Ethainter: Composite Information Flow Security for Ethereum Smart Contracts

Anonymous Author(s)

Abstract

Smart contracts on blockchain platforms represent a software domain with critical correctness needs. We introduce the Ethainter framework, which performs a static security analysis of smart contracts, based on information-flow principles. Ethainter operates on Ethereum—the dominant smart contract blockchain platform—and is partly informed by the domain-specific circumstances of Ethereum smart contracts. Ethainter identifies unique, novel combinatorial attacks to smart contracts, such as tainting to bypass an ownership guard, followed by an untrusted operation that fails to be sanitized. The analysis scales to the entire blockchain, consisting of over 140K unique smart contracts (deployed over millions of accounts). Ethainter flags security issues in contracts collectively corresponding to Ether balances currently equivalent to 976M USD. Manual inspection of a small subset of the flagged contracts reveals many valid warnings, and we demonstrate a realistic complex exploit in a deployed contract.

1 Introduction

Blockchain platforms, such as Bitcoin [38] and Ethereum [7, 52], have the potential to revolutionize all sectors of multi-party interaction, by enabling decentralized, resilient consensus. A blockchain can securely and permanently register transactions (e.g., trading between any partners) in a way that requires no centralized or trusted authority. Trust arises from the collective agreement of a large number of participants in a widely distributed network, and such collective agreement is unforgeable. Current applications of blockchain technology mostly center around the financial exchange, with prominent deployment in cryptocurrencies and banking tasks (e.g., lending, crowdfunding). However, the technology can be applied much more widely. The Ethereum blockchain platform has recently attracted significant attention. Ethereum embodies a generalization and next generation of blockchain applications. Instead of a passive state, it allows arbitrary programs, called *smart contracts*, to represent accounts on the blockchain. Smart contracts are registered immutably on the blockchain and operate autonomously. Therefore, the smart contract domain is one where software correctness is most critical: the contracts are high-value targets (since they manage monetary assets), cannot be quickly patched, and 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 [9, 51].

Conference'17, July 2017, Washington, DC, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Information-flow violations are a major class of security vulnerabilities in modern software. If untrusted ("tainted") information from external sources is allowed to propagate to critical program points, it can alter key aspects of program behavior. In smart contracts, specific coding patterns are employed to prevent an arbitrary user from executing sensitive operations such as destroying the contract. The contract code typically checks 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. It is essential to model this guarding mechanism, and its ensuing vulnerabilities, for any high-fidelity information-flow analysis of smart contracts: if guards are not taken into consideration, many contracts will be incorrectly inferred to be vulnerable, i.e., they would be vulnerable, however the only user that can "exploit" these vulnerabilities would be the owner of the contract themself. Conversely, the guarding code can itself be attacked, by invoking code that manipulates the state of the contract in unexpected ways. The most notorious example of this pattern in practice is the Parity wallet hack, where the equivalent of $280M [9] was stolen or frozen via two separate vulnerabilities. The vulnerabilities involved re-initializing part of the contract that sets owner variables, via a vulnerable library function, before subsequently attacking the contract.

In this work, we present Ethainter: a static analysis for information-flow violations in Ethereum smart contracts. Ethainter aims to detect *composite* information-flow vulnerabilities, such as those sketched above. That is, in addition to the straightforward flow of tainted information to sensitive operations, Ethainter seeks to model several elements: guards, the tainting of guard conditions, other potential ineffectiveness of guards, as well as different kinds of taint that are not prevented by guarding.

The novelty of the Ethainter work can be summarized as follows:

- We formulate an information-flow analysis that takes into account the common sanitization (i.e., guarding) practices in smart contracts. The specification of the analysis is in a formalism of independent value. It ignores orthogonal, well-understood technical complexities, instead of capturing the essence of new information-flow concepts (e.g., tainted guards, taint through storage) in minimal form.

- We identify practical information-flow vulnerabilities in Ethereum. Some of these are straightforward and similar to other contexts—e.g., making a call to a recipient contract value that can be tainted by user input. However, others are highly unique, tuned to the relevant properties of our formulation. For instance, we identify relevant attacks that first introduce taint to bypass an ownership guard, then cause the contract to self-destruct.
- We balance the precision and scalability of the analysis with appropriate tuning choices. This results in a mutually-recursive definition of tainting and overall information flow. For instance, a tainted memory store is considered to induce further information flow a lot more eagerly than an untainted (yet still statically undetermined) memory store.
- 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 3 hours. This permits a large-scale study of all contracts, to identify several thousands of vulnerable contracts. We manually inspect a few tens of flagged contracts and determine that several of them are indeed vulnerable, producing a high true-positive rate for the Ethainter analysis, over complex vulnerabilities—notably, a 40% true-positive rate among contracts with a non-zero Ether balance. To further validate our attack model, we exploit one such flagged contract. (The exploit is applied to an Ethereum testnet and proves the feasibility of an attack without perpetrating it.)

Illustration. Consider a contract flagged by Ethainter as vulnerable to the "accessible `selfdestruct`" and "tainted owner variable" vulnerabilities. The following code snippet[1] shows how the public function `remove()` is implemented in this contract. In this contract, the owner address is stored in storage slot 0x0, shown here as `S0`. This function contains a correct owner-sender guard, and only calls `selfdestruct` if the message sender is the contract's owner.

```
function remove() public {
  // guard: exit if called by non-sender
  if (!(address(msg.sender) == address(S0))) {
    exit();
  } else {
    selfdestruct(address(msg.sender));
  }
}
```

However, the contract contains the following public function `setAmbiAddress` (irrelevant details elided). This function sets the contents of `S0` to the tainted value passed in as argument `varg0`, provided that the prior value of `S0` is `0x0`.

```
function setAmbiAddress(address varg0...) public {
  if ((0x0 == address(S0))) {
    ...
    // overwrite owner with value in varg0:
    S0 = (varg0 | (~0xffffffffffffff... & S0));
  }
  return 0x0;
}
```

Hence, provided that the value in storage slot 0x0 is 0, we can exploit the accessible `selfdestruct` vulnerability by first calling `setAmbiAddress(<our_address>,...)`, followed by a call to `remove()`. We successfully tested this exploit on a private chain. We discovered two instances of this contract on the public chain: one which is not exploitable because someone has already called `setAmbiAddress`, and one which is exploitable at the time of writing and has been so since it was deployed in 2016.

2 Background

Next, we provide a brief background description of blockchain platforms, Ethereum smart contracts, and the language abstractions behind them.

2.1 Blockchain Platforms, Ethereum, and Smart Contracts

A blockchain is a distributed, append-only ledger of transactions, secured using cryptography. One can think of a blockchain as a long and ever-growing list of blocks, each encoding a sequence of individual transactions, always available for inspection and safe from tampering. Each block contains a cryptographic signature of its previous block. Thus no previous block can be changed or rejected without also rejecting all its successors.

Decentralization is a crucial property of blockchains. Rather than trust being bestowed to a central authority, it arises from a majority consensus among all peers in the network. This property has led to assertions that "*Blockchain is a foundational technology: It has the potential to create new foundations for our economic and social systems*" [27].

In this work, we are concerned with the Ethereum platform [7, 52]: an extension of the conventional model popularized by Bitcoin [38]. Unlike the conventional model where each account encodes just a simple balance, accounts in Ethereum may contain distributed programs known as *smart contracts*. Smart contracts can perform arbitrary computations, enabling the encoding of virtually any complex interaction, with its logic permanently and transparently stored on the blockchain: responding to communication from other accounts, dispensing or accepting funds, etc. Such logic can, e.g., encode an interest schedule, a bet, an investment agreement, a lending policy, conditional trading directives,

[1]The language of our examples is that of the public *Contract Library* service [1], which gives decompiled versions of all deployed Ethereum contracts, with occasional manual massaging for conciseness.

payment-upon-receipt agreements, pricing dependent on geographic or other environmental input, and a lot more.

2.2 Smart Contract Programming: Solidity and the EVM

Typically, Ethereum smart contracts are written in the domain-specific Solidity [48] language, which bears some resemblance to JavaScript with the addition of static types, various kinds of assertions, contracts as class-like constructs with inheritance, and numerous other features. Although Solidity is the only officially-supported solution, plenty of other languages for writing smart contracts exist, including Serpent, Vyper, and LLL [46, 47, 49]. All of these languages, however, are significantly removed from the low-level machine code that is actually stored and executed.

Instead, contracts are stored in a compact bytecode language and executed by a built-in low-level stack machine called the *Ethereum VM (EVM)*. Alongside standard arithmetic operations, the EVM instruction set includes primitives for cryptographic hashing, primitives for identifying contracts and performing inter-contract message calls, and exception-related instructions. Data is stored either on the blockchain, in the form of a persistent data structure called *storage*, or in contract-local memory.

Storage exists as a segregated, near-infinite (256-bit) map structure per contract. Primitive state variables are assigned to constant storage locations, starting with storage location 0. As we discuss in more detail in Section 4.3, the location of a dynamic structure in storage is the cryptographic hash of the structure's compiler-assigned numeric identifier.

2.3 Analysis Level

We analyze smart contracts at the bytecode level, which is a decision well aligned with the goal to detect vulnerabilities over all deployed contracts in the wild. It is estimated that the high-level source code of EVM contracts is only available for a tiny fraction (just 0.34%) of contracts on etherscan.io [14, 18]—the best-known Ethereum blockchain exploration site. Targeting the EVM bytecode level offers transparent support for all high-level languages that target the EVM. Furthermore, Solidity contracts can be interleaved with inline assembly, which is similar in structure to EVM bytecode.

When Solidity code is available, an analysis at the abstract syntax tree (AST) level needs to handle different kinds of code complexities and indirections, such as desugaring syntax or performing intermediate assignments before using a value. This is compounded by the fact that Solidity is a rapidly evolving language (whereas the EVM is not evolving as much). A source-level analysis therefore needs to take into consideration the version of the compiler to accurately model the underlying semantics of the program. This is particularly hard for one of the vulnerabilities that we describe next in Section 3, tainted storage index. Depending on the compiler version, different heuristics are used to determine whether to bounds-check storage indices. By analyzing the bytecode directly, we eliminate the need to have our analysis second-guess the semantics and focus on what will be executed on the EVM. At the EVM bytecode level, the input is normalized, with all control flow being explicit, uniform, and simplified.

3 Example Vulnerabilities

We next present examples 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 a direct 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.[2] 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 can very quickly have economic consequences. 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` that is used in function `kill` to check whether the address of the caller of the contract (denoted as `msg.sender`) is the same as the address of the owner of the contract. However, in this contract, a public setter function called `init_owner` makes it possible for anyone to update the `owner` to an arbitrary value, making the contract vulnerable.

```
address owner = ...;

function init_owner(address owner) public {
    this.owner = owner;
}

function kill() public {
```

[2]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.

```
    if (msg.sender == owner) {
        // sensitive operation
    }
}
```

3.2 Tainted delegatecall

The EVM's `delegatecall` instruction allows any code that gets 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 any contract user to cause the contract to call another contract of their choosing. This is probably the most serious kind of vulnerability presented in this paper. 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(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);
}
```

As we already saw in the real-world fragment of the Introduction, the vulnerability is more commonly encountered in combination with a tainted guard.

3.4 Tainted selfdestruct

A variant of the previous vulnerability, which can directly result in profit to the attacker is the *tainted* `selfdestruct` vulnerability. Here, the address into which the funds are sent 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 init_admin(address admin) public {
    administrator = admin;
}

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

3.5 Tainted Store Index

Most of a contract's state information is stored inside *storage*, a word-addressable contract-specific array of machine words, persistently maintained on the blockchain. Owner variables, balances, holdings of tokens, etc. are oftentimes stored here. Most accesses to storage are opaque at a source or ABI level, as the compiler typically abstracts away storage as data structures or other global variables and does a good job of checking bounds to ensure that writes to one data structure do not overwrite arbitrary parts of storage. On the other hand, our analysis of bytecode suggests that there are limited examples when any user of a contract can overwrite arbitrary parts of storage. For instance, the following source program shows that `setAccount` can be used to arbitrarily modify most parts of the storage, if the compiler does not correctly check array bounds, or if the user uses inline assembly to access storage without bounds checking.

```
address[] public accounts =  [
  0x36eaf79c12e96a3dc6f53426ce41c81b62ee992e,
  0xf235aa56dd96bda02acfb361ee9fa3ec61326ce4
];

function setAccount(uint n, address v) public {
    accounts[n] = v; // Assuming no bounds check
}
```

More realistic tainted store attack scenarios involve tainting unexpected parts of a single structure.

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 is conventional, while emphasizing the exact elements that are of particular interest to our analysis. As a result, the input language we consider is designed to be minimal and not Turing-complete: it models taint sources and sinks, transfer through numeric operations, loads and stores to persistent storage, and input sanitization through guards; while omitting technicalities otherwise essential to computation, such as conditionals and calls.[3] At a high-level, the emphasis of the formalism is in capturing in

[3]One can think of the input as a trace language and not as a programming language. However, the rules we give over this language define a static analysis and not a trace semantics: we will explicitly mark which static inferences are over-approximate or under-approximate, as opposed to fully accurate.

the simplest possible information-flow terms the following insights:

- The notion of programmatic guarding/sanitizing so that tainted input cannot propagate.
- The idea that only caller input can be sanitized via guards, whereas, if taint propagates to the persistent storage of the contract, it can elude guards.
- The danger 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 input language instructions 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 A-normal form (i.e., all expressions are primitive—there are no nested expressions) and in 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 `:=` `OP`(y, z)" with the operator being equality.

The `GUARD` instruction deserves some attention: "x `:=` `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 in smart contracts 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. The translation from common input programs in smart contract high-level languages (e.g., Solidity) to our input language is, therefore, involved. 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 `:=` (`sender` `=` z)" (where z is the result of an `SLOAD` from the persistent storage location holding the contract's owner).

Figure 2 shows the relations computed by the information flow analysis. As can be seen, there are two different kinds of taint for variables: $\downarrow^I x$ (*InputTaintedVar*) and $\downarrow^T x$ (*StorageTaintedVar*). This will be an important distinction in the analysis, just as it is in practice: taint from unsanitized user 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 use of sender guards. Relation $\ddagger p$ (*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 minimal 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 DSA(x) and DS(x).

4.2 Information-Flow Core Analysis

Figure 3 shows the information-flow analysis model. 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.

We next discuss the rules, pointing out the ones that deserve more attention.

- LoadInput, Operation-1, Operation-2: These rules are standard, for 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 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. This is clearly a powerful inference, indicating the analysis design choice to disallow tainted writes to addresses that may be influenced by an attacker. We discuss this and other design choices and their impact in Section 4.4.
- Violation, StorageLoad: These rules are also mostly standard. Loads from tainted constant storage locations introduce storage taint into a local variable. Either kind

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	$\downharpoonleft^I x$	Variable x is tainted from input.
StorageTaintedVar	$\downharpoonleft^T x$	Variable x is tainted from storage.
TaintedStorage	$\downharpoonleft^T S(v)$	Storage location with constant address v is tainted.
NonSanitizingGuard	$\not\upharpoonleft p$	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).

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

$$(\textsc{Guard-1})\ \frac{x \ := \texttt{GUARD}(*, y) \qquad \downharpoonleft^T y}{\downharpoonleft^T x} \qquad (\textsc{Guard-2})\ \frac{x \ := \texttt{GUARD}(p, y) \qquad \downharpoonleft^I y \qquad \not\upharpoonleft p}{\downharpoonleft^I x}$$

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

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

$$(\textsc{Uguard-T})\ \frac{p \ := (\texttt{sender} = z) \qquad z \sim S(v) \qquad \downharpoonleft^T S(v)}{\not\upharpoonleft p} \qquad (\textsc{Uguard-NDS})\ \frac{p \ := (y = z) \qquad \neg\mathrm{DS}(y) \qquad \neg\mathrm{DS}(z)}{\not\upharpoonleft p}$$

Figure 3. Inference rules for information-flow analysis.

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

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

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

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

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

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 $\ddagger p$ be 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 also offer 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 $\ddagger p$). 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

The formulation of the analysis shown is careful to capture several design decisions that carry over to the full implementation of Ethainter, i.e., to a practical analysis that aims to balance *completeness* and *precision*. The tradeoff between these two factors is the crucial design element of a static analysis for error detection. The analysis designer has to decide when to be *over-approximate* in the modeling of program behavior, thus warning about more potential errors, but with a possible cost in precision, i.e., introducing *false positives*. Conversely, the analysis can *under-approximate* program behavior, focusing instead on high-confidence inferences of violations, for better precision (i.e., fewer false positives) at the expense of possibly lower completeness, i.e., missing potential errors.

In our analysis formulation, the following design elements come in play:

- 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 under-approximate: 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 over-approximate with respect to control flow: whenever two program paths meet, the union of inferences is propagated.

However, the formulation of the relations themselves is under-approximate: 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. In turn, this means that 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 exhibits a definite over-approximate flavor 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 reflects much of the completeness of the analysis, i.e., it increases its ability to capture errors, but possibly yields false positives.

5 Implementation

The full implementation of the Ethainter system follows the insights of the simplified model of Section 4, but adds significant complexity in dealing with realistic contracts. The analysis of Ethainter consists of several hundred declarative rules in the Datalog language, which is equivalent to first-order logic with recursion [28]. 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 decompilation toolchain that powers the public *Contract Library* service [1]. We consider all smart contracts deployed on Ethereum as of Aug. 30, 2018, comprising more than 140K unique EVM bytecode programs. The Ethainter implementation back-end is the high-performance Soufflé [31] Datalog engine, which translates Datalog input relations into highly-optimized C++ code.

In analysis terms, the implementation add-ons compared to the simplified model can be summarized as follows:

- The implementation uses a standard intra-procedural constant propagation data-flow analysis for propagating constant values (i.e., relation $C(x) = v$ of Section 4) and a 1-call-site-sensitive points-to 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.
- 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 going from a highly simplified input language to realistic programs—the spirit of the analysis logic is well-captured by the simplified model.

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.

Our experiments aim to answer the following research questions:

RQ1. Is our information-flow framework an effective static analysis?

RQ1.A. Is the analysis relevant? What percentage of contracts are vulnerable, and how much currency is at stake?

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?

RQ3.A. Do our experiments on real contracts provide insights that justify our design decisions?

6.1 RQ1: Effectiveness of Analysis

We evaluate the effectiveness of Ethainter by examining the percentage of contracts flagged for each vulnerability, and the total currency balances held in accounts with flagged bytecode. We also select a subset of programs for hand checking, and use this to calculate a false-positive rate for our analysis.

In total, our analysis flags 16.55% of all unique EVM programs for at least one vulnerability. (However, this number is dominated by the "accessible `selfdestruct`" vulnerability, which applies mostly to already-dead contracts.) This corresponds to 44.91% of all contracts, holding a combined balance of 4.72M Ether, equivalent to 976M USD as of exchange rates on 30 Aug, 2018.

0.13% of unique programs are flagged for the "tainted `delegatecall`" vulnerability, 1.32% for "tainted owner variable", 12.49% for "accessible `selfdestruct`", 0.60% for "tainted `selfdestruct`", and 3.27% for "tainted store index". The effect of these vulnerabilities has tangible impact on the Ethereum ecosystem. Throughout our experiments we found that unique programs that are flagged for "accessible `selfdestruct`" are far less likely to contain non-zero Ether: 0.3% vs 0.8%. This is likely because these contracts have already been destroyed, either by their owners or others.

To estimate a true positive rate for Ethainter, we systematically selected two batches of 10 flagged contracts each for manual inspection. To make manual inspection feasible, we limit ourselves to contracts containing fewer than 200 basic blocks. Contracts were sampled by performing a lexicographical sort on their addresses and selecting the first two contracts that were flagged by Ethainter for each kind of vulnerability. The first batch was selected from contracts containing *any* Ether balance, including a zero balance, and the second batch from contracts containing a balance greater than 0.0001 Ether. We theorize that contracts that hold Ether are likely to have been more thoroughly scrutinized, and thus produce a higher false-positive rate. Among the contracts that contain any Ether balance, 50% are found to be true positives. Among only those that contain a balance greater than 0.001 Ether, 40% are found to be true positives. The results from our systematic hand checking are shown in Table 1.

In order to preserve the identity of contracts until we have a responsible disclosure agreement in place, we will refer to contracts by three hexadecimal digits contained in their identifier. Contracts `FDE` and `F32` were flagged for "accessible `selfdestruct`"; both of these immediately destroy themselves. We speculate that they are either simple 'test' contracts or honeypots.

Contract `059` is susceptible to a "tainted `delegatecall`", which is a serious vulnerability, but it depends on the attacker having control on a third party address used for checking addresses. If this third party functionality (which is yet to be initialized on the mainnet) is initialized with the identity function, an attacker can call `sweep` with yet another address. This time the address is used in a `delegatecall`, allowing the supplied contract under the address to take over the entire functionality of contract `059`.

Contract `82A` is easily exploitable since it is vulnerable to tainted `delegatecall`. The attacker only needs to supply a custom smart contract address to a public function that is currently implemented. This can take over the functionality of contract `82A`, if the attacker's contract contains at least 0.01 Ether (around $2).

Contract `5A2` contains a tainted owner variable in storage slot 3. The tainted owner update is guarded by a success status check of a message call to a tainted address read from storage slot 0. To exploit this vulnerability, an attacker can first update the address in storage slot 0 to ensure that the aforementioned message call succeeds, followed by a call to the public function whose signature ends in `3b9`, passing in a new owner address as an argument.

Contract `384` is susceptible to tainted store address, where the entirety of the storage memory can be rewritten by the attacker (parts of storage can only be accessed by forcing integer overflows). The severity of the attack depends on the functionality offered by the contract, which was not entirely clear for this contract.

Contract `52F` can be exploited in a similar way to contract `059`. Contract `E40` appears to be vulnerable to tainted storage address, since a storage index depends on an operand supplied to a public function.

We mark contracts `1EE` and `91D` as "maybe" vulnerable, because a successful attack relies on the behavior of external message calls, which is not possible to predict ahead of time. Contract `1EE` may be vulnerable in a similar way to contract `52F` if an attacker can taint the data returned by an external message call. Similarly, contract `91D` may be exploitable if an attacker can force an externally-called contract to make a reentrant call, since the contract guards its tainted `selfdestruct` instruction by comparing the message sender with its own address. For the purpose of calculating our true positive rate, we count these "maybe" cases as not vulnerable.

6.2 RQ2: Efficiency of Analysis

Ethainter is a highly efficient analysis. Ethainter analyzed all 141.1K unique contracts on the blockchain, corresponding to a total of 38 million lines of 3-address code, in 3.33 hours. The average analysis run-time per contract (including decompilation) was 2.47 seconds. Ethainter timed out for less than 1% of contracts. It is informative to compare the scalability of our analysis to well-known research tools for the Ethereum space, such as Oyente [36], which produces average run-times of 350 seconds with some contracts timing out after a cutoff of 30 minutes [36]. The scalability improvement is also apparent even over state-of-the-art tools such as MadMax [18], which report analysis times of 10 hours for a much smaller version of the Ethereum blockchain (92K contracts)

6.3 RQ3: Design Decisions

To produce a useful analysis, we optimize for precision by modeling source language features and application design patterns that can help us distinguish whether a vulnerability is likely real. One feature in which we have invested significant design effort is the realistic modeling of guards. Table 2 shows that if we disregard the modeling of guards while keeping all other design choices constant (i.e., 'No Guard Model'),

Table 1. Summary of hand-checked contracts.

Contract	Balance?	Flagged For	Vulnerable?	Description
FDE	✗	Accessible `selfdestruct`	✓	Immediately `selfdestruct`s to constant address
F32	✗	Accessible `selfdestruct`	✓	Immediately `selfdestruct`s to constant address
5A5	✗	Tainted `selfdestruct`	✗	Tainted via SHA3 hash
0F1	✓	Tainted `selfdestruct`	✗	Tainted via SHA3 hash
F2C	✗	Tainted Owner Variable	✗	Unconventional owner guard via private function call
8B7	✗	Tainted Owner Variable	✗	Bounds-checked tainted storage index
059	✗	Tainted `delegatecall`	✓	Tainted via memory with `calldatacopy`
82A	✓	Tainted `delegatecall`	✓	Public function argument directly passed to `delegatecall`
5A5	✗	Tainted Store Index	✗	Tainted via SHA3 hash
384	✗	Tainted Store Index	✓	Integer overflows allow storage overwrites
264	✓	Accessible `selfdestruct`	✗	Guarded by hard-coded constant address check
5B3	✓	Accessible `selfdestruct`	✗	Guarded by data structure permission flag
91D	✓	Tainted `selfdestruct`	Maybe	Tainted `selfdestruct` instruction guarded by sender comparison to `address(this)`
EDC	✓	Tainted `selfdestruct`	✗	Guarded by owner check that only becomes tainted via a SHA3
0AE	✓	Tainted Owner Variable	✗	Guarded by hard-coded constant address check
5A2	✓	Tainted Owner Variable	✓	Guarded by success of external call
52F	✓	Tainted `delegatecall`	✓	Tainted via memory with `calldatacopy`
1EE	✓	Tainted `delegatecall`	Maybe	Tainted `delegatecall` address is return of CALL to static address
E40	✓	Tainted Store Index	✓	Bounds checked, but allows indexing arbitrarily to value in a data structure
851	✓	Tainted Store Index	✓	Tainted index into unresolvable data structure

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. Our 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 get a sizable reduction in flagged contracts due to incompleteness introduced (i.e., 'No Storage Model'). 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 [36], 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 completely model storage locations which 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 column 'Complete Storage'. The complete 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.

6.4 Threats to Validity

In this section we discuss some threats to the validity of our evaluation methodology.

Destroyed Contracts. Our experiments use a corpus consisting of all contracts that have ever been deployed on

Table 2. Percentage of flagged contracts with different analysis design decisions. Percentages shown for 'baseline' are for the Ethainter system design as presented in this paper.

Vulnerability	**Baseline** ☺	**No Storage Model** (↓ completeness)	**No Guard Model** (↓ precision)	**Complete Storage** (↓ precision)
accessible `selfdestruct`	**12.60%**	12.46%	19.44%	12.62%
tainted `selfdestruct`	**0.16%**	0.07%	3.41%	3.36%
tainted owner variable	**1.37%**	1.03%	36.08%	3.44%
tainted store index	**3.36%**	2.62%	6.90%	8.65%
tainted `delegatecall`	**0.16%**	0.11%	0.32%	0.18%

Ethereum, including the bytecode of contracts which may have since been destroyed. Destroyed contracts that were historically vulnerable are no longer actively exploitable, but may still hold a balance that would be reflected in our results.

Longitudinal Validity. Contracts developed in the future, and those developed more recently than others, may not follow the same development patterns with respect to, e.g., guard conditions, due to better developer awareness of security issues and the increasing availability of analysis tools. Our analysis does not take into account the level of security awareness and context at the time at which each contract was developed.

Human Inspection. Manual inspection is the weakest link in our evaluation. We allocated an hour for the manual checking of each selected contract. Although we performed detailed manual inspection of a subset of contracts flagged by Ethainter, due to time constraints we were limited to a small subset of 20 contracts, and only formulated and verified an exploit for one of the true positives. The results may also be biased by our decision to select contracts that contained fewer than 200 basic blocks (a necessary concession to make manual analysis feasible); for instance, contracts containing more lines of code may be more likely to contain vulnerabilities.

7 Related Work

Recent security issues in smart contract has 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 (i.e., more exhaustive, yet possibly with false positives), that applies ideas from information-flow literature. Our approach 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 [13], 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 [40]. Taint analysis [23, 41] 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 [36], SASC [53], Majan [39], GASPER [8] and recent work [22] use a symbolic execution/-trace semantics approach that is fundamentally incomplete, since only some program paths of smart contracts will be explored. Oyente checks for some security vulnerabilities including transaction order dependency, time-stamp dependency, and reentrancy. SASC [53] extends Oyente with additional vulnerabilities detection and a visualization of the detected vulnerabilities. Majan [39] analyzes multiple invocations of a smart-contract and covers safety properties including prodigality and self-destruction, as well as liveness properties, such as greediness. GASPER focuses on gas-costly patterns in programs. The work of Grossman et al. [22] focuses on a safety property called Effectively Callback Free (ECF) to detect reentrancy bugs.

Semi-automated *formal verification* approaches have also been proposed [2, 6, 15, 21, 24, 25] for performing complete analyses of smart contracts using interactive theorem provers such as Isabelle/HOL [29], F* [16], Why3 [15], and $\mathbb{K}$ [32]. These approaches have a common theme: a formal model of a smart contract is constructed and mathematical properties are shown via the use of a semi-automated theorem prover.

Recently, a complete small-step semantics of EVM bytecode has been formalized for the F* proof assistant [21]. Another similar effort, KEVM [24], uses the $\mathbb{K}$ framework, based on reachability logic. Due to their reliance on semi-automated theorem provers, which require substantial manual intervention for proof construction, these formal verification approaches do not scale for analyzing the millions of smart contracts currently deployed on the blockchain.

In contrast to formal verification work for smart contracts, abstract interpretation approaches [33, 37] do not require human intervention; however, they introduce false positives. The Zeus framework [33] translates Solidity source code to LLVM bitcode [35] before performing the actual analysis in the SeaHorn verification framework [42]. This approach however fails to abstract all Solidity instructions and the EVM execution platform correctly. An alternative approach is that of Mavridou and Laszka [37], in which Solidity code is abstracted to finite-state automata. EtherTrust [20] is a sound static analysis tool using a small-step semantics. EtherTrust analyzes EVM bytecode directly for single-entrancy vulnerabilities, using Z3 as an underlying SMT solver.

Vandal [50] is a scalable open-source framework for decompiling EVM bytecode. This approach has been recently used for detecting gas-vulnerabilities [18] in smart contracts.

Various exploits have been broadly identified in the literature [4, 5, 12, 43]: exploits related to Solidity, the EVM and the blockchain itself. These exploits have been highlighted by the community outside of publications. For instance, the company Consensys [10] maintains a website [11] outlining the exploits mentioned in the literature, as well as additional exploits, such as data overflows and underflows, and suggestions 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. Livshits [34] has fruitfully explored the use of Datalog for taint analysis in Java. P/Taint [19] 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. Tripp et al. [44] express taint analysis as a demand-driven problem, instead of a complete all-program-points flow. There are many Android taint analysis frameworks, and FlowDroid [3] is one of the more successful tools. Another Android analysis framework is DroidSafe [17], which is similar in approach, except that it benefits from deep levels of object sensitivity, rather than flow sensitivity, in its core static analysis. Another approach to information-flow analysis is via the use of program dependence graphs, as in Pidgin [30]. Program dependence graphs can be used to model any form of data and control dependence between instructions. By also detecting control dependencies, PDGs are a good choice if detection of implicit flow of information is important. In our earlier experiences while developing Ethainter we found that control-flow dependencies lead to higher levels of imprecision and hence we do not consider control-flow dependencies. DroidInfer [26] performs taint analysis using constraint flow graph reachability algorithms. Yet another approach involves the use of program slicing [45], with an efficient representation for multiple program slices with a lot of common nodes.

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 and models 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, possibly 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. We show the scalability of our framework by analyzing the entire blockchain in 3 hours, consisting of over 140K unique smart contracts with around 38MLoC, deployed over millions of accounts. Ethainter identifies security issues in contracts currently holding substantial Ether balances.

References

[1] [n. d.]. Contract Library. https://www.contract-library.com/ Accessed: 2018-11-17.

[2] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. ACM, New York, NY, USA, 66–77. https://doi.org/10.1145/3167084

[3] 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

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2016. *A survey of attacks on Ethereum smart contracts*. Technical Report. Cryptology ePrint Archive: Report 2016/1007, https://eprint.iacr.org/2016/1007.

[5] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. 2017. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact.

[6] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611

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

[8] 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

[9] 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. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html Accessed: 2018-10-31.

[10] Consensys. 2018. Consensys logo. https://new.consensys.net/ Accessed: 2018-04-17.

[11] Consensys. 2018. Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/ Accessed: 2018-04-17.

[12] 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.

[13] 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

[14] etherscan.io. [n. d.]. Ethereum Contracts with Verified Source Codes. https://etherscan.io/contractsVerified Accessed: 2018-03-15.

[15] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 125–128.

[16] FStarLang. 2018. F*: A Higher-Order Effectful Language Designed for Program Verification. https://www.fstar-lang.org/ Accessed: 2018-04-17.

[17] Michael I. Gordon, Deokhwan Kim, Jeff H. Perkins, Limei Gilham, Nguyen Nguyen, and Martin C. Rinard. 2015. Information Flow Analysis of Android Applications in DroidSafe. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*.

[18] 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

[19] 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

[20] 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.

[21] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 243–269.

[22] 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

[23] 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

[24] Everett Hildenbrandt, Xiaoran Zhu, and Nishant Rodrigues. 2017. KEVM: A Complete Semantics of the Ethereum Virtual Machine.

[25] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. 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, 520–535.

[26] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and Precise Taint Analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 106–117. https://doi.org/10.1145/2771783.2771803

[27] Marco Iansiti and Karim R. Lakhani. 2017. The Truth about Blockchain. *Harvard Business Review* 95 (Jan. 2017), 118–127. Issue 1.

[28] Neil Immerman. 1999. *Descriptive Complexity*. Springer. https://doi.org/10.1007/978-1-4612-0539-5

[29] Isabelle. 2018. Isabelle. https://isabelle.in.tum.de/ Accessed: 2018-04-17.

[30] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 291–302. https://doi.org/10.1145/2737924.2737957

[31] 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.

[32] K Framework. 2018. K Framework. http://www.kframework.org/index.php/Main_Page Accessed: 2018-04-17.

[33] 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)*.

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

[35] LLVM. 2018. The LLVM Compiler Infrastructure Project. https://llvm.org/ Accessed: 2018-04-17.

[36] 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

[37] 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)*.

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

[39] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* abs/1802.06038 (2018). arXiv:1802.06038 http://arxiv.org/abs/1802.06038

[40] 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

[41] Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. 2008. *User-input Dependence Analysis via Graph Reachability*. Technical Report. Mountain View, CA, USA.

[42] SeaHorn. 2018. SeaHorn | A Verification Framework. http://seahorn.github.io/ Accessed: 2018-04-17.

[43] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. *CoRR* abs/1702.05511 (2017). arXiv:1702.05511 http://arxiv.org/abs/1702.05511

[44] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 210–225. https://doi.org/10.1007/978-3-642-37057-1_15

[45] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

[46] Various. 2018. Documentation for the LLL compiler – LLL Compiler Documentation 0.1 documentation. http://lll-docs.readthedocs.io/en/latest/index.html Accessed: 2018-04-17.

[47] Various. 2018. GitHub - ethereum/serpent. https://github.com/ethereum/serpent Accessed: 2018-04-17.

[48] Various. 2018. GitHub - ethereum/solidity: The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity Accessed: 2018-04-17.

[49] Various. 2018. GitHub - ethereum/vyper: New experimental programming language. https://github.com/ethereum/vyper Accessed: 2018-04-17.

[50] Various. 2018. Vandal – A Static Analysis Framework for Ethereum Bytecode. https://github.com/usyd-blockchain/vandal/ Accessed: 2018-07-30.

[51] wired.com. 2018. 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/ Accessed: 2018-03-15.

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

[53] 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