure 2: A vulnerability found by *eTainter* but not MadMax.

We also find that many false positive cases reported by MadMax were bounded loops that exist only in the bytecode. Further, several other false-positives were due to loops iterating through static arrays that do not grow over time. Figure 3 shows an example, where

Table 4: Summary of comparison results of *eTainter* with MadMax for the annotated dataset. #N= Reported vulnerabilities. TP = True Positive. FP = False Positive.

	MadMax Contracts							Random Contracts							Overall	
Vulnerability	**Annotated**	**MadMax**			***eTainter***			**Annotated**	**MadMax**			***eTainter***			**MadMax**	***eTainter***
		#N	TPs	FPs	#N	TPs	FPs		#N	TPs	FPs	#N	TPs	FPs		
Unbounded loops	4	7	3	4	5	4	1	33	36	23	13	35	31	4		
DoS with failed call	1	1	1	0	1	1	0	12	13	10	3	11	11	0		
Precision	4/8 (50.0%)				5/6 (83.3%)			33/49 (67.3%)				42/46 (91.3%)			**37/57 (64.9%)**	**47/52 (90.4%)**
Recall	4/5 (80%)				5/5 (100%)			33/45 (73.3%)				42/45 (93.3%)			**37/50 (74%)**	**47/50 (94%)**
F1 score	61.5%				90.9%			70.2%				92.3%			**69.2%**	**92.2%**

the loop at line 2 is reported by MadMax as an *unbounded loop*, as it iterates through the storage array *owners*. – MadMax considers a loop using the size of a storage array (referenced by any storage write instruction) in its exit condition as an unbounded loop. This array is populated by calling the private function *addOwner_*, which is called only within the constructor[1], and is hence not vulnerable. In contrast, by finding no dataflow leading to the storage array *owners*, *eTainter* concludes that the *owners* array does not grow over time, and hence does not report it. Thus, tracking data flow results in a more precise approach to detect gas-based. vulnerabilities.

```
1 function removeOwner(address owner) external onlyOwner {
2     for (uint i = 0; i < owners.length; i++)
3         { // some code//}
4   }
5 function addOwner_(address owner) private {
6     if (!isOwner[owner]) {
7         isOwner[owner] = true;
8         owners.push(owner);  }
9   }
```

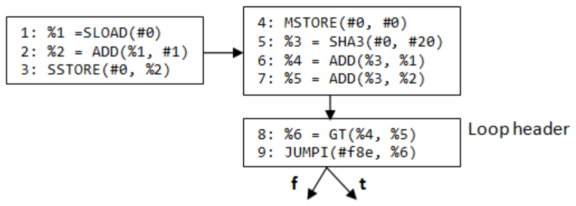
Figure 3: False positive case reported by MadMax.

As mentioned, Table 4 shows that *eTainter* has two false negative cases of unbounded loops and one negative case of DoS with failed call. By analyzing these cases we found that the loop in one case was not defined as a sink by *eTainter* as its form overlaps with a filter that *eTainter* uses to exclude benign loops used to manipulate strings (discussed in 6.3). Therefore, the vulnerability is not detected by *eTainter*. While we can potentially detect this by enhancing the sinks defining module, it may increase the false positive rate of *eTainter*. Therefore, we did not implement it. The second case was not detected due to an internal error in the code of the tool *rattle* used by *eTainter* to build the SSA form. This was also the cause behind the undetected case of DoS with failed call.

Further, *eTainter* reported five false positive cases as "Unbounded loops". After looking into the code of these contracts, we found that two cases were reported in a contract, in which an array size was used as a bound for the two loops. However, the code limits the number of elements that can be added to the array, thus preventing it from growing in an unbounded fashion. This occurs as *eTainter* does not take into account any sanity checks for sizes of the arrays used in loops, as it is not straightforward to decide when a check is valid. However, blindly excluding all such checks by *eTainter* can result in false-negatives.

[1]A contract constructor is executed only once during the contract creation, and its code is not part of the runtime bytecode deployed on the blockchain

Another false positive case is the code pattern "*uint lastIndex = airdrops.length++;*". The reason for this case is that this code pattern increases the length of the array *airdrops* by 1 and forms a loop in the bytecode similar to "*while (i < airdrops.length) {i++; some code}*", and the value of "i" is set to "*airdrops.length+1*", as shown in the simplified SSA form of the bytecode in Figure 4. Although the size of the array, "airdrops", is unbounded, the value of "i" makes the loop condition evaluate to false; it only evaluates to true in the rare case of overflow of "*airdrops.length+1*". Static analysis approaches such as *eTainter* cannot typically handle such dynamic constraints imposed at runtime, and they will hence result in false positives.



```
1: %1 =SLOAD(#0)
2: %2 = ADD(%1, #1)
3: SSTORE(#0, %2)

4: MSTORE(#0, #0)
5: %3 = SHA3(#0, #20)
6: %4 = ADD(%3, %1)
7: %5 = ADD(%3, %2)

8: %6 = GT(%4, %5)
9: JUMPI(#f8e, %6)
```

Figure 4: Another false positive case reported by *eTainter*.

The remaining two false positive cases of *eTainter* arise due to how the EVM deals with strings. As mentioned in Section 5, *eTainter* filters out loops introduced in the bytecode due to string manipulation in the Solidity source code. However, in some cases, the filtering fails, and the loops are incorrectly reported as they use the dynamic sizes of the string arrays as loop bounds.

Answer to RQ1: *eTainter* finds *higher* number of true vulnerabilities than MaxMax, and has *fewer* false-positives. Thus, it has both higher recall and precision than MadMax.

RQ2 (Performance of eTainter*).* *eTainter* timed out in 12% of the contracts in the Ethereum dataset with a timeout value of 5 minutes. We find that the timeouts in 6,210 (10.3%) out of 7,279 contracts (12%) happen during the CFG generation by *teEther* [19], and conversion to SSA form by *rattle* [14]. We experimented with larger timeout values (15 minutes and 90 minutes) for a set of 1000 contracts, but this did not substantially increase the number of the contracts analyzed successfully (the number of such contracts increased by less than 1% in both cases). We ran MadMax on the Ethereum dataset, and we find that MadMax timed out in only 1.4% of the contracts; thus MadMax scales better than *eTainter*.

For the remaining contracts that were analyzed successfully, the average time of analysis by eTainter*, including CFG generation and* *conversion to SSA form, is 8 seconds. This is comparable to the average time taken by MadMax (6s).* The minimum time taken by *eTainter* is 0.1 second, and the maximum time is 300 seconds (5 minutes). The average memory consumption of *eTainter* is 118MB (maximum of 1.77GB). *Finally, 96.97% of the contracts that* eTainter *analyzed successfully had an analysis time of less than 60 seconds.*

Answer to RQ2: *eTainter* has an average analysis time of 8 seconds, and a memory consumption of 118MB per contract.

RQ3 (Prevalence of Gas-Related Vulnerabilities).
Ethereum Dataset: *eTainter* flagged 2,763 contracts in the Ethereum dataset as having gas-related vulnerabilities, which constitutes 4.6% of the contracts. Table 5 shows the percentage of vulnerable contracts for each class in the Ethereum dataset. *The results indicate that the addressed vulnerability classes are prevalent in real-world contracts, even after excluding the fraction of estimated false-positives.* We find that 4.1% of the contracts contain unbounded loops, and 1.2% contain loops with external calls that perform a bulk transfer of Ether to several addresses. These contracts are vulnerable to DoS attacks that can result in blocking the use of the contracts forever.
Popular-Contracts Dataset: Table 5 shows the percentage of the vulnerable contracts for each class in the Popular-contracts dataset. In this dataset, *eTainter* flagged 71 contracts (2.4% of the contracts) as having gas-related vulnerabilities. The percentage of contracts vulnerable to "Unbounded loops" and "DoS with failed call" is lower in the Popular-contracts dataset (1.8% and 0.8%, respectively, compared to 4.1% and 1.2% in the Ethereum dataset). This result is along expected lines because contracts in the Popular-contracts dataset are less likely to be vulnerable than other contracts [26].

Answer to RQ3: *eTainter* finds gas-related vulnerabilities in over 4.6% of the contracts in Ethereum dataset, and in 2.4% of the most frequently used contracts. Both classes of gas-related vulnerabilities are prevalent in both sets of contracts.

8 THREATS TO VALIDITY AND LIMITATIONS

Threats to Validity. An *External* threat to validity is the limited number of smart contracts (28) used to compare *eTainter* with MadMax. This is due to the time needed to inspect contracts manually (by two researchers) and annotate the vulnerabilities in them, due to the lack of any publicly labelled dataset for the same. We have partially mitigated this threat in two ways. First, we have included all the contracts that were used in MadMax's paper [17] (barring those that no longer compile). Second, the remaining 20 contracts were selected randomly from the Ethereum blockchain.

Table 5: Percentage of vulnerable contracts reported by *eTainter* in the Ethereum and Popular-contracts datasets.

Vulnerability	Ethereum	Popular-contracts
Unbounded loops	4.1%	1.8%
DoS with failed call	1.2%	0.8%

An *Internal* threat to validity is the potential bias in annotating the vulnerabilities in the 28 smart contracts of the annotated dataset. We have mitigated this threat by having two researchers in the area performing the annotation independently, and including only the vulnerabilities agreed on by both researchers as true vulnerabilities.

Limitations. *eTainter* works on finding gas-based vulnerabilities caused due to dependency on user data. However, there might be unbounded loops that depend on data items (e.g., arrays) growing over time by adding new items of constant values into the array, which would not be detected (we did not find any such cases).

Another limitation is that *eTainter* uses other tools to generate the CFG, and lift the bytecode to the SSA form. These tools caused a timeout in many contracts (12%), and hence *eTainter* was not able to analyze these contracts.

9 CONCLUSION

This paper proposed *eTainter*, a static analysis approach for finding gas-related vulnerabilities in smart contracts, using static taint analysis on the contract's EVM bytecode. We evaluated *eTainter* on a set of 28 annotated contracts as well as on over 60, 000 unique, real-world contracts deployed on Ethereum. The results show that *eTainter* is able to find gas-related vulnerabilities with a precision of over 90%, and an average time of 8 seconds per contract. Further, the results show that *eTainter* has higher recall than the prior work, MadMax, and achieves higher precision as well. Finally, gas-related vulnerabilities are prevalent in real-world smart contracts on Ethereum, including the most frequently used contracts.

ACKNOWLEDGMENTS

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC), and the Four Year Fellowship from UBC. We thank the anonymous reviewers of ISSTA'22 for their helpful comments.

REFERENCES

[1] Imran Ashraf, Xiaoxue Ma, Bo Jiang, and WK Chan. 2020. GasFuzzer: Fuzzing Ethereum Smart Contract Binaries to Expose Gas-Oriented Exception Security Vulnerabilities. *IEEE Access* 8 (2020), 99552–99564. https://doi.org/10.1109/ACCESS.2020.2995183

[2] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469. https://doi.org/10.1145/3385412.3385990

[3] Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang. 2020. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing* (2020). https://doi.org/10.1109/TETC.2020.2979019

[4] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 442–446. https://doi.org/10.1109/SANER.2017.7884650

[5] Ting Chen, Zihao Li, Hao Zhou, Jiachi Chen, Xiapu Luo, Xiaoqi Li, and Xiaosong Zhang. 2018. Towards saving money in using smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 81–84.

[6] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

[7] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1989. An efficient method of computing static single assignment form.

In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 25–35.

[8] Solidity documentation. 2022. Gas Limit and Loops. https://docs.soliditylang.org/en/v0.5.11/security-considerations.html#gas-limit-and-loops

[9] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. 530–541. https://doi.org/10.1145/3377811.3380364

[10] Etherscan. 2021. PIPOT contract. https://etherscan.io/address/0x14d01b02d1a2aa051082810d77f8d64c80937cd5#code

[11] Asem Ghaleb and Karthik Pattabiraman. 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 415–427. https://doi.org/10.1145/3395363.3397385

[12] Github. 2015. The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity

[13] Github. 2018. MadMax. https://github.com/nevillegrech/MadMax

[14] Github. 2018. rattle. https://github.com/crytic/rattle

[15] Github. 2020. Ethereum ETL. https://github.com/blockchain-etl/ethereum-etl

[16] Github. 2022. eTainter. https://github.com/DependableSystemsLab/eTainter

[17] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27. https://doi.org/10.1145/3276486

[18] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2020. MadMax: Analyzing the out-of-gas world of smart contracts. *Commun. ACM* 63, 10 (2020), 87–95. https://doi.org/10.1145/3416262

[19] Johannes Krupp and Christian Rossow. 2018. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1317–1333.

[20] Sifis Lagouvardos, Neville Grech, Ilias Tsatiris, and Yannis Smaragdakis. 2020. Precise static modeling of Ethereum "memory". *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–26. https://doi.org/10.1145/3428258

[21] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 254–269. https://doi.org/10.1145/2976749.2978309

[22] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[23] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 653–663. https://doi.org/10.1145/3274694.3274743

[24] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[25] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[26] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 2021. Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485540

[27] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. (2018). https://doi.org/10.1145/3194113.3194115

[28] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)* 54, 7 (2021), 1–38. https://doi.org/10.1145/3464421

[29] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 664–676. https://doi.org/10.1145/3274694.3274737

[30] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82. https://doi.org/10.1145/3243734.3243780

[31] Web. 2016. GovernMental. https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck

[32] Web. 2021. contract-library. https://contract-library.com

[33] Web. 2021. Decentralized Application Security Project (or DASP) Top 10. https://dasp.co

[34] Web. 2021. Ethereum Wiki: Ethereum Contract Security Techniques and Tips. https://eth.wiki/en/howto/smart-contract-safety

[35] Web. 2022. DoS with Block Gas Limit. https://consensys.github.io/smart-contract-best-practices/known_attacks/#dos-with-block-gas-limit

[36] Web. 2022. DoS with Failed Call. https://swcregistry.io/docs/SWC-113

[37] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[38] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security assurance for smart contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 1–5. https://doi.org/10.1109/NTMS.2018.8328743

[39] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2942301