Erays: Reverse Engineering Ethereum's Opaque Smart Contracts

Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey, *University of Illinois, Urbana-Champaign*

https://www.usenix.org/conference/usenixsecurity18/presentation/zhou

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-46-1

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

Erays: Reverse Engineering Ethereum's Opaque Smart Contracts

Yi Zhou Deepak Kumar Surya Bakshi Joshua Mason Andrew Miller Michael Bailey

University of Illinois, Urbana-Champaign

Abstract

Interacting with Ethereum smart contracts can have potentially devastating financial consequences. In light of this, several regulatory bodies have called for a need to audit smart contracts for security and correctness guarantees. Unfortunately, auditing smart contracts that do not have readily available source code can be challenging, and there are currently few tools available that aid in this process. Such contracts remain *opaque* to auditors. To address this, we present Erays, a reverse engineering tool for smart contracts. Erays takes in smart contract from the Ethereum blockchain, and produces high-level pseudocode suitable for manual analysis. We show how Erays can be used to provide insight into several contract properties, such as code complexity and code reuse in the ecosystem. We then leverage Erays to link contracts with no previously available source code to public source code, thus reducing the overall opacity in the ecosystem. Finally, we demonstrate how Erays can be used for reverse-engineering in four case studies: high-value multisignature wallets, arbitrage bots, exchange accounts, and finally, a popular smart-contract game, Cryptokitties. We conclude with a discussion regarding the value of reverse engineering in the smart contract ecosystem, and how Erays can be leveraged to address the challenges that lie ahead.

1 Introduction

Smart contracts are programs that facilitate trackable, irreversible digital transactions. Smart contracts are prominently featured in Ethereum, the second largest cryptocurrency. In 2018, Ethereum smart contracts hold over $10 B USD[1]. These can be used to facilitate a wide array of tasks, such as crowdfunding, decentralized exchanges, and supply-chain tracking [32].

[1]At the time of writing in February 2018 the Ethereum to USD conversion is approximately $1.2 K USD per ETH

Unfortunately, smart contracts are historically error-prone [14, 24, 52] and there is a potential high financial risk associated with interacting with smart contracts. As a result, smart contracts have attracted the attention of several regulatory bodies, including the FTC [18] and the SEC [43], which are intent on auditing these contracts to prevent unintended financial consequences. Many smart contracts do not have readily linkable public source code available, making them *opaque* to auditors.

To better understand opaque smart contracts, we present Erays, a reverse engineering tool for Ethereum smart contracts. Erays takes as input a compiled Ethereum Virtual Machine (EVM) smart contract without modification from the blockchain, and returns high-level pseudocode suitable for manual analysis. To build Erays, we apply a number of well-known program analysis algorithms and techniques. Notably, we transform EVM from a stack-based language to a register based machine to ease readability of the output for the end-user.

We next turn to measuring the Ethereum smart contract ecosystem, leveraging Erays to provide insight into code complexity and code reuse. We crawl the Ethereum blockchain for all contracts and collect a total of 34 K unique smart contracts up until January 3rd, 2018. Of these, 26 K (77.3%) have no readily available source code. These contracts are involved with 12.7 M (31.6%) transactions, and hold $3 B USD.

We next leverage Erays to demonstrate how it can be used to link smart contracts that have no readily available source code to publicly available source code. We build a "fuzzy hash" mechanism that can compare two smart contracts and identify whether a function in one contract has similar syntactic structure to functions in another contract. Using this technique, we are able to map a median 50% of functions and 14.7% of instructions per opaque contract, giving immediate partial insight to opaque contracts in the ecosystem.

Finally, we show how Erays works as a reverse engineering tool applied to four case studies — high-value

multi-signature wallets, arbitrage bots, exchange accounts, and finally, a popular smart contract game, Cryptokitties. In investigating high-value wallets, we were able to reverse engineer the access control policies of a large, commercial exchange. We find some standard policies, however, also uncover ad-hoc security devices involving timers and deposits. In studying arbitrage contracts, we find examples of new obfuscation techniques. We then successfully reverse engineer the opaque portion of code from the Cryptokitties game, which plays a role in ensuring fair gameplay. In all of these cases, we find that opacity is expected and sometimes important to the correct functionality of these contracts. In light of this, we posit that smart contract developers may be expecting to achieve "security by obscurity" by withholding their high level code.

We conclude with a discussion of the value of audits, reverse engineering, and where Erays can aid in solving the growing needs of the Ethereum community. We hope Erays will prove useful to the security and cryptocurrency communities to address the challenges that lie ahead.

2 Background

Blockchains and Cryptocurrencies. A blockchain is a distributed network that maintains a globally consistent log of transactions. Public blockchains, such as Bitcoin [40] and Ethereum [50], are typically implemented as open peer-to-peer networks, based on proof-of-work mining. Cryptocurrencies are virtual currencies implemented on a public blockchain, where the transactions are digitally signed messages that transfer balances from one user account (i.e., public key) to another.

Ethereum Smart Contracts. In addition to user accounts, Ethereum also features *smart contract* accounts. A contract account is associated with a fragment of executable code, located at an address. Smart contracts make up approximately 5% of the total Ethereum accounts, account for 31.2% of the overall transactions, and hold 9.4% of total Ether in their balances.

A smart contract is executed when a user submits a transaction with the contract as the recipient. Users include payload data in the transaction, which in turn is provided as input to the smart contract program. A contract is arranged as a collection of functions, which users can invoke. A contract can also trigger the execution of another smart contract through a `CALL` instruction that sends a message, similar to a remote procedure call in other programming paradigms.

Smart contract execution must be replicated by validating nodes on the network. To prevent resource exhaustion, users that create transactions must pay an amount of *gas* for every opcode executed, which translates to certain amount of Ether depending on a market rate.

Contracts are executed in a virtual environment known as the Ethereum Virtual Machine (EVM). EVM defines a machine language called EVM bytecode, which includes approximately 150 opcodes [50]. EVM is a stack-based machine, where opcodes read and write from an operand stack. EVM further provides *memory* and *storage* for additional functionality. Memory is specified as an array used to store volatile data during contract execution. Storage is a key-value store indexed by 256-bit values (one EVM-word). Unlike memory, storage persists across the execution history of a contract and is stored as a part of the global blockchain state.

Developers typically write smart contract code in high-level languages, which are then compiled into EVM bytecode. In 2018, the most popular programming language for Ethereum smart contracts is Solidity [7]. Solidity syntax is heavily influenced by Javascript and C++, and supports a number of complex language features, such as inheritance, libraries, and user-defined types.

Ethereum-based Tokens. In addition to the built-in Ether currency, the Ethereum blockchain is also widely used as a host for "tokens", which are separate currency-like instruments built on top of a smart contract. There are currently more than 33 K such contracts on the Ethereum network. Tokens can be traded as currencies on a variety of market exchanges. Together, the total market capitalization of tokens exceeds $60 B USD.[2] Tokens today are used to support a variety of functions, such as crowd-funding and exchanges.

3 Opacity in Smart Contracts

The bytecode for every smart contract is readily available on the blockchain. However, bytecode alone is is difficult to read and understand, limiting its use in effectively determining what a smart contract does. We begin our analysis of smart contracts by first investigating how many contracts can not be immediately linked back to source code, and characterizing how important those contracts are in the ecosystem.

3.1 Collecting and Compiling Contracts

In order to investigate contracts with missing source code, we first collect all Ethereum smart contracts from the beginning of the blockchain through January 3rd, 2018. This resulted in 1,024,886 contract instances. Not all of these contracts have unique bytecode. After removing duplicates, we find only 34,328 unique contracts, which is a 97% reduction in contracts from the original set.

[2] At the time of writing in February 2018, the Ethereum to USD conversion is approximately $1.2 K USD per ETH.

Type	Contracts	Transactions		Balance (Ether)	
Total	1,024,886	40,380,705	(100%)	9,884,533	(100%)
Unique	34,328	40,380,705	(100%)	9,884,533	(100%)
Opaque	26,594	12,753,734	(31.6%)	2,559,745	(25.9%)
Transparent	7,734	27,626,971	(68.4%)	7,324,788	(74.1%)

Table 1: **Opacity in Ethereum Blockchain** — We show the opacity of contracts in the Ethereum blockchain, as well as the number of transactions and Ether in each category. Although opaque contracts make up 77.3% of unique contracts, they only account for 31.6% of the transactions and 25.9% of the Ether held by contracts.

In order to determine how many blockchain contracts have readily accessibly source code, we turned to Etherscan [3]. Etherscan has become the de facto source for Ethereum blockchain exploration. Etherscan offers a useful feature called "verified" contracts, where contract writers can publish source code associated with blockchain contracts. Etherscan then independently verifies that the compiled source code produces exactly the bytecode available at a given address. Etherscan then makes the verified source available to the public. We scraped Etherscan for all verified contracts as of January 3rd, 2018, collecting a total of 10,387 Solidity files.

We then compiled the Etherscan verified contracts to determine exact bytecode matches with blockchain contracts. Etherscan provides the precise compiler version for each verified source file, so to begin, we compiled each source file with its provided compiler version. From these, we collected 7.5 K unique binaries. To identify variants of contracts that were compiled with older versions of the Solidity compiler, we aggregated every major compiler version from `v0.1.3` to `v0.4.19` and compiled each contract with every version. In total, from the seed set of 10.4 K source files, we collected 88.4 K unique binaries across 35 compiler versions.

3.2 Opacity

We next investigated contract opacity in the Ethereum ecosystem today. Of the 1 M contract instances, we could not successfully match 965 K, or 96.5% to any compiled source code. We find that of the 34 K unique contracts, we are able to successfully match 7.7 K (22.7%) of contracts. Unfortunately, this leaves 77.3% of unique contracts opaque.

We next turn to the question of how important these 77.3% of contracts are to the ecosystem. To quantify importance, we use two metrics: the amount of money stored in each contract, and the transaction volume (by number of transactions) with each contract. Table 1 shows a breakdown of the contracts in our dataset by these two metrics. Although opaque contracts make up most of the smart contracts in the ecosystem, we find that they are in the minority by both transaction volume and balance. Opaque contracts are transacted with 12.7 M times, compared with transparent contracts, which are transacted with 27.6 M times. In addition, opaque contracts only hold $3.1 B USD, while transparent contracts hold $7.3 B USD. Although it appears that transparency in the ecosystem prevails, the fact remains that 12.7 M interactions with contracts and a total of $3.1 B USD are held in contracts for which auditors and regulators have no insight into.

4 System Design

In order to investigate opaque contracts in the Ethereum ecosystem, we introduce Erays, an EVM reverse engineering tool. Erays takes a hex encoded contract as input and transforms it into human readable expressions. In this section, we describe the transformations Erays makes in order to build human-readable representations of smart contracts.

4.1 Disassembly and Basic Block Identification

In the first stage, we disassemble the hex string into EVM instructions, and then partition these instructions into basic blocks. A basic block is a linear code sequence with a single entry point and single exit point [9]. We generate the instructions using a straightforward *linear sweep* [42]. Starting from the first byte in the hex string, each byte is sequentially decoded into the corresponding instruction.

Next, we aggregate instructions into their resultant basic blocks. These are derived through two simple rules. Instructions that alter the control flow (i.e., exits or branches) mark block exit, while the special instruction `JUMPDEST` marks block entry. When all block entries and exits are identified, basic block partitioning is complete. Code Block 1 shows an example of this transformation.

4.2 Control Flow Graph Recovery

In this stage, we recover the control flow graph (CFG) [9] from the basic blocks. A CFG is a directed graph where

```
   hex      instruction
b0:
   6000     PUSH1 0x60
   54       SLOAD
   600a     PUSH1 0xa
   6008     PUSH1 0x8
   56       JUMP
b1:
   5b       JUMPDEST
   56       JUMP
...
```

Code Block 1: **Assembly Code—** We show (part of the) input hex string disassembled and then divided into basic blocks.

each node represents a basic block and each edge denotes a branch between two blocks. In a directed edge `b0` $\rightarrow$ `b1`, we refer to `b1` as the *successor* of `b0`. At its core, recovering a CFG from basic blocks requires identifying the successor(s) of each basic block.

To determine the successor(s) for a basic block `b`, we need to examine the last instruction in the block. There are three cases:

1. An instruction that does not alter control flow
2. An instruction that halts execution (`STOP, REVERT, INVALID, RETURN, SELFDESTRUCT`)
3. An instruction that branches (`JUMP, JUMPI`)

In the first case, control simply flows to the next block in the sequence, making that block the successor of `b`. In the second case, since the execution is terminated, `b` would have no successor. In the last case, the successor depends on the target address of the branch instruction, which requires closer scrutiny.

Indirect branches present a challenge when determining the target address [46]. In a direct branch, the destination address is derived within the basic block and thus can be computed easily. In an indirect branch, however, the destination address is placed on the stack before entering a block. Consider block `b1` in Code Block 1. As mentioned, the destination address is on the top of the stack upon entering the block. We therefore cannot determine the destination address from block `b1` alone.

To address this issue with indirect branches, we model the stack state in our CFG recovery algorithm, shown in Code Block 2. The algorithm follows a conventional pattern for CFG recovery [46]: we analyze a basic block, identify its successors, add them to the CFG, then recursively analyze the successors.

When analyzing a block, we model the stack effects of instructions. The `PUSH` instructions are modeled with concrete values placed on the stack. All other instructions are modeled only insofar as their effect on stack height.

```
explore(block, stack):
   if stack seen at block:
      return
   mark stack as seen at block

   for instruction in block:
      update stack with instruction
   save stack state

   if block ends with jump:
      successor_block = stack.resolve_jump
      add successor_block to CFG
      explore(successor_block, stack)
   if block falls to subsequent_block:
      revert stack state
      add subsequent_block to CFG
      explore(subsequent_block, stack)
```

Code Block 2: **CFG Recovery Algorithm—** We analyze a basic block, identify its successors, add them to the CFG, then recursively analyze the successors

Consider the first two instructions in block `b0` in Code Block 1. Suppose we start with an empty stack at the block entry. The first instruction `PUSH1 0x60` will push the constant `0x60` on the stack. The second instruction `SLOAD` will consume the `0x60` to load an unknown value from storage.

Using this stack model, we effectively emulate through the CFG, triggering all reachable code blocks. At each block entrance reached, we compare the current stack image with stack images observed thus far. If a stack image has already been recorded, the block would continue to a path that has already been explored, and so the recovery algorithm backtracks.

4.3 Lifting

In this stage, we *lift* EVM's stack-based instructions into a register-based instructions. The register-based instructions preserve most operations defined in the EVM specification. Additionally, a few new operations are introduced to make the representation more concise and understandable:

`INTCALL`, `INTRET`: These two instructions call and return from an internal function, respectively. Unlike external functions invoked through `CALL`, internal functions are implicitly triggered through `JUMP` instructions. We heuristically identify the internal function calls [3], which allows further simplification of the CFG.

`ASSERT`: As in many high level languages, this instruction asserts a condition. The solidity compiler inserts

[3] The details of the heuristic are included in the Appendix A.

certain safety checks (e.g., array bounds checking) into each produced compiled contract. In order to eliminate redundant basic blocks, we replace these checks with ASSERT.

NEQ, GEQ, LEQ, SL, SR: These instructions correspond to “not equal”, “greater than or equal”, “less than or equal”, "shift left", and “shift right”. While these operations are not part of the original EVM instruction set, the functionalities are frequently needed. These instructions allow us to collapse more verbose EVM instructions equences (e.g., sequence EQ, ISZERO) into one NEQ instruction.

MOVE: This instruction copies a register value or a constant value to a register. The instructions SWAP (swap two stack items), DUP (duplicate a stack item) and PUSH (push a stack item) are all translated into MOVE instructions.

To derive the registers on which the instructions operate, we map each stack word to a register, ranging from $s0 to $s1023 because the EVM stack is specified to have a maximum size of 1,024 words. Additionally, we introduce two other registers in our intermediate representation, namely $m and $t. The Solidity compiler uses memory address 0x40 to store the free memory pointer. Since that pointer is frequently accessed, we use $m to replace all references to that memory word. The $t register is used as a temporary register for SWAP instructions.

Each instruction is then assigned appropriate registers to replace its dependency on the stack. Consider the instruction ADD as an example. ADD pops two words off of the stack, adds them together, and pushes the result back onto the stack. In our instruction, ADD reads from two registers, adds the values, and writes back to a register. Figure 1 shows both the stack and the registers during an ADD operation. A key observation is that in order to read and write the correct registers, the stack height must be known [49]. In this example, the initial stack height is three, so the ADD reads from $s1 and $s2, and writes the result back to $s1. Our translation for this instruction would be `ADD $s1, $s2, $s1`, where we place the `write_register` before `read_registers`.

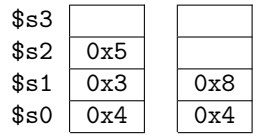


Figure 1: **Lifting an ADD Instruction** — We show both the stack image and the registers before and after an ADD is executed. The initial stack height is three, thus, ADD reads from $s1 and $s2, and writes back the result to $s1.

Knowing the precise stack height is crucial to lifting. As described previously, we collect the stack images for each block during CFG recovery. Given the stack height at the block entrance, all the instructions within the block can be lifted. Code Block 3 shows an example of a basic block being lifted given a stack height of three at the block entrance. We note that the stack images recorded at a block might disagree on height. In most cases, the discrepancy arises from internal function, which is resolved by introducing INTCALL. In other cases, we duplicate the reused block for each unique height observed.

```
PUSH1 0x1              MOVE   $s3, 0x1
SLOAD                  SLOAD  $s3, [$s3]
DUP2                   MOVE   $s4, $s2
LT                     LT     $s3, $s4, $s3
ISZERO                 ISZERO $s3, $s3
PUSH1 0x65             MOVE   $s4, 0x65
JUMPI                  JUMPI  $s4, $s3
```

Code Block 3: **Lifting A BLock** — We show a block of stack-based instructions lifted to register-based instructions given initial stack height of three.

```
SLOAD  $s3, [0x1]
GEQ    $s3, $s2, $s3
JUMPI  0x65, $s3
```

Code Block 4: **Optimizing A Block** — We show the optimized version of Code Block 3.

4.4 Optimization

During the optimization phase, we apply several compiler optimizations to our intermediate representation. We mainly utilize data flow optimizations, including constant folding, constant propagation, copy propagation and dead code elimination. The details of these algorithms are outside the scope of this paper, but they are well described in the literature [8, 38, 47].

The optimizations mentioned aim to simplify the code body. A significant number of available EVM instructions are dedicated to moving stack values. As a result, the lifted code contains many MOVE instructions that simply copy data around. These optimizations eliminate such redundancy in the instructions. Code Block 4 shows the optimized version of the block from Code Block 3. In the example, all the MOVE instructions are eliminated. We also note that the LT, ISZERO sequence is further reduced to GEQ.

4.5 Aggregation

Aggregation aims to further simplify the produced intermediate representation by replacing many instructions

```
SLOAD $s3, [0x1]      $s3 = S[0x1]
GEQ   $s3, $s2, $s3   $s3 = $s2 ≥ $s3
JUMPI 0x65, $s3       if ($s3) goto 0x65
```

Code Block 5: **Three-Address Form** — We show the Code Block 4 in three-address form.

with their analog, compact versions that we term "aggregated expressions." Unlike instructions, expressions can be nested arbitrarily, bearing more resemblance to high level languages.

To begin aggregation, instructions are converted into expressions in three-address form [47]. Each expression is a combination of an assignment and an operator, with the `write_register` to the left of the assignment and the operator along with the `read_registers` to the right of the assignment. Code Block 5 shows the conversion.

Next, we aggregate expressions based on the definitions and usages of registers. A definition is in the form `$r = RHS`, where `$r` is a register and `RHS` is an expression. For each subsequent usage of `$r`, we replace it with `RHS` as long as it is valid to do so. We cease propagating a given definition when either `$r` is redefined or any part of `RHS` is redefined.

Combined with dead code elimination, the aggregation process pushes the definitions down to their usages, producing a more compact output. Consider the example in Code Block 5, by aggregating the first expression into the second one, and then the second into the third, the block can be summarized into a single expression:

```
if ($s2 ≥ S[0x1]) goto 0x65
```

4.6 Control Flow Structure Recovery

We employ structural analysis [44] algorithms to recover high level control constructs (control flow structure recovery). Constructs such as "while" and "if then else" are recovered through pattern matching and collapsing the CFG. If a region is found to be irreducible, we leave the goto expression unchanged. Moreover, each external function is separated by walking through a jump-table like structure at the entrance of the CFG. Code Block 6 shows an external function as an example.

4.7 Validation

Erays transforms the contract into more readable expressions. In order to make use of the expressions for further analysis, we must first validate that they are *correct*. The correctness is evaluated through testing. Given specific contract inputs, we "execute" our representation and check if it produces the correct outputs.

```
assert(0x0 == msg.value)
$s2 = c[0x4]
while (0x1) {
  if ($s2 >= s[0x0])
      break
  if ($s2 <= 0xa) {
    $s2 = 0x2 + $s2
  }
  $s2 = 0xc + $s2
}
m[$m] = $s2
return($m, (0x20 + $m) - $m)
```

Code Block 6: **Structural Analysis** — A simple example of the final output of Erays, where control flow structures are recovered from blocks of expressions.

We use `go-ethereum` (Geth) to generate ground truth for the expected behavior. By replaying an execution (transaction), Geth outputs a debug trace, which is a sequence of execution steps. Each step is a snapshot of the EVM machine state, which includes the opcode executed, the program counter, the stack image, the memory image, and the storage image.

We then "execute" our representation and confirm the result is consistent with the debug trace. For that purpose, we implement a virtual machine that runs our representations. During the execution, the arguments of an expression are first evaluated, then the operation itself is executed given the arguments. There are three classes of operations that need to be treated differently.

In the first case, the operations retrieve some inputs for the contract. As an example, `CALLDATALOAD` fetches part of the input data (calldata). Operations that are dependent on the blockchain world state also fall into this category. An example would be the `BLOCKHASH`, which fetches the hash of a recently completed block. For this class of operations, we look up the resultant value from the debug trace. If an operation is missing in the trace (original trace never issued such call), we mark it as a failure.

In the second case, the operations update the blockchain (world) state. Such operations include storage updates, contract creation, log updates and message calls. We also consider `RETURN` as a member of this category. These operations define the core semantics of a contract. By making sure that all these operations are executed with the right arguments (memory buffers are checked if applicable), we ensure that our representation is correct. If our execution ends up missing or adding any such operations, we mark it as a failure.

The rest of the operations fall into the third case. These operations include the arithmetic operations, memory op-

erations, as well as all the new operations we introduce in our representation. The semantics of the operations are implemented in our virtual machine. As an example, when executing `$s3 = $s2 + $s3`, we would load the values from `$s2` and `$s3`, sum them , modulo by 2^{256} (word size) and put the result in `$s3`. If our machine encounters an exception during these operations, we mark it as a failure.

We leverage historical transactions on the blockchain to construct a set of tests. We start with the set of unique contracts (34 K) described in Section 3. Then, for each unique contract, we collect the most recent transaction up to January 3rd, 2018. In total, we gathered $\tilde{1}$5,855 transactions along with the corresponding contracts in our test set. We note this is only 46% of all unique contracts — the remaining were never transacted with.

If Erays fails to generate the representation in the first place, we mark it as a "construction failure". If our representation behaves incorrectly, we mark it as a "validation failure". In total we fail 510 (3.22%) of the test set, among which 196 are "construction failures" and 314 are 'validation failures".

4.8 Limitations

Erays is not a full decompiler that produces recompilable Solidity code. The major limitation is the readability of the output. While the output is relatively straightforward when only common types are present (`uint array`, `address`), Erays cannot succinctly capture operations on complex types such as `mapping (uint => string)`. Erays's implementation can be improved in a few ways.

Erays uses naive structural analysis for structure recovery. There are several follow-up works on improving the recovery process, including iterative refinement [41] and pattern-independent structuring [51].

Erays does not perform variable recovery and type recovery. Previous work in that area has been focusing on x86 architecture [12, 30]. Though operating with a different instruction set, Erays could draw from the techniques.

5 Measuring Opaque Smart Contracts

In this section, we leverage Erays to provide insight on code complexity and code reuse in the ecosystem. Furthermore, we demonstrate how Erays can be used to reduce contract opacity. We run Erays on the 34 K unique contracts found on the Ethereum blockchain. We fail to create CFGs for 445 (1.3%) unique binaries, which we exclude from our analysis.

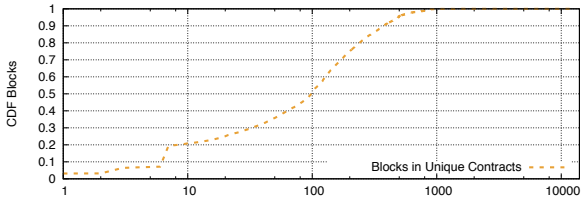


Figure 2: **CDF Contract Blocks** — We show the CDF of the number of blocks in unique smart contracts. The median number of blocks is 100, which denote relatively small programs. However, there is a long tail of very large contracts — the largest contract contains a total of 13,045 basic blocks.

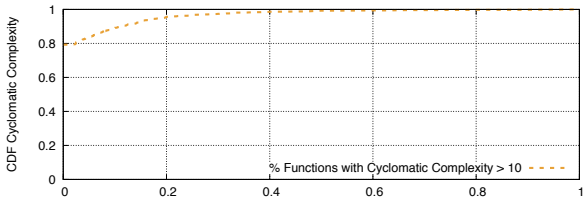


Figure 3: **Complexity of Contracts** — We show the cyclomatic complexity of contracts on the blockchain, by the fraction of functions in each contract with complexity larger than 10. Only 34% of contracts have no functions in this criteria. The median is 0.3, with a long tail of contracts that have increasingly complex functions.

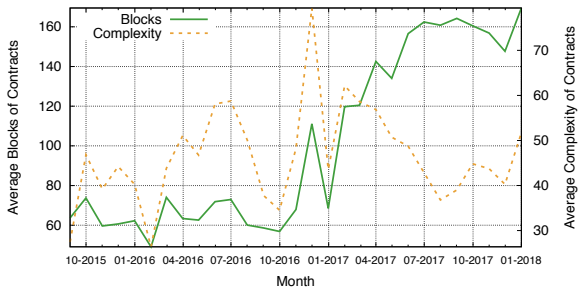


Figure 4: **Longitudinal Complexity** — We show the complexity of unique contracts on the blockchain by the number of blocks and overall McCabe complexity. Contracts have steadily increased in the number of blocks over time, indicating larger contracts today. Despite this, contracts have not increased in overall McCabe complexity, indicating better code hygiene.

5.1 Code Complexity

Our analysis tools give insight into the complexity of contracts found on the blockchain. We begin by investigating the number of blocks in Ethereum contracts (Figure 2). Most contracts are fairly small — the median number of blocks found in contracts is 100, and these blocks contain a median 15 instructions. However, there is a long tail of more complicated contracts. In the largest case, one contract contains a total of 13,045 blocks. However, we find that this contract is one entirely filled with `STOP` instructions, which each terminate their own basic block.

Basic blocks only give one flavor of contract complexity. Just as important are the edges and the connections between the blocks in the CFG. To quantify this, we measure the cyclomatic complexity of each contract, which is a popular software metric introduced by Thomas McCabe [33]. Cyclomatic complexity measures the number of linearly independent paths in a given control flow graph. McCabe suggested that a given function with cyclomatic complexity greater than 10 often needed to be refactored or redone, due to unnecessary complexity and an increased chance of errors in the program. Past work has also noted a weak relationship between increased cyclomatic complexity and software security [45].

Figure 3 shows a CDF McCabe complexity by the fraction of functions in contracts with complexity > 10. We find that 79% of unique contracts do not contain a single function with complexity greater than 10, which indicates that in addition to being small, many contracts do not contain unnecessarily complex functionality. We additionally observe that there is a long tail of complex contracts, and in the worst case, a handful of contracts are entirely filled with overly complex functions.

We finally investigate how code complexity has evolved over time. Figure 4 shows both the number of blocks and the McCabe complexity of new contracts over time. We find that contracts are growing larger at a steady rate — the average number of blocks in contracts published in January 2018 is 170, which is 350% greater than the first contracts published in late 2015. However, we were surprised to find that McCabe complexity has not followed a similar trend. Around January 2017, contract complexity *declined*, and has been relatively stable since. This indicates that contract writers are writing code with better hygiene. We note that around this time, there was a sharp rise in `ERC20 Tokens` on the Ethereum blockchain, which tend to be larger contracts that contain an average of 226 blocks. However, they are not particularly complex, and have an average McCabe complexity of 51.6, which is smaller than many contracts in the ecosystem. ERC20 tokens make up 25% of the unique binaries in our dataset.

5.2 Code Reuse

Erays groups basic blocks into its higher-level functions. From these groupings, we can further compare the structure and code of functions across contracts, giving us a useful metric for determining function similarity. To enable this measurement, we interpret a function as a "set of blocks" and compare the sets across functions in different contracts. Each block, however, may contain contract specific data that would render the comparison useless, such as specific return address information or constants compiled into a block. In order to handle these cases, we remove all references to constant data found in EVM opcodes. As an example, consider the following code block:

hex	opcode	reduced hex
6060	PUSH1 0x60	60
6040	PUSH1 0x40	60
52	MSTORE	52
6004	PUSH1 0x4	60
36	CALLDATASIZE	36
10	LT	10
61006c	PUSH2 0x6c	61
57	JUMPI	57

This shows the original hex string, as well as the decoded opcode and the reduced hex after removing constant values. We then take the hashes of the resultant blocks as the "set" of blocks in a function, and compare these sets in further analysis. From here on, we call this resultant hash set a function "implementation". We find that there are a handful of implementations that are found in many contracts; in the most extreme case, the most popular function appears in 11K contracts. Unfortunately, many of the functions with the same implementation are not particularly interesting — many are simply public "getter" methods for specific data types. For example, the most popular function by implementation is the public getter function for the `uint256` data type.

We next turn to investigate popular external functions included in contracts, and the number of implementations of each of those functions. As mentioned previously, each external function is identified via a 4-byte signature in each solidity contract. Table 2 shows the top 10 function signatures found in our dataset. We note all of the top functions are related to the ERC20 specification, which ERC20 tokens must conform to [26]. Interestingly, we find that although these functions appear in several contracts, there are far fewer implementations of each function. Some of these can be easily explained, for example, the `decimals()` function is simply a 'getter" method for getting the precision of a token. Other functions, however, are harder to explain. The function `transfer(address,uint256)` typically contains busi-

Function Name	Contracts	Implementations
`owner()`	11,045 (32.2%)	63
`balanceOf(address)`	10,070 (29.3%)	240
`transfer(address,uint256)`	9,424 (27.5%)	1,759
`name()`	9,154 (26.7%)	109
`symbol()`	9.087 (26.4%)	120
`decimals()`	8,916 (26.0%)	96
`totalSupply()`	8,732 (25.4%)	200
`allowance(address,address)`	8,102 (23.6%)	152
`transferFrom(address,address,uint256)`	7,979 (23.2%)	1,441
`approve(address,uint256)`	7,713 (22.5%)	479

Table 2: **Function Distribution** — We show the distribution of functions in unique smart contracts. All of the top functions are related to ERC20 tokens [26], which are required to implement a specific interface.

ness logic for a token that defines how token transfers happen, and are somewhat custom. However, despite appearing in 9.4 K contracts, there are only 1.4 K implementations in our dataset. This indicates many contracts sharing the same implementation for such functions.

5.3 Reducing Contract Opacity

A useful product of Erays is the ability to identify the functional similarity between two EVM contracts (Section 5.2). We can extend this technique further to not just investigate code reuse, but to reduce opacity in the ecosystem. We do this by leveraging the compiled dataset of 88.4 K binaries generated from verified Etherscan source code as described in Section 3. From each of these compiled binaries, we extract its functions, and then compare function implementations pairwise from the compiled binaries to binaries collected from the blockchain. An exact function match to a compiled function thus immediately gives us the source code for that function from its originating source file. We view this as similar to the technique of "binary clone detection" [15,39], a technique that overlays function symbols onto stripped binaries using a full binary.

We apply this technique to the opaque contracts on the blockchain, i.e the ones that do not have easily linkable source code. Among the 26 K unique opaque contracts, we are able to reduce the opacity of the opaque contracts to varying degrees. We are able to map a median 50% of functions and 14.7% of instructions per opaque contract. Notably, we reveal 2.4 K unique contracts that we now have *full* source code for. These newly transparent contracts are what we call "frankenstein" contracts — contracts for which source code comes from multiple different contracts.

These techniques additionally improve the opacity in the ecosystem for top contracts. Table 3 shows the top 10 contracts by balance held — the largest of which holds a total of 737 K Ether. Of these contracts, five could not be directly mapped to a verified source contract. After applying Erays, we are able to successfully uncover an average of 66% of the functions in each contract, and in one case, match 100% of the functions in the contract exactly. This contract holds a total of 488 K Ether, which in 2018, is valued at 500 M USD.

6 Reverse Engineering Case Studies

In this section, we show how Erays can be used as a reverse engineering tool in analyzing opaque Ethereum smart contracts.

6.1 Access Control Policies of High-Value Wallets

To begin our analysis, we investigate the opaque smart contracts with the highest Ether balance. Using Erays, we find that many of these are *multisignature wallets* that require multiple individuals to approve any transaction — a standard cryptocurrency security measure.

The opaque wallet with the largest balance contains $597 M USD as of February 2018. Through blockchain analysis using Etherscan, we observed that this contract was accessed every week from the same account, `0xd244...`, which belongs to Gemini, a large cryptocurrency exchange.[4] This address accesses two other high value, opaque wallets in our dataset, with $381 M and $164 M USD in balance, respectively.

We use Erays to reverse engineer these contracts, and uncover their access control policies. We find that the first two contracts are nearly identical. In order to withdraw money from the wallet, they require two out of three administrator signatures. Any party can call the

[4] Gemini used this address to vote in a public referendum on Ethereum governance, see https://web.archive.org/web/20180130153248/http://v1.carbonvote.com/

Code Hash	Ether	Contracts	TXs	Verified	Opacity Reduction (number of functions)
375196a08a62ab4ddf550268a2279bf0bd3e7c56	737,021	1	8	✗	87.5%
0fb47c13d3b1cdc3c44e2675009c6d5ed774f4dc	466,648	1	3504	✗	100%
69d8021055765a22d2c56f67c3ac86bdfa594b69	373,023	1	225	✓	–
a08cfc07745d615af72134e09936fdb9c90886af	84,920	1	151	✗	89.5%
319ee480a443775a00e14cb9ecd73261d4114bee	76,281	3	7819	✓	–
a8cc173d9aef2cf752e4bf5b229d224e17838128	67,747	3	83	✓	–
037ca41c00d8e920388445d0d5ce03086e816137	67,317	1	20,742	✓	–
20f46ba0d13affc396c62af9ee1ff633bc49d8b7	53,961	1	52	✗	54.2%
88ec201907d7ba7cedf115abb92e18c41a4a745d	51,879	1	75	✓	–
c5fbfc4b75ead59e98ff11acbf094830090eeee9	43,418	13	104	✗	0%

Table 3: **Top Contracts by Balance** — We show the top 10 contracts by balance, as well as their transaction volume, whether they matched exactly to verified code, and their opacity reduction after applying Erays if they did not match to source code. Of the top contracts without source code, Erays was able to reduce their function opacity by an average of 66%.

`requestWithdrawal` method, however, the contract will not release the funds until the `approveWithdrawal` function is invoked twice, with at least one invocation message signed by an additional administrator. Thus far, the `approveWithdrawal` transactions are initiated from a different address than the administrators. One administrator address has never been used, indicating that runtime analysis would not adequately capture all of the aspects of this contract.

The third Gemini contract contains a more complicated, time-based access control policy. Withdrawals cannot be approved immediately, but instead must remain pending for a short period of time. Through Erays, we find that the `requestWithdrawal` method in this contract features a *time dependency* hazard, which is a known class of Solidity hazards. When generating a unique identifier for a new withdrawal, the contract uses the hash of both a global counter as well as the hash of the previously mined block. The dependence on the previous block hash means that if a short "fork" happens in the blockchain, two different log events for the same withdrawal may be received by the exchange. The exchange must, as a result, take special care in responding to such log messages on the blockchain. We note that in the past, cryptocurrency exchanges have failed to handle related hazards, resulting in significant losses [21].

Access control policies used internally by financial services would typically be private, not exposed to users or the public. However, due to the public nature of Ethereum bytecode, we have demonstrated the potential to audit such policies when they are implemented as smart contracts.

6.2 Exchange Accounts

We next investigate the contracts that appear most frequently on the blockchain. We anticipated many of these contracts would simply be copy-paste contracts based on publicly accessible code — however, we were surprised to find hundreds of thousands of identical contracts, all opaque. We find that many of these contracts are associated with large exchanges that create one contract instance for each user account.

Poloniex Exchange Wallets The largest cluster of identical opaque contracts appears a total of 349,612 times on the Ethereum blockchain. All of these contracts were created by one address, `0xb42b...579`, which is thought to be associated with the Poloniex exchange.[5] We reverse engineer these contracts and uncover their underlying structure. We find that Poloniex wallets define a customer to whom all wallet deposits are ultimately paid. They directly transfer Ether to the customer whenever Ether is deposited into them, acting as an intermediary between the Poloniex exchange and the customer.

Yunbi Token Wallets We found another cluster of contracts that appeared 89,133 times on the blockchain, that belongs to the Yunbi exchange. Through reverse engineering, we find that the wallets allow any address to deposit Ether, but restrict withdrawal transactions to a whitelisted administrator (Yunbi `0x42da...63dc`). The administrator can trigger Ether and token transfers from the wallet, however, the tokens are transferred out of the balance of the Yunbi exchange — the address of the depositor does not ever own any tokens.

Exchange Splitting Contract We found several opaque contracts thought to be gadgets used by the Gemini[4] and ShapeShift exchanges [23] to defend against replay attacks following the hard fork between Ethereum and Ethereum Classic. The contracts serve as a splitter that sits between the exchange and users depositing to it, checking whether a user is depositing coins to the Ethereum Classic chain or the Ethereum chain. Depending on which chain the transaction appears on, the Ether value of the message is sent to a different address.

Opacity in communications with financial institutions

[5] An Ethereum Developer on Reddit communicated with Poloniex regarding this address and confirmed it belongs to them.

over the Internet is expected practice — we do not see the code that runs the online banking services we use. This expectation has seemingly carried over to Ethereum exchanges, but with unforeseen consequences: publicly available bytecode for a particular program can be reverse engineered, and made simpler with tools like Erays. An expectation for opacity is dangerous, as it may lead to lax attention to security details.

6.3 Arbitrage Bots on Etherdelta

We next leverage Erays to investigate the role of arbitrage bots on EtherDelta [2], a popular decentralized exchange. EtherDelta enables traders to deposit Ether or ERC20 tokens, and then create open offers to exchange their currency for other currencies. EtherDelta is the largest smart contract-based exchange by trade volume, with over $7 million USD daily volume at the time of writing.

On occasion, *arbitrage opportunities* will appear on EtherDelta, where simultaneously buying and selling a token across two currencies can yield an immediate profit. Such opportunities are short lived, since arbitrageurs compete to take advantage of favorable trades as rapidly as possible. A successful arbitrage requires making a pair (or more) of simultaneous trades. In order to reduce risk, many arbitrageurs have built Ethereum smart contracts that send batch trades through EtherDelta. We use Erays to reverse engineer these contracts and investigate their inner-workings.

To begin, we built a list of 30 suspected arbitrage contracts by scanning transactions within blocks 3,900,000 to block 4,416,600, and selected contracts that both make internal calls to EtherDelta and generate two trade events in a single transaction. To prune our list, we ran our similarity metric (described in Section 5) over every pair of the 30 contracts and found three clusters of highly similar ($> 50\%$ similarity) contracts. We then reverse engineered one representative contract from each group.

All three clusters of contracts share the same high-level behavior. The arbitrageur initiates a trade by sending a message to the contract, which first performs an access control check to ensure that it is only invoked by the contract's original creator. Next, the contract queries the `availableVolume` method in EtherDelta, to identify how much of their open offer remains for a given trade. For example, consider a trader who makes an offer of 10 Ether at a price of $1,000 USD. If 8 Ether were purchased, `availableVolume` would return a value of 2. If the contract finds there is sufficient balance on its open offer, it then calls the `trade` function in EtherDelta twice, thus executing the arbitrage trade. If either trade fails, the entire transaction is aborted using the `REVERT` opcode.

Several arbitrage contracts we investigated exhibited different variations of this behavior. Immediately before calling the `trade` function, one group of contracts executes the `testTrade` function, presumably in an attempt to reduce risk. However, since `testTrade` calls the `availableVolume` function *again*, this is redundant and wastes gas.[6] Another group of contracts appears to obscure the values of their method arguments by performing an `XOR` with a hardcoded mask. Such obfuscation is presumably intended to prevent network nodes and other arbitrageurs from front-running or interfering with their transaction. However, this thin veneer becomes transparent through reverse engineering with Erays.

6.4 De-obfuscating Cryptokitties

Cryptokitties is a popular smart contract based trading game on Ethereum. The game involves buying, breeding, and selling virtual pets. As of January 29, 2018, the top 10 "kitties" are worth more than $2.5 M combined. During their peak, they were so popular that gas prices and transaction confirmation times slowed heavily due to Cryptokitties traffic [1, 28].

Although most of the Cryptokitties source code is published, a central component of the game code is *deliberately* kept opaque in order to alter the gameplay. Cryptokitties contain an opaque function, `mixGenes(uint32 matron, uint32):uint32`, which creates a new kitty by splicing together 32-byte genomes from each of two "parents". Kitties are assigned certain visual characteristics based on their genome, and rare attributes can yield very profitable kitties. The gameplay effect of opacity is to make it challenging for users to "game" the gene splicing contract in order to increase the chances of breeding a rare cat. Although the high-level code is known to the developers, the developers have committed to a policy of not playing the game or utilizing this information. As a final case study, we apply Erays to the Cryptokitties contract.

With 3 hours of reverse engineering work using Erays, we were able to create a Solidity contract whose output exactly matches the output of the `mixGenes` function on the blockchain. We find that the `mixGenes` function is comprised of three main parts. The first selects the randomness that will be used: if the hash of the input block number is 0, it is masked with the current block number. The new block number and its hash are concatenated with the parent's genes as input to the `keccak256` hash function, whose output is used as the source of randomness for the rest of the execution. Second, the genes of each parent are split into 5 bit segments and mixed. For each 5-bit gene, one of the parents' genes is chosen as the output gene with 50% probability. Finally, a particular gene is mutated with 25% probability if the larger of the

[6] See Chen et al [16] for a survey of underoptimization in Ethereum contracts.

two parents' corresponding gene is less than 23 and with 12.5% probability otherwise.

Concurrent to our work in reverse engineering, at least three other teams also attempted to reverse engineer the `mixGenes` function [22, 27, 48]. Their analysis largely leverages transaction tracing and blockchain analysis to reverse engineer the "protocol" of the contract. Erays does not rely on transaction data—it directly translates the bytecode to high level pseudocode. As a result, uncommon or unused control paths that do not appear in transaction traces, such as Cryptokitties mutations, can be replicated faithfully.

Deliberate opacity does not serve the intended purpose of black-boxing the gene mixing functionality. Reconstructing the logic and control flow of the contract using Erays, we identify two opportunities to exploit the game with more effective husbandry. First, we can identify kitties with genes valued 23 or greater which are less likely to encounter random mutation when breeding. Second, since randomness is chosen based on block hashes at the time `giveBirth` is called, we can wait to submit the `giveBirth` transaction until after a block hash that results in favorable breeding.

7 Related Work

Program analysis. Our work is guided by existing works in program analysis [9, 10, 38], as well as studies in decompilation [17, 35, 41]. We draw valuable experience from existing optimization frameworks on JVM. In particular, our system design is largely influenced by Soot [49] and Marmot [25].

Blockchain measurement. Our work is closely related to prior efforts in measurement and analysis of Ethereum and other public blockchains. Much of the analysis on the Bitcoin blockchain has focused on clustering transactions by usage patterns (e.g., gambling or trading) [34] and measuring the performance of the underlying peer-to-peer network [19, 20, 36, 37].

Bartoletti and Pompianu provide a taxonomy of the *transparent* Ethereum contracts available from the Etherscan "verified source" dataset [13], whereas our work is the first to analyze opaque contracts. Bartoletti et al. provide a survey of known smart contract vulnerabilities [11].

Comparison with existing Ethereum smart contract analysis tools. Our reverse engineering tool is complementary to a wide range of existing tools in the Ethereum ecosystem:

Symbolic Execution Engines. There are several symbolic execution engines for Ethereum smart contracts, including Oyente [31], Manticore [4], and Mythril [5]. These tools also operate on EVM bytecode, they focus primarily on detecting known classes of vulnerabilities, rather than assisting reverse engineering.

Debuggers. Several tools provide debugging utilities, including Remix [6] and Geth. Debuggers enable an analyst to step through a trace of contract execution, which is helpful in understanding the contract. Although debugging at the EVM opcode level is feasible, debugging with the aid of higher level representations is preferable if available.

Decompilers. Porosity is the only other decompiler we know of that produces Solidity from EVM bytecode. We ran Porosity over the 34 K unique contracts in our dataset to evaluate how well it performs in comparison to Erays. Porosity produces high-level source code without error for only 1,818 (5.3%) unique contracts. In contrast, Erays produces aggregated expression for 33,542 (97.7%).

Exploit Generator. TEETHER [29] is a tool that automatically creates exploits on smart contracts. TEETHER is a concurrent work with Erays.

8 Discussion

We have shown the feasibility of reverse engineering opaque contracts on Ethereum blockchain. Reverse engineering tools like Erays make it easier to reconstruct high level source code even when none is available. We envision that reverse engineering may be used by "white hate" security teams or regulatory bodies in order to carry out public audits of the Ethereum blockchain. Regardless, reverse engineering remains expensive, and such audits would be simplified if the high-level source were available in the first place. We suggest that the Ethereum community should adopt technical mechanisms and conventions that increase the transparency of smart contract programs. Etherscan's verified source code is a step in the right direction, but more work must be done in order to improve transparency in the ecosystem.

Why are so many contracts opaque, given the ease of publishing source code to Etherscan? In some cases, opacity may be a deliberate decision in order to achieve security through obscurity. Another explanation is that publishing Solidity source code is not yet a strong default, and infrastructure support is only partial. For example, we are not aware of any other block explorer services besides Etherscan that provides a Verified Source code repository. Although Ethereum features a decentralized standard called "Swarm" that supports publishing a contract's Application Bytecode Interface (ABI), including the method signatures and argument types, this standard does not include the full source code. This standard should be extended to support high-level source code as well.

9 Conclusion

Many Ethereum smart contracts on the blockchain are *opaque* — they have no easily linkable source code. These contracts control $3.1 B USD in balance, and are transacted with a total of 12.7 M times. To investigate these contracts, we introduced Erays, a reverse engineering tool for EVM. Erays lifts EVM bytecode into higher level representations suitable for manual analysis. We first showed how Erays can be used to quantify code complexity, identify code reuse, and reduce opacity in the smart contract ecosystem. We then applied Erays to four reverse-engineering case studies: high-value multi-signature wallets, arbitrage bots, exchange accounts, and finally, a popular smart contract game. We identified that smart contract developers may be expecting obscurity for the correct functionality of their contracts, and may be expecting to achieve "security by obscurity" in withholding their high level code. We hope Erays will prove useful for both the security and Ethereum communities in improving the transparency in Ethereum.

Acknowledgments

This work was supported in part by the National Science Foundation under contract CNS-151874, as well as through gifts from CME Group and Jump Trading. The work was additionally supported by the U.S. Department of Homeland Security contract HSHQDC-17-J-00170. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of their employers or the sponsors.

References

[1] Cryptokitties craze slows down transactions on ethereum. http://www.bbc.com/news/technology-42237162.

[2] Etherdelta. https://etherdelta.com/.

[3] Etherscan. https://etherscan.io.

[4] Manticore. https://github.com/trailofbits/manticore.

[5] Mythril. https://github.com/ConsenSys/mythril.

[6] Remix. https://github.com/ethereum/remix.

[7] Soldity documentation. https://solidity.readthedocs.io/en/develop/.

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[9] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, 1970.

[10] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137–, Mar. 1976.

[11] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[12] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'07, pages 1–28, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] M. Bartoletti and L. Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International Conference on Financial Cryptography and Data Security*, pages 494–509. Springer, 2017.

[14] R. Browne. Accidental bug may have frozen 280 million worth of digital coin ether in a cryptocurrency wallet. https://www.cnbc.com/2017/11/08/accidental-bug-may-have-frozen-280-worth-of-ether-on-parity-wallet.html.

[15] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan. Bingo: Cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 678–689, New York, NY, USA, 2016. ACM.

[16] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 442–446. IEEE, 2017.

[17] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Softw. Pract. Exper.*, 25(7):811–829, July 1995.

[18] U. F. T. Commission. Know the risks before investing in cryptocurrencies. https://www.ftc.gov/news-events/blogs/business-blog/2018/02/know-risks-investing-cryptocurrencies.

[19] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, et al. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*, pages 106–125. Springer, 2016.

[20] C. Decker and R. Wattenhofer. Information propagation in the bitcoin network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10. IEEE, 2013.

[21] C. Decker and R. Wattenhofer. Bitcoin transaction malleability and mtgox. In *European Symposium on Research in Computer Security*, pages 313–326. Springer, 2014.

[22] M. Dong. Towards cracking crypto kitties' genetic code. https://medium.com/@montedong/towards-cracking-crypto-kitties-genetic-code-629fcd37b09b.

[23] Etherscan. Shapeshift exchange account. https://etherscan.io/address/0x70faa28a6b8d6829a4b1e649d26ec9a2a39ba413.

[24] K. Finley. A 50 million dollar hack just showed that the dao was all too human. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/, 2016.

[25] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An optimizing compiler for java. *Softw. Pract. Exper.*, 30(3):199–232, Mar. 2000.

[26] E. Foundation. Erc20 token standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard.

[27] A. Hegyi. Cryptokitties genescience algorithm. https://medium.com/@alexhegyi/cryptokitties-genescience-1f5b41963b0d.

[28] O. Kharif. Cryptokitties mania overwhelms ethereum network's processing. https://www.bloomberg.com/news/articles/2017-12-04/cryptokitties-quickly-becomes-most-widely-used-ethereum-app.

[29] J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[30] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[31] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[32] R. Marvin. Blockchain in 2017: The year of smart contracts. https://www.pcmag.com/article/350088/blockchain-in-2017-the-year-of-smart-contracts.

[33] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*.

[34] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[35] J. Miecznikowski and L. Hendren. Decompiling java using staged encapsulation. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 368–, Washington, DC, USA, 2001. IEEE Computer Society.

[36] A. Miller, J. Litton, A. Pachulski, N. Gupta, D. Levin, N. Spring, and B. Bhattacharjee. Discovering bitcoin's public topology and influential nodes. *et al.*, 2015.

[37] T. Neudecker, P. Andelfinger, and H. Hartenstein. Timing analysis for inferring the topology of the bitcoin peer-to-peer network. In *Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences*, pages 358–367. IEEE, 2016.

[38] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999.

[39] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 117–128, New York, NY, USA, 2009. ACM.

[40] s. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

[41] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 353–368, Berkeley, CA, USA, 2013. USENIX Association.

[42] B. Schwarz and G. A. Saumya Debray. Disassembly of executable code revisited. In *9th IEEE Working Conference on Reverse Engineering*.

[43] U. Securities and E. Commission. Investor bulletin: Initial coin offerings. https://www.sec.gov/oiea/investor-alerts-and-bulletins/ib_coinofferings.

[44] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[45] Y. Shin and L. Williams. Is complexity really the enemy of software security? In *4th ACM workshop on Quality of protection*.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. (state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 138–157. IEEE, 2016.

[47] L. Torczon and K. Cooper. *Engineering A Compiler*. 2007.

[48] K. Turner. The cryptokitties genome project. https://medium.com/@kaigani.

[49] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *1999 conference of the Centre for Advanced Studies on Collaborative research*.

[50] G. Wood. Ethereum: A secure decentralised generalised transaction ledger.

[51] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[52] W. Zhao. 30 million: Ether reported stolen due to parity wallet breach. https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/.

A Internal Function Identification

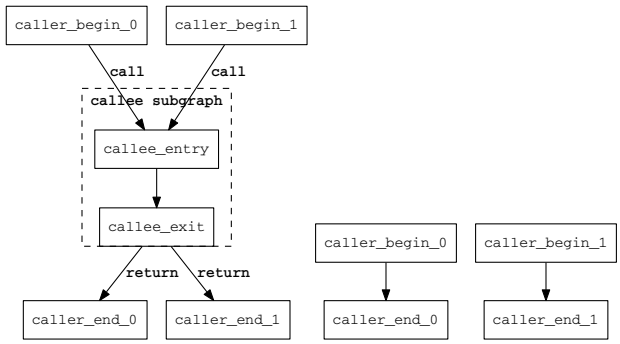
In our heuristic, an internal function is assumed to have a single entry and a single exit. Consequently, there are four basic blocks involved in an internal call that we name `caller_begin`, `callee_entry`, `callee_exit` and `caller_end`. The `caller_begin` issues the call by branching to `callee_entry`, and eventually `callee_exit` returns to the caller by branching to `caller_end`.

We note that callee may have multiple callers. As a result, for an internal function, there is one pair of `callee_entry` and `callee_exit`, but there may be multiple pairs of `caller_begin` and `caller_end`. Figure 5a illustrates an example callee with two callers.

We start by identifying `callee_exit`. We observe that `callee_exit` would normally end with an indirect branch, where the branch address is produced by `caller_begin`. Moreover, `callee_exit` should have more than one successors (the `caller_ends`).

We then correlate each `caller_end` with its `caller_begin`. As mentioned previously, the branch address produced by `caller_begin` guides the callee to `caller_end`. During the CFG recovery, we keep track of where each constant is generated, which enables the correlation. As we identify the `caller_begins`, the `callee_entry` is their common successor.

We then use `INTCALL` as an abstraction for the callee. The subgraph for the callee is first extracted using the CFG recovery algorithm. For each `caller_begin`, we insert an `INTCALL`, and also replace its branch from `callee_entry` to the corresponding `caller_end`. The `INTCALL`, when "executed", will transfer the control flow to the callee. For the `callee_exit`, we insert an `INTRET` to replace its indirect branch to `caller_ends`. The `INTRET`, when "executed", will transfer the control flow



(a) before callee is extracted (b) after callee is extracted

Figure 5

```
delta, stack_size = 0, 0
for bytecode in sequence:
   stack_size -= bytecode.delta
   delta = min(delta, stack_size)
   stack_size += bytecode.alpha
delta = -delta
alpha = stack_size + delta
```

Code Block 7: **Computing the Delta and Alpha of a Sequence**

back to the caller. Figure 5b illustrates the transformations.

To make lifting possible, we also need to determine the number of items popped off and pushed onto the stack by `INTCALL`. In the EVM specification, these are referred to as the delta (δ) and alpha (α) of an operation. For an `INTCALL`, they can be interpreted as the number of arguments and return values.

We note that a sequence of bytecode instructions can be viewed as a single operation, thus the delta and alpha value of the sequence computed in the manner shown in 7.

The stack size is initialized to be zero upon entering the sequence. When the it becomes negative, the sequence is reading prepositioned values. Delta is therefore set to the negation of the minimal stack size. The end stack size indicates the number of values produced by the sequence, but we also need to account for the values popped off the stack. Therefore alpha is the end stack size plus the delta value.

For an `INTCALL`, we select a path from `callee_entry` to `callee_exit`, and compute its delta and alpha. We note that in most cases, the return address is the first argument (at the bottom of the initial stack) and will be popped off eventually, which allows us to fully exhaust the function arguments.