Vandal: A Scalable Security Analysis Framework for Smart Contracts

Lexi Brent[*]
The University of Sydney
lexi.brent@sydney.edu.au

Anton Jurisevic[*]
The University of Sydney
ajur4521@uni.sydney.edu.au

Michael Kong[*]
The University of Sydney
mkon1090@uni.sydney.edu.au

Eric Liu[*]
The University of Sydney
eliu9480@uni.sydney.edu.au

François Gauthier
Oracle Labs, Australia
francois.gauthier@oracle.com

Vincent Gramoli
The University of Sydney
vincent.gramoli@sydney.edu.au

Ralph Holz
The University of Sydney
ralph.holz@sydney.edu.au

Bernhard Scholz
The University of Sydney
bernhard.scholz@sydney.edu.au

Abstract

The rise of modern blockchains has facilitated the emergence of smart contracts: autonomous programs that live and run on the blockchain. Smart contracts have seen a rapid climb to prominence, with applications predicted in law, business, commerce, and governance.

Smart contracts are commonly written in a high-level language such as Ethereum's Solidity, and translated to compact low-level bytecode for deployment on the blockchain. Once deployed, the bytecode is autonomously executed, usually by a virtual machine. As with all programs, smart contracts can be highly vulnerable to malicious attacks due to deficient programming methodologies, languages, and toolchains, including buggy compilers. At the same time, smart contracts are also high-value targets, often commanding large amounts of cryptocurrency. Hence, developers and auditors need security frameworks capable of analysing low-level bytecode to detect potential security vulnerabilities.

In this paper, we present Vandal: a security analysis framework for Ethereum smart contracts. Vandal consists of an analysis pipeline that converts low-level Ethereum Virtual Machine (EVM) bytecode to semantic logic relations. Users of the framework can express security analyses in a declarative fashion: a security analysis is expressed in a logic specification written in the Soufflé language. We conduct a large-scale empirical study for a set of common smart contract security vulnerabilities, and show the effectiveness and efficiency of Vandal. Vandal is both fast and robust, successfully analysing over 95% of all 141k unique contracts with an average runtime of 4.15 seconds; outperforming the current state of the art tools—Oyente, EthIR, Mythril, and Rattle—under equivalent conditions.

[*]Joint first authorship.

1 Introduction

Since the introduction of the Bitcoin cryptocurrency in 2008 [1], blockchain technology has seen growing interest from economists, lawyers, the technology industry and governments [2–5]. Blockchains are decentralized distributed public ledgers, and have recently been used as Turing-complete computational devices for storing and executing autonomous programs called smart contracts. Ethereum [6] and Cardano [7] are two examples of blockchains with smart contract capabilities. Ethereum has become the *de facto* standard platform for smart contract development, with a market capitalization of $20B USD [8]. For this reason, we focus exclusively on Ethereum smart contracts.

Smart contracts are typically written in a high-level language such as Solidity [9], compiled to a low-level bytecode, and deployed on the blockchain. Smart-contracts have unique addresses that are used to identify them on the blockchain. Smart-contracts can then be invoked by users of the network or other smart contracts, and are executed by the Ethereum Virtual Machine (EVM). Each smart contract commands its own balance of Ether, the cryptocurrency used in Ethereum.

Once deployed on the blockchain, a contract's bytecode becomes immutable. This is a high-risk, high-stakes paradigm: deployed code is impossible to patch, and contracts collectively control billions of USD worth of Ether. As a consequence, security bugs in smart contracts can have disastrous consequences. A well-known example is the 2016 attack on a smart contract known as the "The DAO", where an attacker exploited a reentrancy vulnerability, gaining control of 3.6M Ether, worth more than $50M USD at the time [10, 11]. One year later, in July 2017, an attacker exploited a vulnerability in a library contract used by the "Parity multisig wallet", stealing 150k Ether worth $30M USD [12–14]. Later that year, in November 2017, an attacker exploited another vulnerability in a newer version of the Parity multisig wallet contract, permanently freezing over 500k Ether worth $155M USD [15]. In January 2018, an attacker exploited an integer overflow vulnerability in the contract underpinning the "Proof of Weak Hands" contract, making off with 866 Ether worth $2.3M USD [16].

A wide range of known security vulnerabilities have been described [17–23]. So-called *unchecked send* vulnerabilities arise when the success/failure return value of a message call operation is not checked, *i.e.,* always assuming success. This vulnerability is one example of a larger class of vulnerabilities arising from mishandled exceptions, and occurs surprisingly frequently [17, 18, 24]. *Reentrancy* vulnerabilities emerge when a contract is not programmed with reentrancy in mind, allowing an attacker to make reentrant message calls that exploit an intermediate state [17, 18, 20]. *Unsecured balance* vulnerabilities occur when a contract's balance is exposed to theft by an arbitrary caller [21]. This may be due to programmer error; for example, a misnamed constructor function that inadvertently becomes a public function [17]. *Destroyable contracts* are those which contain a SELFDESTRUCT instruction on an exposed program path, *i.e.,* without adequate authentication, allowing an arbitrary caller to permanently destroy the contract [17, 21]. *Origin vulnerabilities* occur when a contract performs authentication by checking the return value of ORIGIN rather than CALLER [18, 22]. In EVM, ORIGIN returns the address of the account that initiated the transaction, whereas CALLER returns the address of the account or contract that initiated the currently executing message call. An ORIGIN

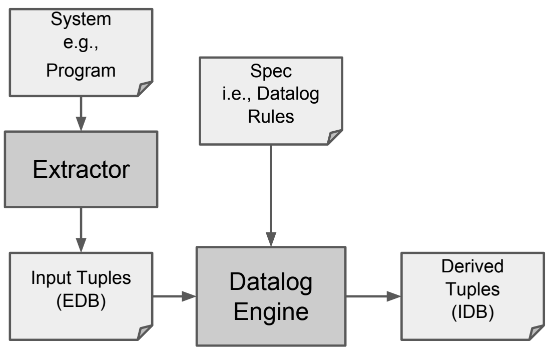


Figure 1: Logic-Driven Program Analysis Approach: a program is converted by an "extractor" to a relational format also known as an Extensional Database (EDB). The logic rules express the program analysis. A Datalog engine computes the result of the program analysis from the EDB and the set of rules. This result is also known as an Intensional Database (IDB), and contains intermediate and final results of the analysis.

vulnerability can be exploited if control is passed to a malicious contract that makes a message call to the vulnerable contract: the vulnerable contract checks ORIGIN and sees the transaction initiator's address, rather than the malicious contract's address. Other vulnerability classes include those related to gas consumption [17, 23], locking away of Ether [21], making calls to dead or unknown contracts [17, 21], timestamp dependence [19, 20], transaction-ordering dependence [19, 20], integer overflow/underflow [19], block state dependence [19], Ether lost in transfer [17], and EVM's stack size limit [17].

Security vulnerabilities in smart contracts stem from a wide range of issues including programmer error, language design issues, and toolchain bugs such as those in the Solidity compiler [25].

To analyze and alleviate the current deficiencies in smart contract programming methodologies, language design, and toolchains, we present Vandal: a new security analysis framework capable of directly analyzing smart contract bytecode[1]. Vandal facilitates a logic-driven static program analysis approach (cf. Chapter 12.3 of [26], and [27]) as depicted in Figure 1. In this approach, Datalog is used as a domain-specific language to bridge the gap between program semantics of security vulnerabilities and the implementation of the corresponding analyses.

In Vandal, security analysis problems are specified declaratively using logic rules. An off-the-shelf Datalog engine then executes the specification for a set of input relations that encode the contract (also known as the extensional database) and produces an output relation containing a list of detected security vulnerabilities and their locations in the bytecode. Our logic-driven approach results in security analyzers that are easy to write, maintain, and less error-prone compared to low-level, hand-crafted implementations. More importantly, our approach allows users to explore the design space of security analyzers with-

[1]Available online: `https://github.com/usyd-blockchain/vandal`

out embarking on the difficult endeavour of writing/modifying a hand-crafted static analyzer. This paradigm is supported by a cornucopia of state-of-the-art Datalog engines that specifically target static program analysis [27–29].

Vandal consists of two parts. First, an analysis pipeline translates low-level bytecode to logic relations for the logic-driven security analysis. This pipeline is represented by the "extractor" component of Figure 1. In Vandal, the logic relations expose data- and control-flow dependencies of the bytecode. The second component is a set of logic specifications for security analysis problems. Vandal uses Soufflé [30] as a Datalog engine, which synthesizes highly performant C++ code from logic specifications.

The contributions of our work are:

- The Vandal security analysis framework that transforms low-level EVM bytecode to logic relations, enabling a logic-driven approach for expressing security analyzers. Vandal's analysis pipeline consists of a bytecode scraper that retrieves EVM bytecode from the blockchain, a disassembler that translates bytecode into opcodes, a decompiler that translates low-level bytecode to register transfer language, and an extractor that translates this register transfer language into logic semantic relations.
- A new decompilation technique for incrementally reconstructing control-flow, by applying symbolic execution for basic-blocks, incremental data-flow analysis, and node-splitting techniques to disambiguate jump targets. The precise detection of jump targets is essential for identifying stack locations statically so that the stack location can be assigned a register.
- A static analysis library containing logic specifications that expose control-flow graph properties, domain-specific properties of EVM operations, and data and control dependencies.
- A case study in which we phrase common security analyses for smart contracts as logic specifications within Vandal. We implement analyses for unchecked send, reentrancy, unsecured balance, destroyable contract, and use of ORIGIN vulnerabilities in Vandal.
- A large-scale empirical analysis of all 141k unique smart contracts scraped from the public blockchain. We demonstrate that Vandal is more robust and efficient than Oyente [20], EthIR [31], Mythril [32], and Rattle [33], successfully decompiling over 95% of all contracts with an average runtime of 4.15 seconds.

The organization of this technical report is as follows: In Section 2, we introduce the stages of the Vandal analysis pipeline and their implementation. In Section 3, we provide a use-case study showing how security analyses for common vulnerabilities can be implemented in Vandal as logic specifications. In Section 4, we present and discuss results from our empirical analysis experiments. Related work is surveyed in Section 5. Finally, in Section 6 we draw our conclusions.

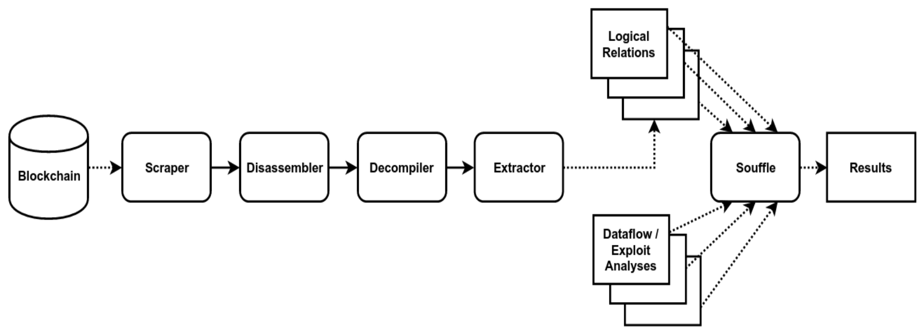


Figure 2: Analysis Pipeline. The **Scraper:** extracts smart contract bytecode in bulk from the blockchain. The **Disassembler** converts the bytecode into mnemonics. The **Decompiler** translates the stack-based bytecode to a register transfer language. The **Extractor** produces logic relations from the register transfer language, reflecting the program semantics of the smart contract. The security analyses that report potential security vulnerabilities in the smart contracts are written in Soufflé [30].

2 The Vandal Framework

Vandal has been designed for phrasing security vulnerability analyses in a declarative language called Soufflé [30]. Expressing vulnerability analyses in a declarative language allows security experts to rapidly prototype new analyses and compose existing analyses with ease [26, 27, 34]. The logic-specification driven approach has several advantages, including this agile capability for designing and implementing security analyses. The design space for security analyses can be explored in terms of a precision vs. time trade-off. A traditional implementation of a security analysis would require several hundreds of thousands of lines of code, becoming very costly to implement, test, and maintain. The performance penalties incurred by the usage of a logic language for expressing security analyses is alleviated by the presence of modern logic synthesizers such as Soufflé that produce highly performant C++ code from logic specifications. The produced C++ code is equivalent or better in performance than hand-written code for program analysis [30].

The challenge in the design of Vandal is the translation of smart contracts into logic relations. The Vandal framework utilizes an analysis pipeline to convert Ethereum bytecode stored on the blockchain to logic relations that reflect the semantics of the smart contract. This is challenging: EVM bytecode is executed by a very low-level stack-based abstract machine. In contrast to Java virtual machine bytecode, low-level EVM bytecode has no abstractions for classes and methods, no memory management, no type checking, no class loading, and no notion of stack frames for function calls. The EVM is a very simplistic computational device created for efficient storage of smart contracts on a blockchain. The control- and data-flows of a smart contract are obfuscated by the virtual machine stack, making the task of static program analysis immensely difficult. To apply a logic-driven approach for analyzing smart contracts, the EVM bytecode must be transformed to a new program representation that reconstructs the data- and control- flow dependencies of the original high-level language.

For the reconstruction of program semantics from Ethereum's low-level bytecode, we introduce an analysis pipeline as shown in Figure 2. This pipeline breaks up the task of translating bytecode to an analyzable form into several stages, as follows: The Ethereum bytecode is scraped from the blockchain by the *Scraper*, disassembled by the *Disassembler*, then decompiled into a register transfer language by the *Decompiler*. The decompiler reconstructs the control- and data-flow of the high-level language of the smart contract. After decompilation, the bytecode program is expressed as a register-transfer language. The *Extractor* converts the intermediate representation of the register-transfer language to logic relations. These logic relations capture the semantics of the smart contracts, and are stored in a simple tab-separated value format. Soufflé synthesizes the security analysis to executable programs that read the logic relations generated by the Extractor, and perform the security analysis for detecting vulnerabilities. Potential security vulnerabilities are reported after the execution of the security analysis.

In the following subsections we describe the design and implementation of each pipeline stage. Note that the decompiler requires substantial research and implementation effort due to the incremental discovery of jump targets. It is an intertwine approach of jump discovery, node-splitting, and incremental data-flow analysis. With the exception of the security analyses and program analysis library, Vandal's infrastructure is written in Python. The Vandal framework is open source and has been published under the BSD license [35].

2.1 Scraper

The first stage of our analysis pipeline is the scraper, which extracts the bytecode representation of smart contracts from the live Ethereum blockchain.

We implement the scraper using the JSON-RPC API of the Parity [36] Ethereum client. For scraping all transactions, including "internal" transactions, the Parity client must be synchronized with command-line flags that enable tracing and disable pruning, as shown in Listing 1.

```
% parity --pruning archive --tracing on --no-warp
```

Listing 1: Parity command-line flags to enable tracing and disable pruning

The scraping procedure is outlined in Algorithm 1. The procedure traverses through all transactions in all blocks, searching for contract creations. The following helper functions are assumed to exist:

- GET-CODE(contract) — retrieves bytecode stored at the specified *contract* address from the Parity client.
- TRACE-TRANSACTION(transaction) — retrieves the execution trace of the specified *transaction* from the Parity client.
- SAVE-CONTRACT(transaction, contract, code) — writes the given *contract*'s *code* to disk, along with *transaction* metadata.

Scraping the whole blockchain is inefficient due to its size when synchronized with full tracing enabled (> 1.5TB), Parity's underlying hash-based database, and the overheads imposed by JSON-RPC HTTP requests. Parity's use of a

Algorithm 1 Ethereum blockchain smart contract scraping procedure.

```
Require: start block number $s$ and end block number $e$
Require: function Get-Code(contract)
Require: function Trace-Transaction(transaction)
Require: function Save-Contract(transaction, contract, code)
 1: for each block $b$ such that $s <= b < e$ do
 2:     for each transaction $t$ in block $b$ do
 3:         if $t$ creates a contract at address $c$ then
 4:             $code$=Get-Code($c$)
 5:             Save-Contract($t$, $c$, $code$)
 6:         else
 7:             $traces \leftarrow$ Trace-Transaction($t$)
 8:             for each $trace$ in $traces$ do
 9:                 if $trace$ created a contract at address $c$ then
10:                     $code \leftarrow$ Get-Code($c$)
11:                     Save-Contract($t$, $c$, $code$)
12:                 end if
13:             end for
14:         end if
15:     end for
16: end for
```

hash-based key-value store, combined with a sequential scraping process, results in slow random disk reads. Our scraper uses various implementation techniques to reduce the runtime from several days to less than a day:

1. parallelization — splitting the blockchain into contiguous chunks and scraping each portion in a separate instance;
2. increasing the number of JSON-RPC threads used by Parity (using the `--jsonrpc-threads` and `--jsonrpc-server-threads` command-line flags); and
3. performing our scrapes with all I/O via a main-memory ramdisk (using a machine with 512GB of main memory).

For each contract on the blockchain, our scraper produces a file on the local filesystem containing the machine code of the smart contract, i.e., a sequence of bytes that requires disassembly and decompilation in the later stages of our pipeline. Due to legal reasons, the scraper has not been released as open source. We will query the legal status of the scraper at a later point in time so that it can be added to Vandal's open source repository on GitHub [35].

2.2 Disassembler

The second stage of the analysis pipeline is the disassembler, which converts EVM bytecode to a series of readable low-level mnemonics annotated with program counter addresses. This conversion is performed by a single linear scan over the bytecode, converting each instruction to its corresponding mnemonic and incrementing a program counter for each instruction and each inline operand. The result is a sequence of program addresses, mnemonics and their arguments.

```
contract Factorial {
  function fact(uint x) returns (uint y) {
    if (x == 0) {
      return 1;
    } else {
      return fact(x-1) * x;
    }
  }
}
```

Figure 3: Example: high-level Solidity code.

```
60606040526000357
c0100000000000000
00000000000000000
00000000000000000
00000000900480631
93ddd2c1460375760
35565b005b6042600
4805050605a565b60
40518082151581526
02001915050604051
80910390f35b60006
00560006000505414
9050606b565b9056

```

(a) Bytecode

```
0x00 PUSH1  0x60
0x02 PUSH1  0x40
0x04 MSTORE
0x05 PUSH1  0xe0
0x07 PUSH1  0x02
0x09 EXP
0x0a PUSH1  0x00
0x0c CALLDATALOAD
0x0d DIV
0x0e PUSH4  0x193ddd2c
0x13 DUP2
0x14 EQ
0x15 PUSH1  0x1a
0x17 JUMPI
```

```
0x18 JUMPDEST
0x19 STOP
0x1a JUMPDEST
0x1b PUSH1  0x00
0x1d SLOAD
0x1e PUSH1  0x05
0x20 EQ
0x21 PUSH1  0x60
0x23 SWAP1
0x24 DUP2
0x25 MSTORE
0x26 PUSH1  0x20
0x28 SWAP1
0x29 RETURN
```

(b) Disassembled bytecode

Figure 4: Disassembler: converts a stream of EVM bytecode to a list of addresses/mnemonic pairs.

Assume that a programmer deploys a Solidity smart contract as shown in Figure 3 on the blockchain. Vandal's scraper will capture the machine code representation of this smart contract in the form of EVM bytecode, producing a file such as the one illustrated in Figure 4(a). The disassembler converts the machine code to readable EVM bytecode as shown in Figure 4(b).

The Vandal disassembler produces a similar output format to the Ethereum Foundation's official disassembler, but with support for basic block boundary delineation. The disassembler and decompiler share a common bytecode parsing implementation. An interface for declaring new instructions is provided in case changes occur to the EVM specification.

2.3 Decompiler

The next stage in the analysis pipeline is the decompiler, which translates the low-level EVM bytecode to a register transfer language. This register transfer language exposes data- and control-flow structures of the bytecode. Conceptually the semantics of the newly defined language has a strong overlap with those of EVM, i.e., the instructions of the EVM are still reflected in the register transfer language of the decompiler. However, the notion of a stack is replaced by a set of registers, and all instructions operate on registers. An abstract syntax of the language is given in Listing 2 below.

```
<operation> ::= Register = <rhs> | Op <args>
<rhs>       ::= Register | Constant | Op <args>
<args>       ::= (Register | Constant ) *
```

Listing 2: Syntax of the intermediate register transfer language.

The language has two types of operations: operations that are side-effect free or have a side-effect. An operation may manipulate values in registers, memory, or storage. The right-hand side of an operation can be either a constant, another register, or an operation that requires zero or more arguments. The left-hand side is a register, a memory/storage location if the operation has a side-effect. All stack-based operations of the EVM can be expressed in this register language. Note that the the storage structure persists on the blockchain, whereas memory exists only transiently at runtime.

```
0x0: V0 = 0x60
0x2: V1 = 0x40
0x4: M[0x40] = 0x60
0x5: V2 = 0x10...0
0x23: V3 = 0x0
0x25: V4 = CALLDATALOAD 0
     x0
0x26: V5 = DIV V4 0x10
     ...0
0x27: V6 = 0xb95d228
0x2d: V7 = EQ V5 0
     xb95d228
0x2e: V8 = 0x33
0x30: JUMPI 0x33 V7

0x31: JUMPDEST
0x32: STOP

0x33: JUMPDEST
0x34: V9 = 0x4a
0x36: V10 = 0x4
0x38: V11 = CALLDATALOAD
     0x4

0x39: JUMPDEST
0x3a: V121 = 0x0
0x3d: V13 = 0x0
0x3f: V14 = EQ 0x0 S0
0x40: V15 = ISZERO V14
0x41: V16 = 0x65
0x43: JUMPI 0x65 V15

0x45: V17 = 0x1
0x47: V18 = 0x60
0x49: JUMP 0x60

0x4a: JUMPDEST
0x4b: V19 = 0x40
0x4e: V20 = M[0x40]
0x51: M[V20] = S0
0x52: V21 = M[0x40]
0x56: V22 = SUB V20 V21
0x57: V23 = 0x20
0x59: V24 = ADD 0x20 V22
0x5b: RETURN V21 V24

0x5c: JUMPDEST
0x5d: V25 = MUL S0 S1

0x60: JUMPDEST
0x64: JUMP {0x4a, 0x5c}

0x65: JUMPDEST
0x67: V26 = 0x5c
0x69: V27 = 0x1
0x6c: V28 = SUB S1 0x1
0x6d: V29 = 0x39
0x6f: JUMP 0x39
```

Figure 5: Example: register transfer language.

For example, consider the code in Figure 5 that was decompiled from the EVM bytecode of Figure 4. The first EVM instruction `0x00 PUSH1 0x60` pushes the constant value `0x60` onto the EVM stack, as shown in the example of Figure 4. In Figure 5, the decompiler replaces this EVM instruction with the register transfer instruction `0x0: V0 = 0x60`. The output is generated such that instruction opcodes generally correspond directly to EVM instructions excluding stack operations. The instructions have a program address followed by a colon. Note that `M[x]` represents a memory location `x`. Registers are denoted `Vn` where `n` is a number. Register values can be read from or written to, however, our register transfer language does not permit indirect access to registers. An operation may have several arguments that are separated by spaces (e.g. `EQ V5 0xb95d228`). Due to limitations of the decompilation, we may not be able to determine jump addresses uniquely and we use curly braces to denote sets of possible jump addresses — e.g. `JUMP {0x4a, 0x5c}` For the translation work it is important to realize that the decompiler does not produce *executable* register transfer language code due to the indeterminism caused by unknown jump addresses. The main purpose of the translation is the detection of security vulnerabilities and hence the indeterminism can be seen as a precursor to the program analysis expressed in logic.

To achieve the translation, the decompiler assigns to stack positions to registers. However, the control-flow of a smart contract is obfuscated by stack-based data-dependencies. Hence, the decompiler must comprehend the contract's stack access patterns, necessitating a deep program analysis.

The control-flow of an EVM bytecode program is initially unknown, requiring an incremental analysis to iteratively construct the flow and determine the stack locations. To get a handle on the problem, we incrementally build a control flow graph (CFG) [26] and statically propagate constant values for to identify potential jump addresses. Note that in EVM bytecode, all valid jumps must jump to a JUMPDEST instruction. Using this property, we can easily slice the disassembled bytecode into basic blocks. However, determining the destination of a jump operation is difficult since the EVM jump instructions do not have explicit arguments, and instead pop their operands from EVM's stack at runtime. These dependencies may be distinctly non-local, in that a value may be used only after following a number of jumps. So in order to resolve these dependencies, jump destinations must have been resolved beforehand. Yet these destination values may themselves have been defined non-locally. So the bytecode, even when disassembled into a sequence of more-readable opcodes, has very little structure.

We use two phases to build the control-flow structure:

- In the first phase, we determine the basic blocks of the smart contract and assume a symbolic stack with symbolic values. We symbolically execute each basic block and de-stackify its operations. Some registers may point to stack locations whose values were produced by the prior basic blocks. Other registers may point to stack locations that were produced by the current basic block. We introduce symbolic labels for stack locations that are used to express data-dependencies across basic blocks, and try to resolve them via registers in the second phase. For example, an ADD EVM instruction that is placed as the first instruction of a basic block would generate a new register transfer operation `V = ADD S0 S1`, where `S0` and `S1` are the top symbolic stack elements which were written by an ancestral block, and `V` is a new register holding the result of the addition, which is pushed to the symbolic stack.

- The second phase builds the control-flow graph incrementally. It resolving symbolic stack locations from the first phase, and new jump addresses emerge as a side effect. These new jump addresses, in turn, may lead to further addresses being discovered during the next iteration. This second phase is expressed as a fixed-point algorithm that propagates constant values that may carry jump addresses.

After decompilation, most jump addresses or sets of potential jump addresses for basic blocks have been determined. For example, the corresponding control flow graph of Figure 5 is shown in Figure 6.

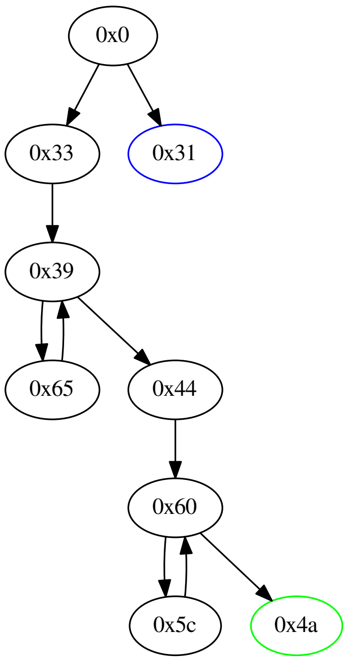
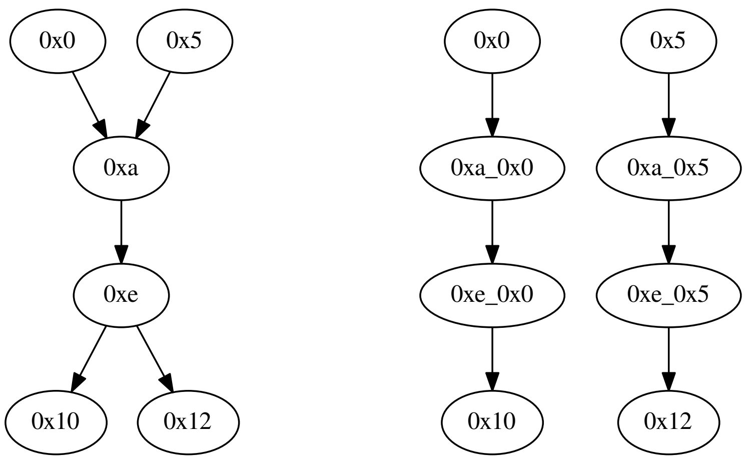


Figure 6: Control flow graph for the FACTORIAL contract, as output by Vandal.

In order to obtain a precise dataflow analysis and resolve as many jump destinations as possible, we implemented a node splitting technique in which CFG nodes with multiple outgoing edges are split into several paths, each originating at at a distinct common ancestor. For example, consider the CFG in Figure 7a, and suppose that the jump address used in block 0xE is pushed in 0xA's predecessors. In this case, we have a set of two possible values {0x10, 0x12} — one value from each of 0xA's predecessors. However, if we split node 0xE by cloning the path from 0xE to 0xA for each of 0xA's predecessors, we can resolve the jump address at 0xE to a constant value for each path. This information can then be propagated down the graph during data flow analysis and used to determine values in deeper nodes more precisely. The result of splitting at node 0xE is shown in Figure 7b.



(a) Before node splitting. (b) After splitting at node 0xE.

Figure 7: Demonstration of node splitting within Vandal.

After data flow analysis is complete, we merge duplicated nodes back together, so that our final output is consistent with the input program, and does not contain duplicated program counters, for instance.

2.4 Extractor

The next step in the analysis pipeline is the extractor that translates the register transfer language to logic relations. Logic relations are comma-separated files that can be later read by the security analyses expressed in Soufflé [30]. The logic relations express the register transfer language of an EVM bytecode program.

The logic relations use various domains such as the set of statements S, and the set of registers V in a smart contract. There are other domains such as O for the set of operations in the EVM. Edges of the control-flow graph are expressed by the relation $\mathit{edge} \subseteq S \times S$. The first statement in the control-flow graph is given by the singleton set $\mathit{entry} \subseteq S$. For tracking data flow of smart-contracts, we introduce def-use sets [26] that track where registers are written to and read from, i.e., relation $\mathit{def} \subseteq V \times S$ enumerates all write positions of a register, and relation $\mathit{use} \subseteq V \times S$ enumerates all read positions of a register in a smart-contract. The relation $\mathit{op} \subseteq S \times O$ associates the op-code to a statement. The relation $\mathit{value} \subseteq V \times \mathbb{N}$ associates constant values to registers if the set of potential constant values for the register is known. The logic relations are defined in Soufflé as shown in Listing 3.

We have other relations that pre-compute dominance and post-dominance relations outside of Datalog such that control-dependencies in smart contracts can be expressed in few lines of logic. We have specific logic relations for memory, storage, and jump operation to reflect the semantics of the smart contract in more detail. These relations are required for the detection of some security analyses that are presented in the next section.

```
.type Statement
.type Variable
.type Opcode
.type Value

.decl entry(s:Statement)
.decl edge(h:Statement, t:Statement)
.decl def(var:Variable, stmt:Statement)
.decl use(var:Variable, stmt:Statement, i:number)
.decl op(stmt:Statement, op:Opcode)
.decl value(var:Variable, val:Value)
.input entry, edge, def, use, op, value
```

Listing 3: Definition of logic relations in Soufflé.

3 Use-Case Study: Security Analyses in Vandal

To demonstrate the user experience for defining vulnerability analyses, we describe some common vulnerabilities and how they can be implemented within Vandal using Souffle Datalog code. For simplicity, our explanations of each vulnerability use snippets of Solidity source code.

3.1 Vulnerability: Unchecked Send

Explanation. Solidity does not have exceptions that may be caught, so some low-level operations report a success/failure status via a boolean return value. One such operation is the `address.send()` method, which performs a message call and transfers Ether to a specified address. If this operation fails, then any execution that occurred up until the point of failure is rolled back, but *execution of the calling function continues as normal.* Hence, failing to check for and handle an error state can lead to unintended effects, and is therefore considered to be dangerous behaviour.

Consider the Solidity function in Listing 4 which transfers 100 wei to a creditor using `creditor.send`, provided they have not already been paid:

```
function pay() {
  require(!paid);
  creditor.send(100);
  paid = true;
}
```

Listing 4: Vulnerable – return value of send is not checked

```
function pay() {
  require(!paid);
  require(creditor.send(100));
  paid = true;
}
```

Listing 5: Safe – return value of send is checked and failure handled

Assume that the account at the creditor address is itself a smart contract, and its execution throws an exception. In this case, the funds will not be transferred and the called function will fail. However, the state of the contract is still updated as if the creditor had been paid as the next operation in `pay()` sets `paid` to true. The state of the contract has fallen out of sync with the reality that it is supposed to represent. The creditor will no longer be able to obtain what they are owed as the funds are now locked in the contract.

The correct implementation of this `pay()` must check the return code of `creditor.send(100)`. If the return code indicates failure, it must not update the contract state, e.g. by throwing an exception as shown in Listing 5.

Implementation. On the bytecode level, `address.send` corresponds directly to EVM's CALL instruction An analysis implementation for unchecked send is shown in Listing 6. Here, we check that the return value `u` of a call operation (line 4) is neither used to control the execution of a `throw` (line 5), nor an update to persistent storage (line 6). In other words, if no tangible action is taken based on the return value of CALL, then the operation is flagged as vulnerable.

```
1 .decl uncheckedCall(u:Statement)
2
3 uncheckedCall(u) :-
4   callResult(_, u),
5   !checkedCallThrows(u),
6   !checkedCallStateUpdate(u).
```

Listing 6: Analysis implementation in Vandal for unchecked send.

The relations `callResult`, `checkedCallThrows`, and `checkedCallStateUpdate` are all provided by Vandal's static analysis library.

3.2 Vulnerability: Reentrancy

Explanation. Another common mistake that can be much harder to spot is caused by *reentrancy*. When an Ethereum contract performs a message-call to another contract by sending value or calling a function on the other contract, the recipient contract may perform an arbitrary execution before returning control to the caller. The recipient contract may, for example, call another contract, or even the original contract. If the original contract is not reentrancy-safe, i.e., guarantees that contract state is always correct independent of reentering calls, then a malicious contract can make reentrant calls that take advantage of intermediate state. In 2016, a reentrancy vulnerability was infamously exploited by an unknown attacker to "steal" then-equivalent of over $50M USD from a contract known as *TheDAO* [10, 11].

In Listing 7 we show an excerpt from a contract that stores Ether for several users and keeps track of how much Ether is owned by each user in an `accounts` map. The contract allows users to withdraw their share of Ether by calling `withdraw()`, which reads the caller's balance from `accounts`, sends the Ether, and sets the new account balance to 0. However, if the caller is a contract, it can make a reentrant call to `withdraw()` when control is passed to it by the message call on line 3. At that point, the `accounts` map has not yet been updated, so the caller's balance would be sent again. Recursive reentrant calls of this nature would allow an attacker to drain all Ether from the contract. Note, however, that recursion is not a requirement: depending on the code, a single reentrancy could be just as destructive.

```
1 function withdraw() public {
2   uint balance = accounts[msg.sender];
3   msg.sender.call.value(balance)();
4   accounts[msg.sender] = 0;
5 }
```

Listing 7: Vulnerable – `accounts` is updated after message call

```
1 function withdraw() public {
2   uint balance = accounts[msg.sender];
3   accounts[msg.sender] = 0;
4   msg.sender.call.value(balance)();
5 }
```

Listing 8: Safe – all state is updated before message call

There is no way to prevent reentrant calls in Ethereum. Hence, all functions must be reentrancy-safe. This can be achieved by using `send()` instead of `call.value()()`, which does not forward enough gas for the callee to perform computations. Alternatively, ensuring that all state changes occur before any external calls would prevent an attacker from taking advantage of intermediate state. Our revised function in Listing 8 demonstrates the latter approach, by simply swapping lines 3 and 4 of the vulnerable code.

```
function mutexProtected() {
  require(!mutex);
  mutex = true;
  <protected CALL code>
  mutex = false;
}
```

Listing 9: Example of a non-reentrant CALL protected by a mutex.

Implementation. At the bytecode level, we will consider any CALL operation to be reentrant if it can be reached in a recursive call to the enclosing contract. This simple definition can be generalised to a mapping from specific CALL operations to program points they reach in a reentrant manner. We also require that it forwards any remaining gas on to the callee, so that there is sufficient gas for further execution to take place. Finally, we require that reentrant statements are not protected by a mutex-like structure. For example, Listing 9 is not reentrant, because the update of the `mutex` field is carried through to recursive calls.

```
.decl reentrantCall(stmt: Statement)

reentrantCall(stmt) :-
  op(stmt, "CALL"),
  !protectedByLoc(stmt, _),
  gassy(stmt, gasVar),
  op_CALL(stmt, gasVar, _, _, _, _, _, _).
```

Listing 10: Analysis implementation in Vandal for reentrancy.

An analysis for reentrancy vulnerabilities can be implemented in Vandal as shown in Listing 10. Here, `protectedByLoc` is defined by Vandal, and `protectedByLoc(stmt, _)` means that statement `stmt` is protected by a mutex. The `gassy` relation is also defined by Vandal, with `gassy(stmt, var)` implying that statement `stmt` uses a variable `var` that depends on the result of a GAS operation. In other words, a

CALL is flagged as reentrant if it forwards sufficient gas and is not protected by a mutex.

3.3 Vulnerability: Unsecured Balance

Explanation. Unsecured balance vulnerabilities arise when a contract's balance is left exposed to theft. This can arise due to programmer error: for example, in Solidity, a contract's constructor function is not qualified with a dedicated keyword and giving it a name that differs from the contract's name turns it into a public function. Since it is a common pattern in contract constructors to set a state variable to define the "contract owner", i.e., an address that can perform privileged actions, a misnamed constructor often allows any caller to assume ownership and access to privileged functionality. Such a vulnerability was discovered in the wild in the "Rubixi" Ponzi scheme contract [17]. The Rubixi developers had renamed their contract at some point, but failed to rename the constructor, allowing anyone to assume ownership and withdraw Ether intended for the contract owner.

The contract in Listing 11 contains a misnamed constructor called `TaxOffice()`, which becomes a public function instead of a constructor. Anyone could call `TaxOffice()` to take ownership of the contract and then, by calling `collectTaxes()`, withdraw Ether intended for the contract's rightful owner.

```
contract TaxMan {
  address private owner;
  ...
  function TaxOffice() {
    owner = msg.sender;
  }
  function collectTaxes() public {
    require(msg.sender == owner);
    owner.send(tax);
  }
}
```

Listing 11: Vulnerable – misnamed constructor `TaxOffice()` sets `owner`

```
contract TaxMan {
  address private owner = msg.sender;
  ...
  function collectTaxes() public {
    require(msg.sender == owner);
    owner.send(tax);
  }
}
```

Listing 12: Safe – `owner` is initialised directly during contract creation

In Listing 12, the constructor function is eliminated altogether in favour of direct initialization of the `owner` state variable. Here, `owner` is initialised when the contract is first created and is never updated from within a function.

Implementation. We say that a contract is vulnerable if an arbitrary caller can force it to transfer Ether, and can manipulate the address to which that Ether is transferred.

Our example analysis implementation is shown in Listing 13. At the EVM level, Ether transfers are handled by the CALL instruction. Hence, this specification checks for a CALL instruction that:

1. has a destination address that can be manipulated (line 5); and
2. transfers a nonzero amount of Ether (lines 5-7); and

```
1 .decl unsecuredValueSend(stmt:Statement)
2
3 unsecuredValueSend(stmt) :-
4   op_CALL(stmt, _, target, val, _, _, _, _),
5   nonConstManipulable(target),
6   def(val, _),
7   !value(val, "0x0"),
8   !fromCallValue(val),
9   !inaccessible(stmt).
```

Listing 13: Analysis implementation in Vandal for unsecured balance.

3. does not simply forward the Ether from in the incoming message call (line 8); and
4. can be executed by an arbitrary caller (line 9).

The relations `fromCallValue(var)` and `inaccessible(stmt)` are provided by Vandal's static analysis library, with `inaccessible(stmt)` meaning that the operation at `stmt` is either unreachable code, or its execution is conditionally dependent on a value that can not be manipulated by an arbitrary caller.

3.4 Vulnerability: Destroyable Contract

Explanation. An EVM instructions exists that is capable of nullifying the bytecode of a deployed contract: SELFDESTRUCT. When called, it halts EVM execution, deletes the contract's bytecode, and sends any remaining value to a designated address. In a pattern that is similar to exposed ownership control, this can be exploited if this function is accessible for non-authorized callers.

Listing 14 shows such an accessible SELFDESTRUCT vulnerability. The public function `destroy()` should probably only be called by authorized callers, but performs no authentication before calling `selfdestruct()`.

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

Listing 14: Vulnerable – code path to `selfdestruct()` is not protected

```
function destroy() public {
  require(msg.sender == owner);
  selfdestruct(msg.sender);
}
```

Listing 15: Safe – `selfdestruct()` may only be executed by the contract owner

Listing 15 shows how easily this can be mitigated by requiring that the caller of `destroy()` be the contract's owner.

Implementation. We implement this in Vandal as shown in Listing 16 by requiring that a SELFDESTRUCT instruction exists (line 5), and is directly reachable from the contract's entry point with its execution not conditional upon a value outside the attacker's control (line 4). The `inaccessible` relation is defined by Vandal.

```
1 .decl destroyable(stmt:Statement)
2
3 destroyable(stmt) :-
4   !inaccessible(stmt),
5   op(stmt, "SELFDESTRUCT").
```

Listing 16: Analysis implementation in Vandal for accessible SELFDESTRUCT.

Any code path leading to a SELFDESTRUCT instruction, that can be executed by an arbitrary caller, is almost certainly unintended behaviour and is flagged by this implementation.

3.5 Vulnerability: Use of `ORIGIN`

Explanation. Solidity's low-level `tx.origin` contains the address of the account that made the *first* message call in the currently executing transaction. Hence, its use should be avoided when authenticating the sender of a message call, otherwise a malicious intermediary contract could perform authenticated message calls. Instead, programmers should use `msg.sender`, which contains the address that created the currently executing message call.

```
function protected() public {
  require(tx.origin == self.admin);
  // do something sensitive
}
```

Listing 17: Vulnerable – authentication mechanism checks `tx.origin`.

```
function protected() public {
  require(msg.sender == self.admin);
  // do something sensitive
}
```

Listing 18: Safe – authentication mechanism checks `msg.sender`.

A simple example of a broken authentication mechanism is shown in Listing 17, with a correct version shown in Listing 18.

Implementation. In bytecode form, `tx.origin` corresponds to EVM's ORIGIN instruction, and `msg.sender` to CALLER. We can implement an analysis in Vandal by checking that an ORIGIN instruction exists, and that its result is used in some other potentially sensitive operation such as a conditional or a write to storage. This is demonstrated in Listing 19.

```
.decl originUsed(stmt:Statement)

originUsed(stmt) :-
  op(stmt, "ORIGIN"),
  def(originVar, stmt),
  depends(useVar, originVar),
  usedInStateOrCond(useVar, _).
```

Listing 19: Analysis implementation in Vandal for use of ORIGIN.

Here, `op` and `def` are relations output by Vandal's decompiler. The `depends` relation is defined by Vandal, with `depends(x,y)` meaning that the value in variable x depends on the value in variable y. The `usedInStateOrCond` relation is also defined by Vandal, with `usedInStateOrCond(var, stmt)` meaning that variable *var* is used in *stmt*, and *stmt* is an operation corresponding to a conditional or a write to storage.

4 Experimental Evaluation

We perform an empirical evaluation of Vandal using a corpus consisting of all 141k unique smart contracts currently deployed on the live Ethereum blockchain, in bytecode form. We compare Vandal's performance to that of the Oyente [20], EthIR [31], Rattle [33], and Mythril [32], and show that Vandal outperforms all systems in terms of average runtime.

4.1 Bytecode Corpus

We build our corpus by running Vandal's scraper (see Section 2) on the live Ethereum blockchain to retrieve bytecode for every contract deployed as of block number 6237983 (2018-08-30). In total there are 7.4M contracts, of which only 141k have unique bytecode.

The total amount of Ether controlled by each unique bytecode in our corpus is shown in Figure 8. We use the number of basic blocks as a simple measure of complexity, and define 'balance' to be the sum of the balances of all contracts with the same bytecode. We observe that bytecodes with a higher complexity control greater amounts of Ether, and that most bytecodes are non-trivial, consisting of between 100 and 1k basic blocks. Figure 9 shows all bytecodes that have a positive balance and occur more than once among all 7.4M. We observe that many heavily-duplicated contracts control balances greater than 100 Ether, which is equivalent to $29k USD as of this writing. These bytecodes are high-value targets: an exploit in just one of these would allow an attacker to compromise and drain thousands of contract accounts.

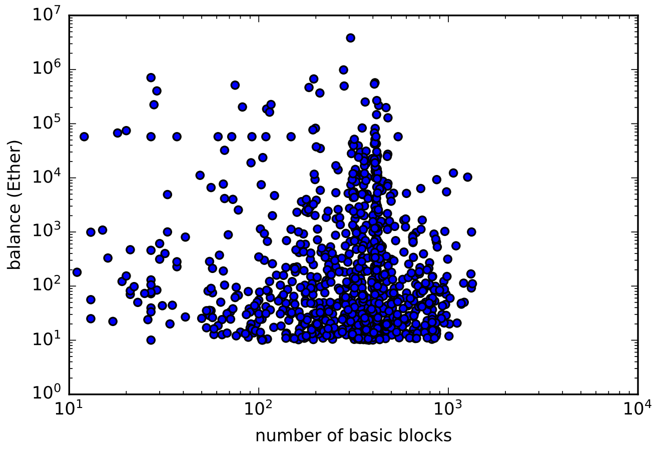


Figure 8: Number of basic blocks and Ether balance for all unique bytecodes.

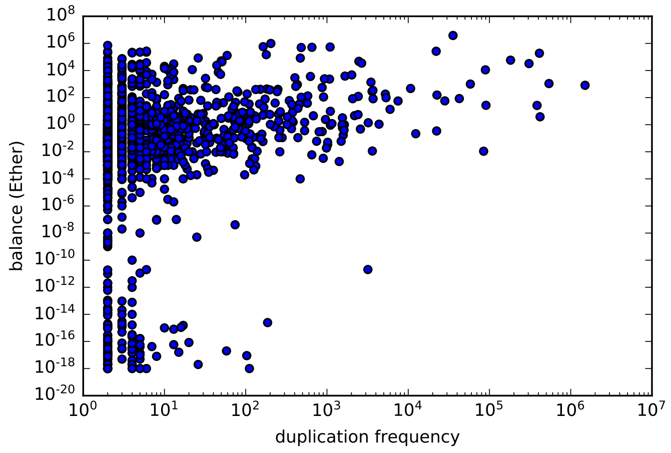


Figure 9: Duplication frequency and Ether balance for all unique bytecodes.

4.2 Comparison of Tools

We compare Vandal's performance to that of Oyente, EthIR, Rattle, and Mythril using our corpus of 141k unique contract bytecodes. In our experiments, a time-out of 60s is used for decompilation of each contract. This is necessary to bound the overall runtime of the experiment. All experiments were run on a machine with an Intel Xeon E5-2680 v4 2.40GHz CPU and 512GB of RAM. Other than our experimental workload, the machine was idle. We used 56 concurrent analysis processes, since our machine has 56 logical cores.

Tool	Min (s)	Max (s)	Average (s)	Total (h)
Oyente	0.26	60.00	13.68	9.69
EthIR	0.25	59.99	11.99	8.89
Mythril	1.47	59.99	11.10	13.51
Rattle	0.12	60.00	4.47	3.56
Vandal	0.29	59.99	4.15	8.08

Table 1: Runtime statistics for each tool for all successfully analyzed contracts. The total runtime includes both successful and unsuccessful contracts.

As can be seen in Table 1, Vandal outperforms all existing tools in terms of average runtime per contract. The total runtimes shown here represent total wall clock time, which is not correlated with the averages because it includes contracts that time out or cause the tool to exit in an error state. We observe that the average runtime of all tools for successfully-decompiled contracts is more than four times below our timeout threshold of 60s, indicating that increasing the timeout further would likely result in diminishing returns.

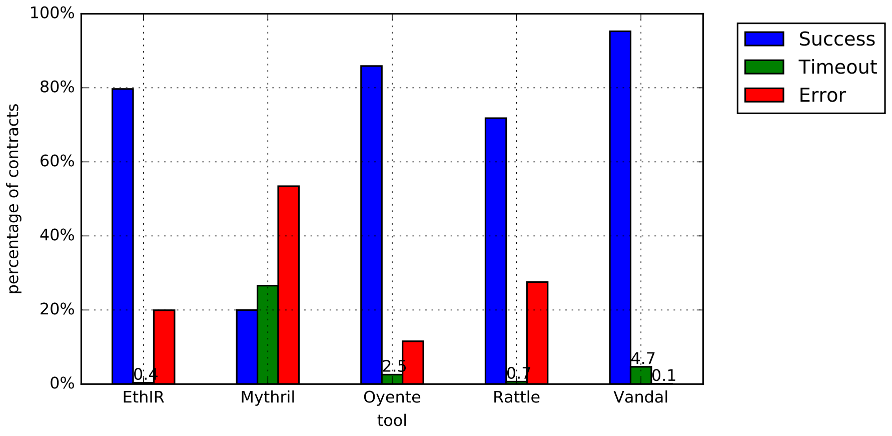


Figure 10: Percentage of contracts that resulted in successful decompilation, timeout, or error states with each analysis tool.

In Figure 10 we show the percentages contracts that were successfully analyzed, vs. the percentages that resulted in an error or timed out after 60s, for each tool. We see that Vandal has the highest success rate and lowest error rate in comparison to every other tool, successfully decompiling over 95% of all contracts and exiting in an error state for only 0.1% of contracts.

Vulnerability Analysis	Vandal	Mythril	Oyente
Destroyable	✓	✓	✗
Reentrancy	✓	✓	✓
Unchecked Send	✓	✓	✗
Unsecured Balance	✓	✓	✗
Use of ORIGIN	✓	✓	✗

Table 2: Comparison of tool support for our chosen vulnerability analyses.

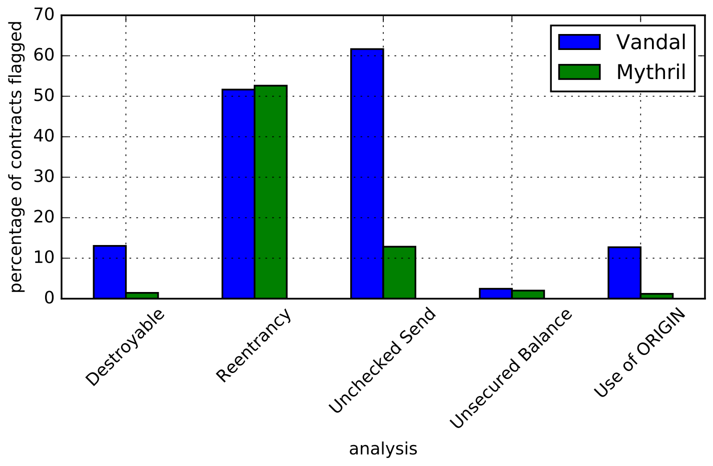


Figure 11: Percentage of contracts flagged as vulnerable for each analysis with Vandal and Mythril.

Out of these tools, only Vandal, Oyente, and Mythril are capable of flagging security vulnerabilities. The detection capabilities of each tool with respect to the analyses described in Section 3 are shown in Table 2. (Note that Oyente and Mythril both implement analyses for a wider range of vulnerability classes than just those shown in Table 2.) In Figure 11 we compare the percentage of successfully-decompiled contracts that were flagged by Vandal and Mythril for each analysis. With the exception of reentrancy, we observe that Vandal flags a higher percentage of contracts for all analyses. This is expected: Vandal's use of abstract interpretation ensures that *all* possible executions are explored, whereas Mythril's concolic analysis examines only a subset of execution traces, possibly missing some true positives. For reentrancy, the difference between Mythril and Vandal in Figure 11 is less than 1%. We attribute this discrepancy to noisy data, given the high error rate of Mythril (see Figure 10). Although Vandal flags more than 50% of contracts for reentrancy and unchecked send, some of the flagged contracts may not be practically exploitable, and others may be false positives.

5 Related Work

Approaches and tools for analysis and verification of smart contracts have received widespread attention in the literature due to the high-risk environment of smart contract development.

Decompilers for Smart Contracts. Porosity [37] is a high-level decompiler from EVM bytecode to Solidity-like source that was introduced by M. Suiche at DEF CON 25. Porosity is implemented in C++ and is intended to be a preliminary prototype for decompiling EVM bytecode to high level Solidity source code. The EthIR [31] framework is built upon the trace-based Oyente tool [20] and performs high-level analysis of Ethereum bytecode. It outputs an intermediate representation in which local variables are introduced to each basic block, simplifying the analysis. Although EthIR reconstructs fragments of high-level control- and data-flows from Oyente traces, it does not improve on Oyente's control-flow graph recovery capability. Mythril is a security analysis tool for Ethereum smart contracts [32]. Mythril performs decompilation aided by a dynamic symbolic execution engine (Laser-EVM). It produces traces that are used to generate an intermediate representation and it therefore suffers similar incompleteness issues as EthIR. Similarly, Rattle [33] also constructs an IR in SSA form [38] and performs program analysis on it. The teEther [39] exploit generation tool, although not a decompiler, operates on EVM bytecode and uses a form of iterative data-flow analysis to reconstruct contract control flow graphs. teEther automatically generates exploits for vulnerable contracts using symbolic execution with the Z3 constraint solver to solve path constraints. It generates exploits by focusing on "critical paths" in the control flow graph; those containing sensitive state-changing operations such as inter-contract calls and writes to persistent storage. Unlike our Vandal framework's Datalog interface, however, teEther does not have a focus on usability and rapid prototyping of new vulnerability analyses. teEther is implemented in Python and will be released as open source, but is not yet available as of this writing.

Exploit Identification and Analysis. Previous works which applied static program analysis for smart contracts can be classified according to their underlying techniques, including dynamic symbolic execution, formal verification, and abstract interpretation.

Systems including Oyente [20], MAIAN [21], GASPER [23] and recent work [40] use an approach based on dynamic *symbolic execution* or trace semantics, which is fundamentally unsound since only some program paths can be explored.

Semi-automated *formal verification* approaches have also been proposed [41–46] for performing complete analyses of smart contracts using interactive theorem provers such as Isabelle/HOL [47], F* [48], Why3 [45], and $\mathbb{K}$ [49]. 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 [41]. Other systems such as KEVM [42] use 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 analysing the millions of smart contracts currently deployed on the blockchain.

In contrast to formal verification work for smart contracts, *abstract interpretation* approaches [19, 50] do not require human intervention; however, they introduce false-positives. The ZEUS framework [19] translates Solidity source code to LLVM [51] before performing the actual analysis in the SeaHorn verification framework [52]. An alternative approach is that of [50], in which Solidity code is abstracted to finite-state automata.

The approach of our Vandal framework [54] is also partly that of abstract interpretation. However, in contrast to the aforementioned abstract interpretation frameworks, Vandal performs analysis directly on EVM bytecode using a purpose-built decompiler that translates EVM bytecode to an analyzable intermediate representation.

Coverage of Vulnerabilities by Existing Tools. Oyente [20] identifies four vulnerabilities: transaction-ordering dependence, timestamp dependence, exceeding the call stack limit of 1024 (callstack attack) and reentrancy. The formal verification tool by [44] detects three classes of vulnerabilities, two of which were covered by Oyente. These include checking the return value of external address calls, and reentrancy. However, an upper bound analysis on gas required for a given transaction was created. These patterns were verified in F* by translating the contracts into F* code, from which patterns were applied to detect vulnerabilities. Similarly, the FSolidM framework [50] checks for reentrancy and transaction ordering vulnerabilities. It can also detect coding patterns such as time constraint and authorization issues [55]. The MAIAN framework [21] focuses on finding vulnerabilities in smart contracts such as locking of funds indefinitely, leaking funds to arbitrary users, and smart contracts that can be killed by anyone. The GASPER [23] tool identifies gas-costly patterns in contract bytecode, often caused by inefficiencies in the Solidity compiler. The ZEUS system [19] conducts policy checking for a set of policies including reentrancy, unchecked send, failed send, integer overflow, transaction state dependence/order and block state dependence. teEther has support for detecting vulnerabilities that allow an attacker to take control of funds, execute third party code, or kill the contract. Another work focuses exclusively on detecting non-callback free contracts. [40]

6 Conclusion

We presented Vandal, a new static analysis framework for detecting security vulnerabilities in smart contract bytecode. Vandal consists of an analysis pipeline, including a decompiler that performs abstract interpretation to translate bytecode to a higher level intermediate representation in the form of logic relations. Vandal uses a novel logic-driven approach for defining security vulnerability analyses, and includes a static analysis library to ease the development of new analysis specifications. Through a use-case study, we demonstrated the ease with which vulnerability analyses can be implemented in Vandal, often requiring only a few lines of Soufflé code. Finally, we performed a large-scale empirical

[1]Soundy: *"as sound as possible without excessively compromising precision and/or scalability"* [53]

experiment by running Vandal on all 191k unique smart contracts scraped from the Ethereum blockchain. We showed that Vandal outperformed the Oyente, EthIR, Mythril, and Rattle analysis tools in terms of average analysis time and error rate.

References

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

[2] Tapscott D, Tapscott A. *Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World*. London : Portfolio Penguin, 2016: Penguin . 2016.

[3] R3 Consortium . Corda—an open-source distributed ledger platform. https://www.corda.net/; 2017.

[4] Flood MD, Goodenough OR. Contract as automaton: the computational representation of financial agreements. https://dx.doi.org/10.2139/ssrn.2648460; 2015.

[5] Science fUKGO. Distributed ledger technology: beyond block chain. https://www.gov.uk/government/publications/distributed-ledger-technology-blackett-review; 2016.

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

[7] Blockchain C. Cardano: decentralized public blockchain. http://www.cardano.org; .

[8] web.archive.org . Ethereum (ETH) price, charts, market cap, and other metrics | CoinMarketCap. https://web.archive.org/web/20180911060543/https://coinmarketcap.com/currencies/ethereum/; 2018. Accessed: 2018-09-11.

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

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

[11] Leising M. The Ether Thief. https://www.bloomberg.com/features/2017-the-ether-thief/; 2017.

[12] blog.zeppelin.solutions . The Parity Wallet Hack Explained – Zeppelin Blog. https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7; 2018. Accessed: 2018-06-28.

[13] BlockCAT . On the Parity Multi-Sig Wallet Attack. https://medium.com/@BlockCAT/on-the-parity-multi-sig-wallet-attack-83fb5e7f4b8c; 2017. Accessed: 2017-08-22.

[14] Qureshi H. A hacker stole $31M of Ether—how it happened and what it means for Ethereum. http://haseebq.com/a-hacker-stole-31m-of-ether/; 2017. Accessed: 2017-08-22.

[15] medium.com . Another Parity Wallet hack explained – Sergey Petrov – Medium. https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c; 2018. Accessed: 2018-06-28.

[16] steemit.com . Proof of Weak Hands (PoWH) Coin hacked, 866 eth stolen. https://steemit.com/cryptocurrency/@bitburner/proof-of-weak-hands-powh-coin-hacked-866-eth-stolen; 2018. Accessed: 2018-06-28.

[17] Atzei N, Bartoletti M, Cimoli T. A Survey of Attacks on Ethereum Smart Contracts. In: Springer-Verlag New York, Inc. Springer-Verlag New York, Inc.; 2017; New York, NY, USA: 164–186

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

[19] Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: Analyzing Safety of Smart Contracts. In: NDSS. ; 2018.

[20] Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making Smart Contracts Smarter. In: CCS '16. ACM. ACM; 2016; New York, NY, USA: 254–269

[21] Nikolic I, Kolluri A, Sergey I, Saxena P, Hobor A. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* 2018; abs/1802.06038.

[22] medium.com . Solidity: Tx Origin Attacks – Coinmonks – Medium. https://medium.com/coinmonks/solidity-tx-origin-attacks-58211ad95514; 2018. Accessed: 2018-09-11.

[23] Chen T, Li X, Luo X, Zhang X. Under-optimized smart contracts devour your money. In: IEEE. ; 2017: 442-446

[24] Zikai AW, Miller A. Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug. http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/; 2016.

[25] solidity.readthedocs.io . List of Known Bugs — Solidity 0.4.25 documentation. https://solidity.readthedocs.io/en/latest/bugs.html; 2018. Accessed: 2018-09-11.

[26] Aho AV, Lam MS, Sethi R, Ullman JD. *Compilers: Principles, Techniques, & Tools with Gradiance*ch. 12: 921-964; USA: Addison-Wesley Publishing Company. 2nd ed. 2007.

[27] Whaley J, Avots D, Carbin M, Lam MS. *Using Datalog with Binary Decision Diagrams for Program Analysis*: 97–118; Berlin, Heidelberg: Springer Berlin Heidelberg . 2005

[28] Hoder K, Bjørner N, Moura dLM. *Z*- An Efficient Engine for Fixed Points with Constraints. In: . 6806 of *Lecture Notes in Computer Science*. Springer. Springer; 2011: 457-462.

[29] LogicBlox I. Declarative cloud platform for applications that combine transactions & analytics. `http://www.logicblox.com`; .

[30] Jordan H, Scholz B, Subotić P. *Soufflé: On Synthesis of Program Analyzers*: 422–430; Cham: Springer International Publishing . 2016

[31] Albert E, Gordillo P, Livshits B, Rubio A, Sergey I. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In: Springer. Springer; 2018.

[32] Mueller B. Smashing Ethereum Smart Contracts for Fun and Real Profit. `https://github.com/b-mueller/smashing-smart-contracts/raw/master/smashing-smart-contracts-1of1.pdf`; 2018. The 9th annual HITB Security Conference.

[33] Various . Rattle – An EVM Binary Static Analysis Framework. `https://github.com/trailofbits/rattle`; 2018. Accessed: 2018-08-24.

[34] Bravenboer M, Smaragdakis Y. Strictly Declarative Specification of Sophisticated Points-to Analyses. In: OOPSLA '09. ACM. ACM; 2009; New York, NY, USA: 243–262

[35] github.com . GitHub - usyd-blockchain/vandal: Static program analysis framework for Ethereum smart contract bytecode. `https://github.com/usyd-blockchain/vandal`; . Accessed: 2018-03-16.

[36] parity.io . Parity. `https://www.parity.io/`; 2018. Accessed: 2018-03-15.

[37] Various . Porosity – a decompiler for EVM bytecode into readable Solidity-syntax contracts. `https://github.com/comaeio/porosity`; 2018. Accessed: 2018-08-24.

[38] Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 1991; 13(4): 451–490. doi: 10.1145/115372.115320

[39] Krupp J, Rossow C. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In: USENIX Association. USENIX Association; 2018; Baltimore, MD: 1317–1333.

[40] Grossman S, Abraham I, Golan-Gueta G, et al. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2017; 2(POPL): 48:1–48:28. doi: 10.1145/3158136

[41] Grishchenko I, Maffei M, Schneidewind C. A Semantic Framework for the Security Analysis of Ethereum smart contracts. *CoRR* 2018; abs/1802.08660.

[42] Hildenbrandt E, Zhu X, Rodrigues N. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In: Everett Hildenbrandt and Xiaoran Zhu and Nishant Rodrigues. ; 2017.

[43] Hirai Y. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In: Brenner M, Rohloff K, Bonneau J, et al., eds. *Financial Cryptography and Data Security*Springer International Publishing. Springer International Publishing; 2017; Cham: 520–535.

[44] Bhargavan K, Delignat-Lavaud A, Fournet C, et al. Formal Verification of Smart Contracts: Short Paper. In: PLAS '16. ACM. ACM; 2016; New York, NY, USA: 91–96

[45] Why3 . Why3. `http://why3.lri.fr/`; 2018. Accessed: 2018-04-17.

[46] Amani S, Bégel M, Bortin M, Staples M. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In: CPP 2018. ACM. ACM; 2018; New York, NY, USA: 66–77

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

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

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

[50] Mavridou A, Laszka A. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. `http://aronlaszka.com/papers/mavridou2018designing.pdf`; 2018.

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

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

[53] Livshits B, Sridharan M, Smaragdakis Y, et al. In Defense of Soundiness: A Manifesto. *Commun. ACM* 2015; 58(2): 44–46. doi: 10.1145/2644805

[54] Grech N, Kong M, Jurisevic A, Brent L, Scholz B, Smaragdakis Y. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 2018(OOPSLA).

[55] Bartoletti M, Carta S, Cimoli T, Saia R. Dissecting Ponzi schemes on Ethereum: identification, analysis, and impact. `https://arxiv.org/pdf/1703.03779.pdf`; 2017.