Ethainter: A Smart-Contract Security Analyzer for Compositional Vulnerabilities

Anonymous Author(s)

Abstract

Smart contracts on permissionless blockchains are exposed to inherent risks because of the interaction with untrusted entities. Static analyzers are essential for identifying security risks in smart contracts, avoiding losses of millions of USD.

We introduce Ethainter, a security analyzer checking information-flow with data sanitization in smart-contracts. Ethainter identifies combinatorial attacks that include escalation of taint, through multiple transactions, leading to serious violations. The analysis scales to the entire blockchain, consisting of hundreds of thousands of unique smart contracts (deployed over millions of accounts). Ethainter is more precise than previous approaches, as confirmed by automatic exploit generators (e.g., destroying over 800 contracts on the Ropsten network) and by manual inspection, showing a very high precision of 82.5% valid warnings for end-to-end vulnerabilities.

1 Introduction

Permissionless blockchain platforms, such as Bitcoin [27] and Ethereum [6, 40] promise to revolutionize all sectors of multi-party interaction, by enabling decentralized, resilient consensus. The Ethereum platform, in particular, allows the execution of arbitrary programs, which are called *smart contracts*. Smart contracts are registered immutably on the blockchain and operate autonomously. Therefore, software correctness in the smart contracts is most critical: (1) the contracts are high-value targets (since they manage monetary assets), (2) cannot be quickly patched, and (3) are fully available for inspection (and invocation) by potential attackers. The need for contract correctness has gained prominence via several recent high-profile incidents, resulting in losses of cryptocurrency amounts valued in the hundreds of millions [8, 39].

Smart contracts are Turing-complete programs that are typically expressed in a domain-specific language called Solidity [38]. Solidity contracts are compiled to low-level bytecode and executed by the Ethereum Virtual Machine (EVM) on the blockchain. Smart contracts store data either on the blockchain or in execution-ephemeral memory. Solidity programs make use of rather unconventional data-structure layouts using cryptographic hashes, making the static analysis of smart contracts at bytecode level a challenge. Furthermore, the EVM bytecode does not explicitly expose control-flow requiring complex decompilation techniques [5, 12] for exposing it. Hence, security analyzers for smart contracts require sophisticated techniques to overcome these problems [13, 35] checking for basic security vulnerabilities. By analyzing the bytecode directly, they eliminate the need to have their analyses second-guess the actual semantics and focus on what will be actually executed on the EVM, which is better specified [41].

With the rising awareness of Ethereum security risks and the development of recommended practices, realistic attacks have become involved. They often chain successive exploits, each enabling more vulnerabilities. A new class of security analyzers is required to identify such *composite* attacks. The most notorious (albeit relatively simple) example of a composite attack is the Parity wallet hack, where the equivalent of $280M [8] was stolen or frozen via two separate vulnerabilities. The attack involved re-initializing part of the contract that sets owner variables, via a vulnerable library function, before subsequently attacking the contract.

To safeguard against sophisticated attacks, it is essential to model accurately the information flow [31] inside smart contracts. Information-flow classifies information into trusted and untrusted and considers how it propagates in the code. For example, if untrusted ("tainted") information from external sources is allowed to propagate to critical program points, it can alter key aspects of program behavior.

Smart contract programmers employ guarding patterns to prevent an untrusted entity from executing sensitive operations, such as destroying the contract. For example, the contract code could check whether the user of the contract (a.k.a., its *caller* or *sender*) has specific privileges. The simplest check is to permit only a specific contract owner account to perform critical actions. For security analyzers, it is essential to model this guarding mechanism, for any high-fidelity information-flow analysis: if guards are not taken into consideration, many contracts will be incorrectly inferred to be vulnerable, when the only user that can "exploit" these vulnerabilities would be the owner of the contract themselves. Conversely, the guarding code can itself be attacked, by invoking code that manipulates the state of the contract in unexpected ways.

In this work, we present Ethainter: the first security analyzer for detecting composite information-flow violations in Ethereum smart contracts. Ethainter enhances the understanding of flow of tainted information with the tainting of

PL'18, January 01–03, 2018, New York, NY, USA
2018.

guard conditions, other potential ineffectiveness of guards, as well as different kinds of taint that are not prevented by guarding. Ethainter effectively attempts to model not just the existence of guards but also whether they are genuinely effective in sanitizing information. The specification of the analysis is in a formalism of independent value. It ignores orthogonal, well-understood technical complexities, capturing, instead, the essence of new information-flow concepts (e.g., tainted guards, taint through storage) in minimal form. For example, we consider a tainted memory store to induce further information flow more eagerly than an untainted (yet still statically undetermined) memory store. The precision, recall, and scalability of Ethainter are highly fine-tuned using mutually-recursive Datalog rules for tainting and overall information flow. The contribution of our work is as follows:

- We formulate an information-flow analysis for smart contracts that takes into account the sanitization (i.e., guarding) coding practices in smart contracts, as well as attacks that circumvent them.
- The precision and recall of Ethainter are highly-tuned, in a sophisticated recursive analysis written in Datalog. As a result, Ethainter exhibits very high precision (82.5%, based on manual inspection of a random sample of flagged contracts). Over largely similar vulnerabilities, the state-of-the-art tool Securify [35] (among the best known static analyzers for Ethereum smart contracts) exhibits a high false-positive rate, in comparison.
- The Ethainter implementation is highly scalable, applying to the entire set of smart contracts on the Ethereum blockchain (around 38MLoC in non-duplicate contracts) in 6 hours.
- Ethainter's companion tool, Ethainter-Kill, can automatically exploit (on live Ethereum networks) one of the security violations flagged by Ethainter. Ethainter-Kill has destroyed over 800 smart contracts on a fork of the Ropsten Network (16.67% of the flagged contracts) confirming Ethainter's practical relevance and a high lower bound for Ethainter's end-to-end precision.

2 Illustration

Composite vulnerabilities escalate a weakness through multiple transactions, resulting in increasingly more tainted elements of a contract. Consider an abstracted example:

```
contract Victim {
 mapping(address => bool) admins;
 mapping(address => bool) users;
 address owner;

 modifier onlyAdmins() {
   require(admins[msg.sender]);
   _;
 }
 modifier onlyUsers() {
   require(users[msg.sender]);
   _;
 }

 function registerSelf() public
  { this.users[msg.sender] = true; }

 function referUser(address user) public onlyUsers
  { this.users[user] = true; }

 function referAdmin(address adm) public onlyUsers
  { this.admins[adm] = true; }

 function changeOwner(address o) public onlyAdmins
  { this.owner = o; }

 function kill() public onlyAdmins
  { selfdestruct(self.owner); }
}
```

At first glance, the `Victim` contract is seemingly unsusceptible to the "accessible `selfdestruct`" vulnerability, as function `kill()` can only be called by administrators and, in any case, when called it only sends the contract's balance to the `owner`. Furthermore, all functions are guarded. Some are only accessible to registered users, some to admins and some to owners. However, the contract exhibits an error in the modifier of method `referAdmin`. This functionality, which adds administrators, should be protected by modifier `onlyAdmins` rather than `onlyUsers`—a mistake perhaps caused by the developer copying and pasting the `referUser` definition. As a consequence, an attacker may exploit the contract, in four successive steps, with each step invoking a separate internal transaction. This is encoded in the following attack:

```
contract Attacker {
 Victim victim = ...;

 function attack() public {
  victim.registerSelf();   // make myself a user
  victim.referAdmin(this); // make myself an admin
  victim.changeOwner(this);// make myself an owner
  victim.kill();           // destroy contract
 }
}
```

The attack starts by registering the attacker as a user, after which the attacker is (erroneously) allowed to add themselves to the list of admins. This enables them to subsequently set themselves as owner. Consequently, the attacker can then destroy the smart contract, draining its funds.

Notably, there are two primitive vulnerabilities in this contract: "accessible `selfdestruct`" after the second step of the attack (which would be harmless on its own) and "tainted `selfdestruct`" after the fourth step. This is precisely the kind of attack that our analysis can detect: the tainting of a guard condition successively enables more tainting, which invalidates guards that might locally appear sound.

3 Background: Kinds of Information-Flow Vulnerabilities

We next present examples of (the building blocks of) information-flow vulnerabilities in smart contracts, as well as the usual programming patterns that prevent them. We refer to these vulnerabilities by name in the rest of the paper. The presentation is intended to be simplified, showing the simplest form of every vulnerability, rather than combinations or occurrences in the wild.

3.1 Tainted Owner Variable

Smart contract programming languages, such as Solidity, typically offer `require`/`assert` features that guard a smart contract's execution. These guards are often a mechanism employed by the smart contract developer to protect against the execution of sensitive operations by untrusted parties.[1] Precise modeling of these guards is important for reducing the number of false positives by any analysis. Guards can, however, be attacked by third parties, for instance, by misplaced constructor functions in libraries, as has happened in the notorious Parity wallet hack.

Numerous further attacks may happen when guards are successfully attacked. These attacks may result in economic consequences quickly. For instance, an attacker may want to artificially dilute or inflate the value of ERC20 tokens, functionality that would undoubtedly be guarded. We consider *Tainted Owner Variable* a vulnerability in its own right, however, this vulnerability may introduce other information-flow vulnerabilities, such as making contract self-destruction accessible.

The following example shows a contract with a global variable `owner`, used in function `kill` to check whether the address of the caller of the contract (denoted `msg.sender`) is the same as the address of the owner of the contract. However, in this contract, a public setter function called `initOwner` makes it possible for anyone to update the `owner` to an arbitrary value, making the contract vulnerable.

```
address owner = ...;

function initOwner(address _owner) public {
    owner = _owner;
}

function kill() public {
    if (msg.sender == owner)
    { /* sensitive operations */ ... }
}
```

[1]Strictly speaking, only a `require` guard is appropriate for eliminating undesired input, but an `assert` is equally valuable for preventing an unexpected violation of contract invariants from propagating further.

3.2 Tainted delegatecall

The EVM's `delegatecall` instruction allows any code that is called to make any state changes to the caller's contract, including sending funds to any address or destroying the contract. The *tainted* `delegatecall` vulnerability allows an attacker to cause the contract to call another, resulting in a high-risk security vulnerability. The following Solidity code shows a naïve function called `migrate` that allows any user to supply the address used by the `delegatecall`. Such functions could well exist as private members of a contract, for reasons of code reuse, but a vulnerability arises, e.g., when the function is publicly accessible or called from a publicly accessible function without guarding.

```
function migrate(address delegate) public {
    delegate.delegatecall.gas(msg.gas - 1500)();
}
```

3.3 Accessible selfdestruct

When the EVM's `selfdestruct` instruction is invoked, the contract is deactivated and any remaining balance is sent to the address supplied. The following code demonstrates the pattern. Naturally, `selfdestruct` is a very sensitive operation that results in the permanent termination of a contract, so it should be guarded.

```
function kill() public {
    selfdestruct(msg.sender);
}
```

This vulnerability is more commonly encountered in combination with a tainted guard.

3.4 Tainted selfdestruct

This is a variant of the previous vulnerability, with potential direct profit for an attacker. With a *tainted* `selfdestruct` vulnerability, the receiving address of the funds can be controlled by any user of the contract. Notice that, even though `selfdestruct` is not accessible by third-party users, any user can taint the address (held by `administrator`), so that, when `selfdestruct` is finally called, any remaining balance will be transferred to the attacker.

```
address owner = ...;
address administrator = ...;

function initAdmin(address admin) public {
    administrator = admin;
}

function kill() public {
    if (msg.sender == owner) {
        selfdestruct(administrator);
    }
}
```

3.5 Unchecked Tainted `staticcall`

A recently-introduced EVM opcode is `staticcall`. This can be used to call a smart contract, while disallowing any modifications to the state during the call. The `staticcall` opcode enables more secure smart contract development by allowing purely functional calls, thus disallowing state-modifying reentrancy.

A problematic `staticcall` is one where the return values do not overwrite the current memory buffer used for passing and returning data to `CALL` instructions. A recent example of this vulnerability was found in the 0x [1] decentralized exchange protocol. This vulnerability enabled a single wallet to sign off transactions that should have been signed off by multiple parties. An example of such suspect code (in Solidity assembly) is shown below. The output of the `staticcall` invocation is meant to overwrite its input. However, `staticcall` does not perform this unless the client that is called returns at least 32 bytes of data. Otherwise, the input is read as output. As a consequence, a tainted `staticcall` potentially allows the attacker to pass untrusted input as output from a trusted external call, potentially fooling the caller.

```
let cdStart := add(calldata, 32)
let success := staticcall(
  walletAddress,     // address of Wallet contract
  cdStart,           // pointer to start of input
  mload(calldata),   // length of input
  cdStart,           // write output over input
  32                 // output size is 32 bytes
)

switch success
...
case 1 {
    // Signature is valid if call did not revert
       and returned true
    isValid := mload(cdStart)
}
```

In order to defend against such a vulnerability, the latest Solidity compilers insert instructions to first check that the address that is being called has valid code. In addition, when less than the minimum expected amount of data is returned by the callee, the `RETURNDATASIZE` opcode is used to pad the output accordingly. As this issue was fixed less than a year ago, many unsafe libraries still exist which do not properly handle these cases.

4 Information-Flow Analysis Framework

In this section, we present a distilled, formalized version of the essence of our analysis on a minimal input language. The key tenet is to abstract away from the complexity of language elements whose treatment has engineering complications, while emphasizing in pure form the exact elements that are of particular interest for the detection of compositional vulnerabilities. For this purpose, we design a small abstract input language that captures information-flow semantics of smart contracts sufficiently: it has taint sources and sinks, transfer through numeric operations, loads and stores to persistent storage, and input sanitization through guards. We provide information-flow rules which we will mark as either an over- or an under-approximation. The emphasis of the formalism is to capture sufficient information-flow for the detection of compositional vulnerabilities. We consider:

- programmatic guarding/sanitizing so that tainted input cannot be propagated,
- only caller input can be sanitized via guards, whereas, if taint propagates to the persistent storage of the contract, it can elude guards,
- the problem of tainting either arbitrary storage locations or the condition of a guard,
- the complex form of effective guards, possibly using the contract caller's address as a key into persistent data structures, to look up permissions, ownership, balances, etc.

4.1 Input and Output Schema

The syntax for the instructions of the abstract input language is shown in Figure 1. We use lower-case letters $(f, t, x, y, \ldots)$ to designate program variables. `sender` is a reserved variable name, designating the caller of a contract. The input program is in three-address (i.e., all expressions are expanded, up to variables) static single-assignment (SSA) form (i.e., a variable name implicitly encodes a single program location—the variable's definition). SSA phi-instructions, for merging different versions of a variable, are mere `OP`s in our language. Equality is also an `OP` for most analysis purposes, however we sometimes need to refer to equality comparisons explicitly, in which case we write them in infix form, i.e., "x := $(y = z)$", is just an instance of an "x := $\texttt{OP}(y, z)$" with the operator being equality.

The `GUARD` instruction deserves some attention: "x := $\texttt{GUARD}(p, y)$" means that x receives a sanitized value from y if a predicate with truth value p is satisfied. This captures in terms of values (instead of guarded code statements) the common pattern of reading possibly-tainted inputs only when the `msg.sender` value (i.e., the contract's caller) satisfies some criterion: either it is equal to a known storage value, or it is looked up in a data structure of allowed callers. A single syntactic block guarded by, e.g., "`if (msg.sender == owner)`" is equivalent to multiple `GUARD` statements, sanitizing every caller-input value that may flow into this block, with a predicate such as "p := $(\texttt{sender} = z)$" (where z is the result of an `SLOAD` from the persistent storage location holding the contract's owner).

Figure 2 illustrates the relations computed by our information flow analysis. Note that there are two different kinds of taint values for variables, denoted by $\downarrow^I x$ (*InputTaintedVar*) and $\downarrow^T x$ (*StorageTaintedVar*). Taint from unsanitized user

Instruction	:=	x := `OP`(y, z)	*operation, including arithmetic, boolean ops, equality, phi*
	|	x := `INPUT`()	*load input data: a taint source instruction*
	|	x := `SHA3`(y)	*SHA3 hash*
	|	x := `GUARD`(p, y)	*value guarded by sender predicate: contract caller checked in predicate p to sanitize y*
	|	`SSTORE`(f, t)	*write to persistent storage: from local variable f to storage address t*
	|	`SLOAD`(f, t)	*load from persistent storage: from storage address f to local variable t*
	|	`SINK`(x)	*sensitive instruction: a taint sink*

Figure 1. Syntax of the source language instructions.

Relation	**Notation**	**Description**
InputTaintedVar	$\downarrow^I x$	Variable x is tainted from input.
StorageTaintedVar	$\downarrow^T x$	Variable x is tainted from storage.
TaintedStorage	$\downarrow^T S(v)$	Storage location with constant address v is tainted.
NonSanitizingGuard	$\not\uparrow p$	Guard predicate p fails to sanitize.
ConstValue	$C(x) = v$	Variable x is inferred to have constant value v .
StorageAliasVar	$x \sim S(v)$	Variable x is inferred to be an alias for storage slot v .
SenderDataStructElem	$\mathrm{DS}(x)$	x holds (aliases) a data structure element, whose key is the message sender's address.
SenderDataStructAddr	$\mathrm{DSA}(x)$	x is the address of a data structure element, whose key is the message sender's address.

Figure 2. Relations computed by the information flow analysis. The first two are output relations, computed simultaneously (in mutual recursion) with the next two, while the last four are auxiliary relations. The auxiliary relations are computed before the output relations (i.e., do not depend on taint propagation, therefore their contents are fixed in a previous stratum in the fixpoint computation).

input can be eliminated by the use of sender guards, which ensure that the caller of a contract is approved. However, taint that has propagated into storage cannot be removed by the use of sender guards. The relation $\not\uparrow p$ (*NonSanitizingGuard*) captures conditions under which a guard fails to successfully sanitize caller input: either the guard condition is itself tainted, or the guard does not compare `sender` (either directly or by indexing into storage data structures).

Relations $C(x) = v$ (*ConstValue*) and $x \sim S(v)$ (*StorageAliasVar*) encode standard value-flow and alias analyses. These are the only relations whose definitions we elide, since they are entirely conventional and orthogonal to the information-flow analysis. In fact, these can well be considered input relations, computed using extra information that is not present in the abstracted input language (e.g., function boundaries). Note that $x \sim S(v)$ applies only to storage locations indexed by a statically-known constant address. We complement this relation with a less strict, but more specialized, concept, of all (unknown) addresses/data reachable through storage data structures, based on `sender`, in relations $\mathrm{DSA}(x)$ and $\mathrm{DS}(x)$.

4.2 Information-Flow Core Analysis

Figure 3 lists the rules of the information-flow analysis. We employ some conventions for syntactic convenience: Wildcard symbol $*$ indicates a "don't care" value, for a variable or a taint flavor (i.e., "I" or "T", for input vs. storage taint). The notation $\downarrow^{I/T}$ also denotes either kind of taint, but all occurrences in the same rule are required to have matching taint kinds. The discuss the information-flow rules in the following:

- LoadInput, Operation-1, Operation-2: These rules perform taint introduction and propagation through binary operators. (Note that, per our $\downarrow^{I/T}$ convention, the taint flavor that propagates is the same as the incoming one.) The only nuance is that taint from `INPUT` statements is explicitly designated as being of a weaker flavor, input taint.
- Guard-1, Guard-2: The first rule states that storage taint (but not input taint) propagates through `GUARD` operations. Conversely, input taint propagates through `GUARD` operations only if the guard predicate is non-sanitizing.
- StorageWrite-1, StorageWrite-2: The first of these rules turns any kind of taint from a tainted source into

$$(\textsc{LoadInput})\ \frac{x\ \texttt{:=}\ \texttt{INPUT}()}{\downarrow^{I} x}$$

$$(\textsc{Operation-1})\ \frac{x\ \texttt{:=}\ \texttt{OP}(y,*) \quad \downarrow^{I/T} y}{\downarrow^{I/T} x} \qquad (\textsc{Operation-2})\ \frac{x\ \texttt{:=}\ \texttt{OP}(*,y) \quad \downarrow^{I/T} y}{\downarrow^{I/T} x}$$

$$(\textsc{Guard-1})\ \frac{x\ \texttt{:=}\ \texttt{GUARD}(*,y) \quad \downarrow^{T} y}{\downarrow^{T} x} \qquad (\textsc{Guard-2})\ \frac{x\ \texttt{:=}\ \texttt{GUARD}(p,y) \quad \downarrow^{I} y \quad ⇞ p}{\downarrow^{I} x}$$

$$(\textsc{StorageWrite-1})\ \frac{\texttt{SSTORE}(f,t) \quad \downarrow^{*} f \quad C(t)=v}{\downarrow^{T} S(v)} \qquad (\textsc{StorageWrite-2})\ \frac{\texttt{SSTORE}(f,t) \quad \downarrow^{*} f \quad \downarrow^{*} t}{\forall i : \downarrow^{T} S(i)}$$

$$(\textsc{StorageLoad})\ \frac{\texttt{SLOAD}(f,t) \quad \downarrow^{T} S(v) \quad C(f)=v}{\downarrow^{T} t} \qquad (\textsc{Violation})\ \frac{\texttt{SINK}(x) \quad \downarrow^{*} x}{\textsc{violation}}$$

$$(\textsc{Uguard-T})\ \frac{p\ \texttt{:=}\ (\texttt{sender}\ \texttt{=}\ z) \quad z \sim S(v) \quad \downarrow^{T} S(v)}{⇞ p} \qquad (\textsc{Uguard-NDS})\ \frac{p\ \texttt{:=}\ (y\ \texttt{=}\ z) \quad \neg\mathrm{DS}(y) \quad \neg\mathrm{DS}(z)}{⇞ p}$$

Figure 3. Inference rules for information-flow analysis.

$$(\textsc{DS-SenderKey})\ \frac{}{\mathrm{DS}(\texttt{sender})}$$

$$(\textsc{DS-Lookup})\ \frac{x\ \texttt{:=}\ \texttt{SHA3}(y) \quad \mathrm{DS}(y)}{\mathrm{DSA}(x)} \qquad (\textsc{DSA-Lookup})\ \frac{x\ \texttt{:=}\ \texttt{SHA3}(y) \quad \mathrm{DSA}(y)}{\mathrm{DSA}(x)}$$

$$(\textsc{DS-AddrOp-1})\ \frac{\mathrm{DSA}(y) \quad x\ \texttt{:=}\ \texttt{OP}(y,*)}{\mathrm{DSA}(x)} \qquad (\textsc{DS-AddrOp-2})\ \frac{\mathrm{DSA}(y) \quad x\ \texttt{:=}\ \texttt{OP}(*,y)}{\mathrm{DSA}(x)}$$

$$(\textsc{DSA-Load})\ \frac{\mathrm{DSA}(x) \quad \texttt{SLOAD}(x,y)}{\mathrm{DS}(y)}$$

Figure 4. Inference rules for identifying data structures in storage.

storage taint, when written into a statically-known storage location. The second rule taints *all* such statically-known storage locations when the destination of the store operation is also tainted, capturing the high risk of tainted writes to addresses that may be influenced by an attacker.

- Violation, StorageLoad: These rules are mostly standard. Loads from tainted constant storage locations introduce storage taint into a local variable. Either kind of taint on an operand of a `SINK` instruction results in a reported violation.
- Uguard-T, Uguard-NDS: The *NonSanitizingGuard* relation is established under two conditions: Either the guard predicate directly compares the contract caller to a tainted storage address, or the guard predicate does not involve (neither in its left- nor in its right-hand-side) values possibly based on `sender`, via direct reference or data-structure lookup. The full definition of the latter concept is discussed in Section 4.3.

The analysis definition of Figure 3 makes the four relations $\downarrow^{I} x, \downarrow^{T} x, \downarrow^{T} S(v)$, and $⇞ p$ mutually recursive. All of them grow monotonically, with each inference for one relation only able to lead to a growing set of inferences for others. Therefore, the rules can be directly translated to an algorithm for the analysis: the rules iterate starting from empty and up to fixpoint for all relations. Conversely, relation DS(x) is negated in rule Uguard-DS, therefore, its growing leads to fewer inferences for other relations (starting from $⇞ p$). The evaluation of DS(x) is independent of taint propagation, however, and can complete before the main analysis, as described next.

4.3 Sender-Keyed Data Structure Lookups

Figure 4 shows the definition of relations DS(x) and DSA(x), which capture the notion of predicates that scrutinize the caller of a contract (by direct comparisons or in a persistent data structure). Per rule DS-SenderKey, the `sender` variable is a base case of information that pertains to the identity of a contract's caller. The only other way to produce a y that satisfies DS(y) is via rule DSA-Load: by dereferencing an address x that satisfies DSA(x). Therefore, most of the relevant rules concern relation DSA, i.e., the definition of *addresses* that may hold data pertaining to a contract's caller.

The rules conservatively capture the unconventional Ethereum memory layout and data structure policy. The keccak256 hash function (`SHA3`) is considered collision-free and used for a variety of mapping purposes, including data structure nesting. For instance, a 2-dimensional array `arr[][]` is typically identified merely by a constant storage location (e.g., `42`) that stores the length of the outer array (and not an address/pointer to its contents, as might be expected). Addresses are then "invented" by `SHA3` hashing: the contents of the array are found starting at location `SHA3(SHA3(42))` for subarray `arr[0]`, location `SHA3(SHA3(42)+1)` for subarray `arr[1]`, etc. (Accordingly, location `SHA3(42)` stores the length of subarray `arr[0]` and location `SHA3(42)+1` stores the length of subarray `arr[1]`.) Rule DS-Lookup encodes the hashing of caller-related information (DS(y)), to be used as a storage address. Rule DSA-Lookup captures nested data structures, i.e., hashes of addresses, as in our previous example, also aided by arithmetic on addresses, as shown in rules DS-AddrOp-1 and DS-AddrOp-2.

4.4 Analysis Design Discussion

We designed our information-flow analysis keeping the *completeness* and *precision* of Ethainter's implementation in mind. The tradeoff between completeness and precision heavily impacts the error detection of a security analyzer. In our design, we made conscious decisions when to *over-approximate*, which may introduce a higher false positive rate), and when to *under-approximate*, obtaining high-confidence inferences of violations for improving precision at the cost of completeness. We summarize our design decisions below:

- Relations DS() and DSA() have an over-approximate definition: the rules of Figure 4 capture any dereference, arithmetic, etc., expression that may involve the key of the contract's caller (i.e., the value of the `sender` variable). However, relation DS() is *negated* in the main information-flow analysis (Figure 3), therefore the analysis pertaining to caller lookups is an under-approximation: it favors precision, avoiding warnings whenever the analysis establishes that a guard expression *might* be scrutinizing the contract's caller. Instead, warnings are emitted only in high-confidence scenarios.
- As discussed earlier, auxiliary relations $C(x) = v$ (*ConstValue*) and $x \sim S(v)$ (*StorageAliasVar*) encode standard static value-flow and alias analyses. As conventional in static reasoning, the definition of such analyses is an over-approximation with respect to control flow: whenever two program paths meet, the union of inferences is propagated. However, the formulation of the relations themselves is an under-approximation: they are defined only for constant values (numeric values or storage addresses). This means that aliasing with a statically-unknown storage location is not captured. Aliasing with a constant storage location through memory operations that involve run-time variable addresses is also not captured. This design decision means that the analysis favors precision: it establishes taint violations and tainted guards with high-confidence only.
- The one analysis rule that over-approximates, is StorageWrite-2. If a store operation has both its value and its address tainted, then all constant storage locations that arise in the analysis are considered tainted. This design choice focuses on completeness of the analysis, i.e., it increases its ability to capture errors, but possibly yields false positives.

5 Datalog Implementation

Ethainter implements the information flow model as a set of several hundred declarative rules in the Datalog language.

Ethainter's input is a set of logic relations containing a functional 3-address code representation of an EVM bytecode program. We obtain this representation by decompiling each contract using the Gigahorse toolchain [12]. The Ethainter implementation back-end is the high-performance Soufflé [20] Datalog engine, which translates Datalog input relations into highly-optimized C++ code.

Our information-flow analysis implementation is enriched with additional smart contract analyses to obtain further precision:

- The implementation uses intra-procedural constant propagation data-flow analysis for propagating constant values (i.e., relation $C(x) = v$ of Section 4) and a call site sensitive analysis for computing storage locations that are aliases (i.e., relation $x \sim S(v)$). We have experimented with other options for more completeness or precision and believe that the above offer a good balance, hard to noticeably improve upon.
- In addition to tainting (i.e., data-dependencies), our analysis also considers some control-dependence patterns as information-flow violations. The treatment is under-approximate by design: we only capture limited cases that have a high likelihood to be true vulnerabilities.

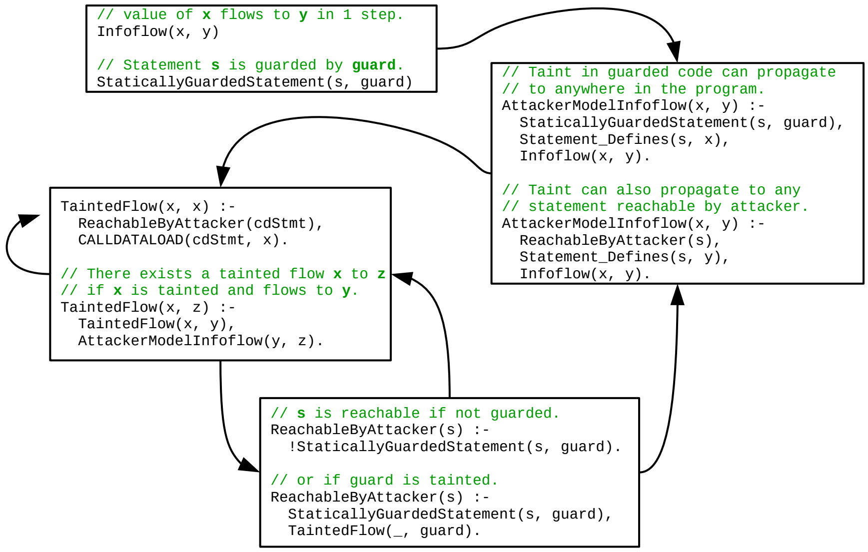


Figure 5. Skeleton of the core of the implementation in Datalog. Arrows depict complex recursion between the rules. `Infoflow` accounts for all flows, including inter-procedural flow and flows through data structures.

- In addition to the modeling of storage (i.e., persistent contract data on the blockchain), the analysis models "memory", i.e., data that pertain to a single transaction. Memory can still be inter-procedural, since the control flow of a transaction crosses different functions. Because memory is non-local (e.g., can be tainted in a function call in order to alter the execution several function calls later) taint flow through memory is often present in realistic vulnerabilities. The Ethainter modeling of memory mostly follows the principles used for storage in Section 4 but memory taint does get sanitized via guards, much like input taint.
- The implementation contains more complexity in dealing with patterns not captured in the input language of the model (e.g., tainted `delegatecall` and `selfdestruct`). Generally, this complexity is of the flavor expected when transiting from a highly simplified input language to realistic programs—the spirit of the analysis logic is well-captured by the simplified model.

The simplified core of the Datalog implementation is depicted in Figure 5, illustrating the mutual recursion between tainting, attacker-reachable code, and conditional information flow. In more detail: `ReachableByAttacker` models which parts of the code are reachable by the attacker. The rules also model (via `AttackerModelInfoflow` and `TaintedFlow`) how information can flow in a model where the attacker can only execute certain parts of the program while a priviliged user is not expected to introduce new tainted information to the state. `AttackerModelInfoflow` is a subset of the general taint analysis `InfoFlow`, a single-step information flow relation which does not distinguish between attackers and priviliged users. As the attacker can reach more parts of the program by tainting guards, `AttackerModelInfoflow` and `ReachableByAttacker` need to be mutually recursive (via the intermediate `TaintedFlow`) in order to model composite violations soundly. `InflowFlow` abstracts away most of the complexity of handling a full language, including inter-procedural flow and data structures. Most of the vulnerability patterns are then expressed in terms of `TaintedFlow`, which is a transitive closure on `AttackerModelInfoflow`.

6 Evaluation

This section presents the results of our evaluation of the Ethainter analysis. We ran the analysis using 45 concurrent analysis processes on an idle machine with two Intel Xeon E5-2687W v4 3.00GHz CPUs (each with 12 cores x 2 hardware threads, for a total of 48 hardware threads) and 512GB of RAM. We use a combined cutoff of 120 seconds for decompilation and the information-flow analysis step. Programs that did not finish analyzing within these cutoffs are considered to have timed out. (This is not too-sensitive a parameter: we decompile/analyze around 98% of available contracts. Doubling the cutoff time does not substantially increase this percentage.)

Our experiments aim to answer the following research questions:

RQ1. Is Ethainter an effective static analysis?

RQ1.A. Is the analysis relevant? What percentage of contracts are vulnerable?

RQ1.B. Is the overall analysis precise, i.e., does it have a low false-positive rate?

RQ2. Is our information-flow analysis efficient?

RQ3. Are the individual analysis components and design decisions justified?

We performed two main experiments:

1. For the *simplest* vulnerabilities reported by Ethainter ("accessible `selfdestruct`" and "tainted `selfdestruct`") we produce a completely automated exploit generator. We apply this to the most recent contracts on the Ropsten testnet to show that a large number of the flagged ones are indeed vulnerable. This is a crude experiment, without much accuracy but with great effectiveness at establishing a lower bound: the experiment shows that Ethainter is relevant in the real world, in an analysis domain where translating warnings to end-to-end vulnerabilities is very hard [29].
2. For *all* Ethainter-reported vulnerabilities, we collect statistics over a snapshot of the entire Ethereum mainnet (141.1K unique contracts), and then scrutinize further a small random sample of contracts, via manual inspection. We report results for Ethainter and compare to probably the best-known state-of-the-art tool, Securify [35].

6.1 Experiment 1: Automated End-to-End Exploits

In many cases, Ethainter pinpoints vulnerabilities with enough precision to actually exploit them end-to-end. To demonstrate that it can do so with reasonable effectiveness, showcasing the execution of these vulnerabilites in the process, we have built a simple prototype tool called Ethainter-Kill that exploits vulnerabilities on real programs. Ethainter-Kill is fully automated—it reads Ethainter's output, connects to Ethereum nodes and proceeds to exploit a subset of vulnerabilities that are flagged by Ethainter. Currently, Ethainter-Kill supports only two vulnerabilities, which are "accessible `selfdestruct`" and, to a lesser extent, "tainted `selfdestruct`".

We deployed Ethainter-Kill on a private fork of the Ropsten Ethereum testnet.[2]

In this experiment, Ethainter was used to find vulnerabilities from smart contracts deployed in the most recent 1.2 million blocks. This yielded 4800 flagged contracts out of 882000 (0.54%). Out of these, Ethainter could pinpoint the vulnerability in 3003 contracts. (For the rest, Ethainter-Kill was unable to find a public entry point that would reach the private, Ethainter-flagged vulnerable statement.) Ethainter-Kill was called on these contracts, where it proceeded to connect to the relevant Ethereum node and call the API of the contracts which trigger a `selfdestruct` with some generated parameters. As expected, many calls resulted in an error, mostly due to the limitations of Ethainter-Kill. (Automated exploit generation is a challenging area of research, and actual exploits often require significant human ingenuity.) Ethainter-Kill also verified whether the transactions resulted in the contract actually being destroyed by analyzing the exact VM instruction trace and identifying whether the `selfdestruct` opcode was executed. In total **805** contracts (**16.7%** of warnings) were successfully destroyed in this experiment. This rate represents a lower bound of the true positive rate. However, even such a lower bound is amply sufficient at demonstrating the effectiveness of Ethainter in a practical setting (i.e., to partly answer RQ1): identifying several hundred practical *end-to-end* exploits automatically is significant. For comparison, Pérez and Livshits [29] recently report that, of the vulnerabilities reported (as "true positives") in Ethereum program analysis publications, only 0.3% (in terms of Ether value) have been exploited in the wild, strongly suggesting that most of the rest are not truly end-to-end exploitable.

[2]We considered applying Ethainter-Kill directly on the public testnet. However, we were explicitly discouraged from doing so in conversations on Ethereum security forums [37]. The argument is that, although the testnets are intended for testing, many smart contracts on public test networks closely resemble smart contracts currently in production. Thus, any transactions we would perform on these networks could disclose the exploits publicly.

6.2 Experiment 2: Statistics and Manual Inspection

We evaluate the effectiveness of Ethainter by examining the percentage of contracts flagged for each vulnerability. We also select a subset of programs for manual inspection, and use this to estimate a true-positive rate for our analysis.

The percentage of flagged unique contract code bases (among 141K) for each vulnerability are shown below:

Vulnerability	Percentage
accessible `selfdestruct`	**1.2%**
tainted `selfdestruct`	**0.17%**
tainted owner variable	**1.33%**
unchecked tainted `staticcall`	**0.04%**
tainted `delegatecall`	**0.17%**

Notice that the unchecked tainted `staticcall` vulnerability is relatively rare. However, this is to be expected as it only applies to recently deployed smart contracts that utilize the new `staticcall` opcode. The number of contracts that utilize this opcode is expected to increase.

To estimate the precision (i.e., true-positive rate) of Ethainter, we randomly selected 40 contracts among those flagged by Ethainter that have verified sources on Etherescan.io (for ease of manual inspection). Contracts were repeatedly sampled randomly, after performing a lexicographical sort on their addresses (i.e., 20 byte hashes), until a random sample of 40 contracts had at least one flagged in every vulnerability category.

The results from our systematic manual inspection of the Ethainter sample are shown in Figure 6, and yield an estimated precision of 82.5%.

In order to preserve the identity of contracts until we have a responsible disclosure agreement in place, we will refer

Contracts	Remark
Accessible `selfdestruct`. True positives: 10/10	
`6fd`✓, `9d0`✓	Poor design
`14f`✓, `342`✓, `4ea`✓	Programming error
`292`☆✓, `4fa`☆✓	Ownership guard can be tainted
`a33`	Race condition
`adf`☆✓, `d4f`☆✓	Ownership guard can be "bought"
Tainted `selfdestruct`. True positives: 6/6	
`292`☆✓, `4fa`☆✓, `d4f`☆✓	Tainted via ownership
`adf`✓	Programming error
`6fd`✓, `9d0`✓	Poor design
Tainted Owner Variable. True positives: 15/21	
`169`✓	Token supply can be manipulated
`135`✓, `26a`✓, `292`✓	-
`2cc`✓, `3a2`✓, `adf`✓	-
`4d9`✓	Conflated ownership of token&contract
`600`✓, `6c3`✓, `96b`✓	Public initializer (race condition)
`6fd`✓, `9d0`✓	Poor design
`aa4`✓, `d4f`✓	Via payment
`fc5`✗`124`✗	Complex path condition
`373`✗, `fc3`✗	Not an owner variable
`577`✗	Bug in inter-function flow
`c5c`✗	Imprecise data structure inference
Tainted `delegatecall`. True positives: 1/1	
`ef0`✓	Via complex flow through array
Unchecked Tainted `staticcall`. True positives: 1/2	
`8c0`✓	Missing redurn data size check
`152`✗	Complex memory conditions
Total Precision: 82.5% (33/40)	

Figure 6. Summary of results of manually inspected warnings for Ethainter. Contract ids are a 12-bit substring of their true address. Ids marked with ✗are false positives. The ones marked with ✓are true positives. Moreover, ones marked with ☆can only be exploited via a composite tainting vulnerability.

to contracts by three hexadecimal digits contained in their identifier.

In addition to the randomly sampled contracts, Ethainter correctly flags the vulnerability in [8], demonstrating its effectiveness against vulnerabilities that have led to significant losses in real-world exploits.

Comparison to Securify. Under the same setting as our overall sampling of Ethainter vulnerability warnings, we also perform a comparison to the recent Securify [35] tool. Securify implements two "violation patterns" that are roughly applicable to four of the vulnerabilities described in Section 3.

To ensure a fair comparison, we only consider contracts that are flagged for vulnerabilities that have an approximate analogue supported by the other tool. For instance, Securify's "unrestricted write" pattern is similar to Ethainter's tainted owner variable vulnerability specification: it models precisely the case of owner-sender guards, but without propagation of taintedness into guards. Likewise, Securify's "missing input validation" pattern corresponds approximately to Ethainter's tainted `delegatecall`/owner variable/`selfdestruct`/tainted unchecked `staticcall`. Since Securify is capable of flagging both "violations" and "warnings", we consider only "violations", as these are the most likely to be true positives.

Just as for Ethainter, we select 40 Securify-flagged contracts at random for manual inspection. (We ran Securify over a random sample of 2K contracts instead of on all 141K contracts of our dataset.)

We find that *none* of the 40 Securify-flagged contracts are truly vulnerable, giving Securify a precision (i.e., true-positive rate) of 0%. Note that although the Securify paper [35] appears to report a much higher precision, its definition of a "true violation" is the violation of a *security property* rather than the existence of an apparent end-to-end *exploitable* vulnerability. Still, our experimental results initially looked surprising so we further investigated this lack of precision. Firstly, this result is due to Securify's lack of support for context-sensitivity, data structure modeling, ownership guards and their tainting. In our random sample of 2K contracts, Securify flags a very high 39.2% for these violations, so a low precision for end-to-end vulnerabilities should not be surprising. Securify generally flags 75% of all contracts for *some* violation, with 10 or more violations per flagged contract.

For instance, a typical contract we inspected that would be severely flagged by Securify with "unrestricted write" or "missing input validation" would contain logic such as this:

```
if (balances[_from] < _value ||
    allowed[_from][msg.sender] < _value) throw;
balances[_to] += _value;
balances[_from] -= _value;
```

In this case: (i) the condition that checks for underflows is not understood (hence a "missing input validation") and (ii) the maps (e.g., `balances`) are not modeled as high-level data structures (the compiled code contains only pointer arithmetic), and hence the store gets interpreted as an "unrestricted write".

Research Question 1: Summary. The above experiment completes the picture on RQ1: Ethainter is an effective static analysis, practically relevant, flagging a usefully large number of contracts, with high precision.

6.3 Efficiency of Analysis

Ethainter is a highly efficient analysis (answering RQ2 positively). It analyzes all 141.1K non-duplicate contracts on the blockchain, corresponding to a total of 38 million lines of

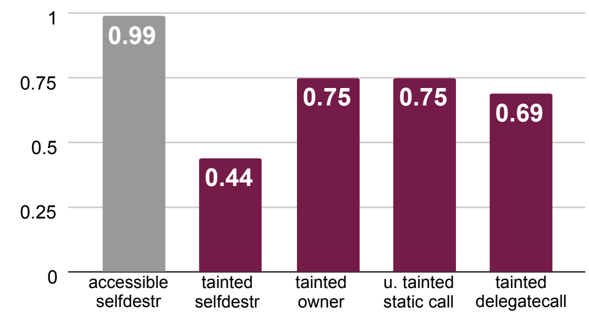
3-address code, in 6 hours (or about 200 CPU hours, given our concurrency factor of 45, plus overheads). The average analysis run-time per contract (including decompilation) is under 5 seconds. It is informative to compare the scalability of our analysis to well-known research tools for the Ethereum space, such as Oyente [25], which produces average run-times of 350 seconds with some contracts timing out after a cutoff of 60 minutes [25]. Similarly, Securify analyzes contracts over 5x more slowly than (single-thread) Ethainter (this is a lower bound, since the exact number depends on how one counts timeouts, which are more numerous for Securify) and is not parallelizable in its current form, due to its use of shared resources (e.g., temporary file). The scalability improvement is also apparent even over state-of-the-art tools such as MadMax [13], which report analysis times of 10 hours for a much smaller version of the Ethereum blockchain (92K contracts).

6.4 Design Decisions

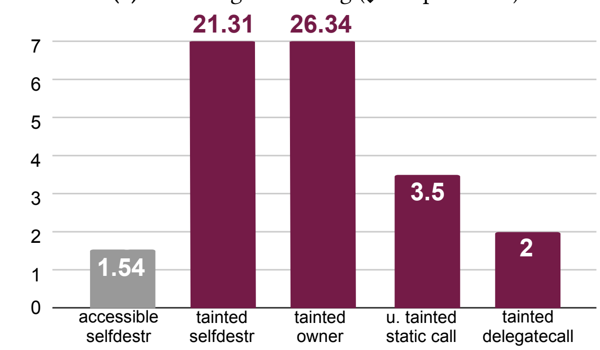
RQ3 concerns the effectiveness of the analysis under different design decisions. Ethainter optimizes for precision by modeling source language features and application design patterns that can help distinguish whether a vulnerability is likely real. One feature in which we have invested significant design effort is the realistic modeling of guards. Figure 7b shows that if we disregard the modeling of guards while keeping all other design choices constant (i.e., 'No Guard Model'), the percentage of unique programs that are flagged as vulnerable by Ethainter increases drastically. The newly flagged contracts are (overwhelmingly, if not exclusively) false positives. The increase in flagged contracts (and hence false positive rate) is most pronounced for the "tainted `selfdestruct`" vulnerability. This is because oftentimes contracts are designed to take an address as a parameter to the public function that calls `selfdestruct`, to transfer the remaining balance of the contract to this address. The Ethainter design recognizes this, and, since the `selfdestruct` is usually guarded and the taint cannot be transferred from unguarded to guarded portions of the program, the modeling is very precise.

On the other hand, if we do not allow taint to propagate via storage (and hence across multiple transactions, as is needed to execute some of the exploits that were checked manually), we obtain a sizable reduction in flagged contracts due to *incompleteness* introduced (Figure 7a). Notice that the reduction in flagged contracts is most pronounced also for the "tainted `selfdestruct`" vulnerability, since many of these exploits require overriding a guard (by overwriting an owner variable, residing in storage). Note that systems that employ symbolic execution, such as Oyente [25], tend to not consider value flow across multiple transactions.

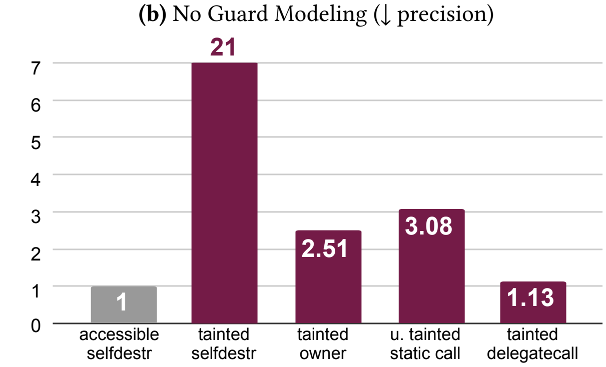
Some of the design decisions we have taken also sacrifice some of the completeness of Ethainter. If we attempt to model imprecisely but completely (much like the Securify



(a) No Storage Modeling (↓ completeness)



(b) No Guard Modeling (↓ precision)



(c) Conservative Storage Modeling (↓ precision)

Figure 7. Effect of analysis design decisions: number of analysis reports, normalized to default analysis. Chart 7a shows reduced completeness (fewer reports), Charts 7b and 7c show reduced precision. (Disclosure: due to a technical glitch, reports for "accessible selfdestruct", shown gray, are for an earlier version of our experiments than the rest.)

tool does [35]—Fig.8, non-const memory rules) storage locations that cannot be resolved as being part of a data structure or have a known constant address, we get an unacceptable decrease in precision, as shown in Figure 7c. The conservative modeling assumes that any store to an unknown location in storage can propagate to any location in storage, and the converse for loads. The false positive rate for several vulnerabilities (e.g., "tainted `selfdestruct`", "tainted owner variable") becomes significantly higher as a result.

7 Related Work

Recent security issues in smart contracts have spawned a strong interest for analysis and verification in the community, with a rapidly growing number of papers.

Our approach is a static analysis approach and analyzes EVM bytecode, modeling data structures, notions of ownership, and notions of tainted information-flow. Information flow has received a lot of attention starting with the seminal work of Denning and Denning [11], which introduced a compile-time mechanism for certifying programs as secure with respect to information-flow properties. Later works have introduced important extensions to information-flow [31]. Taint analysis (e.g., [2, 17]) is a simplified information flow problem that tracks the propagation of input values in a program.

Previous works for smart contracts can be categorized according to their underlying techniques, including symbolic execution, formal verification, and abstract interpretation.

Systems including Oyente [25], SASC [42], Maian [28], GASPER [7], teEther [22] and ECFChecker [16] use a symbolic execution/trace semantics approach that (compared to exhaustive static analysis) is much less complete/exhaustive, since only some program paths of smart contracts will be explored. Oyente is probably the earliest and best-known representative of such work: it checks for security vulnerabilities including transaction order dependency, time-stamp dependency, and reentrancy. Maian [28] analyzes multiple invocations of a smart-contract and covers safety properties including prodigality and self-destruction. TeEther [22] is a recent representative of symbolic execution tools, offering both detection of information-flow-like properties and exploit generation. Due to these similarities, we considered a thorough experimental comparison with teEther, however the difference in completeness between the static analysis and symbolic execution approaches is immediately evident in a small experiment: on 20 hand-checked contracts flagged by Ethainter, teEther does not flag at all 13, times out on 5 after 120s, and exits with an exception for 2. Although symbolic execution warnings are expected to have high precision, the goal of our work is to combine feasibly high precision and completeness.

The smart contract analysis literature also includes formal verification approaches, with more emphasis on deep modeling and less on automatic analysis. The Zeus framework [21] translates Solidity source code to LLVM bitcode [24] before performing the actual analysis in the SeaHorn verification framework [32]. This approach however fails to abstract all Solidity instructions and the EVM execution platform correctly. An alternative approach [26] abstracts Solidity code to finite-state automata. EtherTrust [15] is a sound static modeling tool using a small-step semantics. EtherTrust analyzes EVM bytecode directly for single-entrancy vulnerabilities, using Z3 as an underlying SMT solver.

Rodler et al. [30] present an interesting approach for protecting contracts vulnerable to reentrancy vulnerabilities, instead of merely detecting such vulnerabilities. Interestingly, they confirm (for an attack domain different from our information-flow vulnerabilities) that Securify employs "very conservative violation pattern[s] ..., which consequently results in a very high false positive rate."

Other tools [18, 19, 36] employ fuzzing techniques in order to detect various vulnerabilities. The recent ILF [18] tool trains a neural network using inputs generated by symbolic execution in order to increase fuzzing effectiveness.

Various exploits have been broadly identified in the literature [3, 4, 10, 33]: exploits related to Solidity, the EVM and the blockchain itself. These exploits have been highlighted by the community outside of publications. For instance, the security company Consensys maintains a website [9] outlining the exploits mentioned in the literature, as well as additional exploits, such as data overflows and underflows, and suggestions on how to write better smart contracts (such as isolating external calls into their own transactions, instead of executing them in a single transaction, to minimize the risk of failures and side-effects).

Finally, taint analysis tools for other domains are in abundance and some have used Datalog, as in our work. Livshits [23] has fruitfully explored the use of Datalog for taint analysis in Java. P/Taint [14] is a recent declaratively specified unified taint and pointer analysis framework for both Java and Android. Its implementation methodology allows full-featured context-sensitive taint analysis frameworks to emerge out of existing pointer analysis frameworks at little additional implementation and run-time cost.

8 Conclusions

We presented the Ethainter framework for analyzing smart contracts for security vulnerabilities. Our framework employs notions of information-flow for tracking tainted values, while extending them to model key concepts for the domain, such as sanitization via guards and taint through persistent storage. We give a specification of the analysis in a distilled formalism, likely of independent value. We identify practical information-flow vulnerabilities in the full Ethereum blockchain, e.g., introduce taint to bypass an ownership guard to destroy the contract. The analysis is finely tuned for precision and scalability. Ethainter identifies real security issues in deployed contracts, all the way to automatic exploitation of hundreds of contracts on the Ropsten test network.

References

[1] [n. d.]. 0x: Powering the decentralized exchange of tokens on Ethereum. https://0x.org.

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick

McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Springer-Verlag New York, Inc., New York, NY, USA, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

[4] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2020. Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact. *Future Generation Computer Systems* 102 (2020), 259 – 277. https://doi.org/10.1016/j.future.2019.08.014

[5] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *CoRR* abs/1809.03981 (2018). arXiv:1809.03981 http://arxiv.org/abs/1809.03981

[6] Vitalik Buterin. 2013. A Next-Generation Smart Contract and Decentralized Application Platform. https://github.com/ethereum/wiki/wiki/White-Paper.

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

[8] cnbc.com. 2018. 'Accidental' bug froze $280 million worth of ether in Parity wallet. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html

[9] Consensys. 2018. Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/ Accessed: 2019-11-19.

[10] Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. 2015. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *IACR Cryptology ePrint Archive* 2015 (2015), 460.

[11] Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (July 1977), 504–513. https://doi.org/10.1145/359636.359712

[12] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *International Conference on Software Engineering (ICSE)*.

[13] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proc. ACM Programming Languages* 2, OOPSLA (Nov. 2018). https://doi.org/10.1145/3276486

[14] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: Unified Points-to and Taint Analysis. *Proc. ACM Programming Languages (PACMPL)* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133926

[15] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 51–78.

[16] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Programming Languages* 2, POPL, Article 48 (Dec. 2017), 28 pages. https://doi.org/10.1145/3158136

[17] Christian Hammer and Gregor Snelting. 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.* 8, 6 (2009), 399–422. https://doi.org/10.1007/s10207-009-0086-1

[18] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, New York, NY, USA, 531–548. https://doi.org/10.1145/3319535.3363230

[19] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[20] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[21] Sukrit Kalra, Seep Goel, Seep Goel, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[22] Johannes Krupp and Christian Rossow. 2018. TEETHER: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, Berkeley, CA, USA, 1317–1333. http://dl.acm.org/citation.cfm?id=3277203.3277303

[23] Benjamin Livshits. 2006. *Improving Software Security with Precise Static and Runtime Analysis*. Ph.D. Dissertation. Stanford University.

[24] LLVM. 2018. The LLVM Compiler Infrastructure Project. https://llvm.org/

[25] 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 (CCS '16)*. ACM, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[26] Anastasia Mavridou and Aron Laszka. 2018. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In *Proceedings of the 7th International Conference on Principles of Security and Trust (POST)*.

[27] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. https://www.bitcoin.org/bitcoin.pdf.

[28] 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 (ACSAC '18)*. ACM, New York, NY, USA, 653–663. https://doi.org/10.1145/3274694.3274743

[29] Daniel Pérez and Benjamin Livshits. 2019. Smart Contract Vulnerabilities: Does Anyone Care? *CoRR* abs/1902.06710 (2019). arXiv:1902.06710 http://arxiv.org/abs/1902.06710

[30] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/sereum-protecting-existing-smart-contracts-against-re-entrancy-attacks/

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

[32] SeaHorn. 2018. SeaHorn | A Verification Framework. http://seahorn.github.io/

[33] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *Financial Cryptography and Data Security*, Michael Brenner, Kurt Rohloff, Joseph Bonneau, Andrew Miller, Peter Y.A. Ryan, Vanessa Teague, Andrea Bracciali, Massimiliano Sala, Federico Pintore, and Markus Jakobsson (Eds.). Springer International Publishing, Cham, 478–493.

[34] Software Reliability Lab ETH Zurich. 2017. Securify: Formal Verification of Ethereum Smart Contracts. http://securify.ch/.

[35] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

[36] Various. [n. d.]. Echidna - Ethereum fuzz testing framework. https://github.com/crytic/echidna. Accessed: 2019-11-20.

[37] Various. [n. d.]. ETHSecurity Community on Telegram. Accessed: 2019-05-11.

[38] Various. 2018. GitHub - ethereum/solidity: The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity

[39] wired.com. 2016. A $50 Million Hack Just Showed That the DAO Was All Too Human. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/

[40] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. http://gavwood.com/Paper.pdf.

[41] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf.

[42] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara. 2018. Security Assurance for Smart Contract. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–5. https://doi.org/10.1109/NTMS.2018.8328743