Detecting Token Systems on Ethereum

Michael Fröwis[1], Andreas Fuchs[2], and Rainer Böhme[1,2]

[1] Department of Computer Science, Universität Innsbruck, Austria
michael.froewis@uibk.ac.at
[2] Department of Information Systems, University of Münster, Germany

Abstract. We propose and compare two approaches to identify smart contracts as token systems by analyzing their public bytecode. The first approach symbolically executes the code in order to detect token systems by their characteristic behavior of updating internal accounts. The second approach serves as a comparison base and exploits the common interface of ERC-20, the most popular token standard. We present quantitative results for the Ethereum blockchain, and validate the effectiveness of both approaches using a set of curated token systems as ground truth. We observe 100% recall for the second approach. Recall rates of 89% (with well explainable missed detections) indicate that the first approach may also be able to identify "hidden" or undocumented token systems that intentionally do not implement the standard. One possible application of the proposed methods is to facilitate regulators' tasks of monitoring and policing the use of token systems and their underlying platforms.

Keywords: Smart Contract · Symbolic Execution · ERC-20 · Token Systems · Ethereum

1 Introduction

Arguably, it has been easier to create a virtual asset on Ethereum in 2017 than a website on the Internet in 1997. In September 2018, the market valuation of the well observable virtual assets ("tokens") on the Ethereum platform amounts to US$ 35 billion, not counting the US$ 17.6 billion of ether, the platform's hardwired cryptocurrency.[3] These figures are the result of the 2017 boom of *initial coin offerings* (ICOs), enabled by a combination of a hype around blockchain technology, lack of attractive conventional investment alternatives, and greed.

The sheer amount of money involved calls for regulators to take note and, where necessary, step in. While governments' concerns with cryptocurrencies, such as Bitcoin, were mainly focused on tracking payment flows of criminal origin (e. g., from trade with illegal goods, ransomware, money laundering, terrorism financing), the vast growth of an investment universe in virtual assets poses new challenges. These include enforcement of security laws [10], consumer protection [28], and prudential monitoring in the interest of financial stability [11]. These tasks require proven methods and adequate tools to detect, classify, and monitor virtual assets

[3] Sources: Etherscan.io and Coinmarketcap.com on 12 September 2018, own calculations

on platforms that can in principle host any kind of decentralized application. Therefore, in this work we set out to offer a scientific approach for the relevant case of token detection on Ethereum.

In jargon, *token* is a shorthand for a transferable virtual good. The community distinguishes fungible from non-fungible tokens. Although the notion of fungibility is not precisely defined for all corner cases, a token is said to be fungible if all units are alike, i. e., each unit is interchangeable with every other unit. By contrast, a non-fungible token has an identifying feature, such as a serial number, color, etc.

Typical token systems on Ethereum are computer programs that allow its users to exchange tokens with each other in a decentralized, secure, and atomic way, up to the extent enforceable by the underlying blockchain-based system. Such tokens can be useful in many scenarios. For instance, fungible tokens can serve as means of payment (e. g., sub-currencies), securitized rights (e. g., to vote or claim profit), or store of value. Non-fungible tokens are virtual collectibles.

Our approach is novel in that we detect fungible token systems by the characteristic program behavior, which is related to the secure exchange functionality. The behavior is detected by combining symbolic execution and taint analysis, two established static code analysis techniques, which were adapted to the application. As a comparison base, we also propose a signature-based detection method that searches for instances of standard interfaces for token systems. We compare the effectiveness of both methods on a curated ground truth dataset before we generalize and present results for the entire Ethereum blockchain.

The paper is organized as follows. The next Section 2 introduces necessary background. Sections 3 and 4 present our behavior-based and signature-based methods, respectively. Performance measurements are reported and discussed in Section 5. Section 6 connects to relevant related work, before Section 7 concludes with a discussion and an outlook to future applications and research directions.

2 Background and Principles

This section recalls relevant properties of the Ethereum platform, specifically its virtual machine and calling conventions. It further sets up the static analysis techniques: symbolic execution, taint analysis, and the Ethereum call graph.

2.1 Ethereum Virtual Machine (EVM)

Ethereum is a decentralized system that updates a global state in a public, append-only data structure called *blockchain* [29]. At every point in time, the global state is an injective mapping from addresses to account states. Account states include the balance in ether, permanent storage, and optionally code controlling the account. By convention, accounts with code are called *smart contracts*, whereas accounts without code are called *externally owned accounts*. *Transactions* sent to the Ethereum network update the global state. A transaction can (1) transfer ether between accounts, (2) create new accounts, (3) invoke code

of any smart contract of the current state, or combinations thereof. Arguments can be passed to code by supplying *input data* in the transaction.

The Ethereum Virtual Machine (EVM) is a stack-based virtual machine that executes the bytecode in account states. Single-byte opcodes are followed by an optional immediate argument of length between 1 and 32 bytes. To prevent long-running or infinite computations, Ethereum charges a fee for every instruction executed, accounted in units of *gas*. Most developers program the EVM in *Solidity*, a high-level imperative programming language.

2.2 Ethereum Application Binary Interface (ABI)

The Application Binary Interface (ABI) specifies the calling conventions between smart contracts. Since the EVM has no native concept of functions, every transaction sent to a contract starts the execution at the same entry point. Function-like behavior is implemented by a *function dispatching* mechanism, which evaluates the leading 4 bytes of the input data. Specifically, every function is identified by a 4-byte *function selector*, which is deterministically derived from the hash value of the *function signature*. A function signature is a concatenation of the function name and a list of argument types as defined in Solidity. For example, `transfer(address,uint256)` is a signature for a function called "transfer" accepting two arguments of type "address" and unsigned 32-byte integer, respectively.

Listing 1 illustrates the function dispatching mechanism in EVM bytecode as generated by the Solidity compiler. The full ABI definition can be found at [1].

```
4 : PUSH1 0x4          // Push constant 4 on stack
5 : CALLDATALOAD       // Load first 4 bytes from input data
6 : PUSH4 0xa9059cbb   // Function selector transfer(address,uint256)
7 : EQ                 // Check equality
8 : PUSH1 0x20         // Push jump target 0x20 = 32
9 : JUMPI              // Jump if true (cf. line 7)
10: PUSH1 0x4          // If not equal, continue with this instruction
...
32: JUMPDEST           // Implementation of transfer(address,uint256)
33: ...
```

Listing 1: EVM bytecode illustrating the ABI function dispatching.

The ABI specification is not part of the Ethereum protocol. Anyone is free to define their own calling conventions. However, to our knowledge, all popular compilers targeting the EVM produce ABI-compliant bytecode.

2.3 Symbolic Execution and Taint Analysis

Symbolic execution is a program analysis technique [17]. In contrast to concrete execution, symbolic execution does not only explore one execution path through a program by using concrete inputs, but tries to explore *all* paths in a systematic manner. Program inputs are therefore represented as symbols. The symbolic execution engine executes instructions akin the actual runtime environment as long as no symbolic values are involved. When an instruction depends on at least one symbolic value, the symbolic execution engine cannot execute the instruction directly, but builds a symbolic expression that describes the execution result.

Special consideration is needed when it comes to control flow. Whenever a conditional branch is reached that depends on a symbolic branch condition c within path π_n, the engine cannot decide which path to follow. Consequently, it follows both ($\pi_{n|true} \leftarrow \pi_n \wedge c$, $\pi_{n|false} \leftarrow \pi_n \wedge \overline{c}$) execution paths using backtracking. To avoid the exploration of impossible paths, typical engines use an SMT solver to find a satisfying assignment for the path condition in question. If a suitable assignment is found the path is further explored.

For example, when the code in Listing 1 is symbolically executed with initial path constraint $\pi \leftarrow true$, the symbolic execution engine generates a path constraint $\pi_{true} \leftarrow \delta = \texttt{0xa9059cbb}$ for the path $\langle ..., 4, 5, 6, 7, 8, 9, 32, ... \rangle$, where δ is a symbolic variable representing the first four bytes of the input data. When the symbolic execution of the path corresponding to π_{true} completes, the symbolic execution engine performs backtracking, generates a constraint $\pi_{false} \leftarrow \delta \neq \texttt{0xa9059cbb}$, and continues on the path $\langle ..., 4, 5, 6, 7, 8, 9, 10, ... \rangle$.

In this work we mainly exploit two properties of symbolic execution. First, we use the explored paths as input to *static taint analysis* [25]. Taint analysis is a technique to trace data flows of interest through a program execution. More concretely, we label user inputs with markers ("taint") and track which storage locations are affected by it. Our second use of symbolic execution is to access the structure of symbolic expressions generated by the engine.

Symbolic execution faces many limitations in practice [12]: path explosion, unbounded loops, and the NP-hardness of the SMT problem all require tradeoffs, such as imposing timeouts and skipping paths. The success of symbolic execution can be measured in terms of code coverage. Gladly, most smart contracts on Ethereum are very short programs, gas makes unbounded loops expensive, and therefore Ethereum is more amenable to symbolic execution than other platforms.

2.4 Ethereum Call Graph

Both detection methods introduced in this work operate locally. This means we only analyze code of one address at a time. Consequently, the methods are blind to behavior or signatures located outside the smart contract under analysis. Recall from Section 2.1 that transactions can invoke code of any smart contract active in the current state. Smart contracts can create transactions using the call family[4] of instructions. Such calls are used in smart contracts to (1) interact with other parties (smart contracts), and (2) reuse code already deployed.

A useful tool to look beyond the local address is the Ethereum call graph [16]. It holds information on relationships between contracts obtained by parsing all bytecode on the Ethereum blockchain and extracting all statically encoded addresses used in instructions of the call family. Nodes in the graph are addresses with code. Directed edges denote static calls from caller to callee.

The so-constructed call graph captures only statically encoded references. References to other contracts set on construction, calculated at runtime, or provided as user input are missed. The only practical way to work around this

[4] *CALL*, *DELEGATECALL*, *CALLCODE*, *STATICCALL*

limitation is dynamic analysis, which makes a different trade-off as it limits the analysis to actually executed rather than all possible paths.

3 Behavior-based Token Detection

Now we describe our behavior-based heuristic detection method for fungible token systems on the Ethereum platform. We first justify the behavioral pattern, then present our detection method, and finally discuss known limitations.

3.1 Pattern

Fungiblity means that all tokens in a given token system are alike. As a result, token systems do not need to store which specific token belongs to which party. The only relevant information is who owns how many tokens. A straightforward (and gas-efficient) way to implement the state of a token system is storing a mapping of owners (identified by addresses) to a non-negative number of tokens.

An important property of token systems is the ability to transfer tokens. We assume that a token system wants to preserve the total amount of tokens in circulation as they are transferred. In order to detect smart contracts that behave like token systems we define:

Definition 1. *A* token system *according to its behavior, is a smart contract that (1) stores users' balances as integers in permanent storage, and (2) provides a function to transfer tokens between users while keeping the total balance constant, where (3) the transferred value is controlled by user input.*

Fixing the data type to integers in (1) is reasonable as the EVM does not natively support floating point or rational numbers.

```
1 contract FungibleTokenPattern {
2   mapping(address => uint) balance;
3
4   function sendToken(address to, uint value) public {
5       require(balance[msg.sender] >= value);
6       balance[msg.sender] = balance[msg.sender] - value;
7       balance[to] = balance[to] + value;
8   }
9 }
```

Listing 2: Transfer pattern in Solidity, typical for fungible tokens.

Listing 2 shows a Solidity implementation of a minimalistic token system that complies with Definition 1.

3.2 Detection Method

We propose an approach that analyzes the behavior of potential token systems based on symbolic execution and static taint analysis.

Our approach works as follows. We look for a possible execution path that updates two integers in storage, one for the sender and recipient, by a value

defined as parameter. For (1) and (3) of Definition 1, we use taint analysis to find storage write states (sws), where the value stored can be influenced by user input. For each of those stores of input data sws_0, we try to find a matching store sws_1 that follows our tainted store on some possible execution path. Furthermore, we look for constraints in the path condition that check if the value in some storage field is larger than or equal to some user input field (c_{ge}). This captures the check that the sender's balance cannot be negative. We organize our stores and path constraint in triplets of the form (sws_0, sws_1, c_{ge}), meaning we found a storage write sws_0 with its value influenced by user input. sws_0 is followed by sws_1 on some viable execution path. Additionally, we have a constraint c_{ge} on this path that checks if some storage field is larger than a user input field. We call such a triplet *transfer candidate*. What remains to verify is (2), i. e., whether the operations on sws_0 and sws_1 are really transferring value and if c_{ge} is a constraint on one of the fields written to. Here we apply a heuristic that looks at the term structure of transfer candidates.

Algorithm 1 shows the analysis done for every possible transfer candidate triplet. We use $\lhd$ and $\trianglelefteq$ to denote the proper subterm and subterm relation.

Algorithm 1 Analyzing transfer candidates.

function IsToken$_{sym}(S)$ ◇ a set S of triplets (sws_0, sws_1, c_{ge})
  $s_{ops} \leftarrow \{+, -\}$
  for $(sws_0,\ sws_1,\ c_{ge}) \in S$ **do**
    $b_{rss},\ s_{opsLeft},\ b_{usedC} \leftarrow$ CheckStoreTerm$(sws_0,\ s_{ops}, true)$
    if $b_{rss} \wedge |s_{opsLeft}| = |s_{ops}| - 1$ **then**
      $b_{rss}, s_{opsLeft}, b_{usedC} \leftarrow$ CheckStoreTerm$(sws_1,\ s_{opsLeft}, \neg b_{usedC})$
      if $b_{rss} \wedge s_{opsLeft} = \emptyset \wedge b_{useC}$ **then**
        return $true$ ◇ Found a token-like behavior.
  return $false$ ◇ None of the candidates indicates a tokens system.
function CheckStoreTerm$(sws_n, c_{ge}, s_{ops},\ b_{cToEqC})$
  $b_{selfRef},\ b_{callData},\ b_{toEqC} \leftarrow false$
  $t_{to},\ t_{val} \leftarrow sws_n.to, sws_n.value$ ◇ Store has an address and a value.
  $s_{opFirst} \leftarrow$ FindFirstOpBFS(t_{val}, s_{ops}) ◇ Get first matching function symbol.
  $b_{selfRef} \leftarrow t_{to} \lhd t_{val}$ ◇ Store updates itself?
  $b_{callData} \leftarrow c_{ge}.smallerTerm \lhd t_{val}$ ◇ Term contains input from constraint?
  $b_{toEqC} \leftarrow t_{to} \trianglelefteq c_{ge}.largerTerm$ ◇ Is constraint on assignment?
  return $(t_{selfRef} \wedge t_{calldata},\ s_{ops} \setminus s_{opFirst},\ (b_{cToEqC} \wedge b_{toEqC}) \vee \neg b_{cToEqC})$

Example: We use the example contract in Listing 2 to illustrate how the algorithm works. We refer to source code when possible, although the actual analysis is done on bytecode. First we perform taint analysis to find storage writes influenced by user input. We find stores in lines 6 and 7. Then we look for followup stores along a viable execution path. Only for the store in line 6 we find a following store, namely in line 7. Furthermore, we look at path conditions at the program state of the first store in line 6. We find one suitable condition that matches our restrictions that the condition checks if a storage field is larger

than (or equal to) some user input in line 5. This means we found one transfer candidate to check ($line\ 6, line\ 7, line\ 5$). First we execute CheckStoreTerm on `balance[msg.sender] = balance[msg.sender] - value` with `balance[msg.sender] >= value` as a constraint and $\{+, -\}$ as possible operations, and $b_{cToEqC} = true$. Then we check if the *right hand side* (RHS) of the store term contains itself, which it does. It follows a check if the RHS of the constraint `value` is a subterm of our store term, meaning that the constraint and store refer to the same user input. If that is the case, we check if we can either find an addition or subtraction in our term. FindFirstOpBFS checks all function applications in the term against a list of operations (starting with $\{+, -\}$) and returns a set with the first operation to occur, or an empty set if the operations are not found. Finally, we check if the storage field used in the constraint is the target of the store, which is true in our case. The function then returns a tuple with $(true, \{+\}, true)$, since the terms of our transfer candidate fulfill all conditions. We found that minus is the root operation on the term and already found the constraint value to be written on. We then continue with calling CheckStoreTerm again for the second term, with a reduced list of operations, only looking for plus and no longer looking for writes on our constraint values. This call returns $(true, \emptyset, true)$, thus we found token-like behavior according to our definition.

3.3 Known Limitations

We inherit the limitations from symbolic execution (cf. Sect. 2.3). We use `mythril` [6], a tool designed for security analyses that is known to reach high accuracy [22] despite using heuristics. For our experiments, we run `mythril` with a timeout of 60 seconds and a maximum path length of 58. Furthermore, mythril is under active development and has a couple of limitations that may influence our results and their replicability. For example in taint analysis, the current version of `mythril` (0.18.11) cannot spread taint over storage or memory fields. This can cause problems when function parameters are passed by reference.

The locality is dealt with in the following way: whenever the symbolic execution reaches a call, we consider it as communication with the unknown environment. Hence, the engine introduces a fresh unrestricted symbol for the return value and carries on. That means the analysis is blind to everything that happens outside of the code of the current address. We evaluate the impact of this limitation empirically with the call graph in Section 5.3.

Another limitation lies in the definition of the pattern. It is not straightforward to find the best approximation for the behavior we search for, since the same behavior can be implemented in various ways that may result in vastly different bytecode. What eases this problem somewhat is that much of the bytecode currently deployed on Ethereum is produced by a pretty homogeneous toolchain (Solidity and `solc`). Moreover, gas favors simple programs, often rendering abstractions that would complicate the underlying bytecode uneconomic.

4 Signature-based Token Detection

Now we present a simple signature-based heuristic to detect token systems. It evaluates if the bytecode implements the ABI standard for the ERC-20 interface. We need this method as a benchmark to evaluate the behavior-based approach.

4.1 Pattern

To improve the interoperability of tokens in the Ethereum ecosystem, the community has established a set of standards for token systems. ERC-20 [3] is the most popular standard for fungible tokens. It also serves as basis for extensions, such as ERC-223 and ERC-621. Even ERC-777, while still at draft stage at the time of writing, is backward compatible: a token can implement both standards to interact with older systems that require the ERC-20 interface [2]. Given the vast dominance of ERC-20 today, we restrict our analysis to this standard.

The standard defines six functions and two events that must be implemented to be fully compliant. (Listing 3 in Appendix B shows the ERC-20 interface skeleton in Solidity.) Since the applications of tradable tokens are diverse, the standard does not define how tokens are created, initially distributed, or how data storage should be organized. It only defines that ERC-20 tokens must have functions to securely transfer tokens, and some helper functions to check balances.

4.2 Detection Method

A naïve way to detect tokens is to check if the code implements the methods defined by the ERC-20 standard. From the ABI definition (see Sect. 2.2) we know how function calls are encoded and how functions are dispatched. In order to detect token systems based on a signature we define:

Definition 2. *A* token system *according to its signature is a smart contract that introduces at least 5 of the 6 function selectors defined by the ERC-20 standard.*

We used five as a threshold to account for incomplete implementations of ERC-20.

We use Definition 2 and the fact that the only way to introduce constants in the EVM are `PUSH` instructions. Since function selectors are 4 bytes long according to the ABI, the detection method looks for `PUSH4` instructions. Algorithm 2 takes as input a list of EVM instructions, inspects all 4-byte constants introduced, and checks membership in the pre-determined set of ERC-20 function selectors (variable $s_{signatures}$).

4.3 Known Limitations

This method is obviously prone to false positives if a contract pushes all required constants to the stack but never uses them. This may even happen in dead code. Hence, we also get false positives if we analyze so-called *factory contracts* that create new token systems when called [4]. The code of the factory includes the

Algorithm 2 Detection method based on disassembly and signatures.

$s_{signatures} \leftarrow \{18160ddd, 70a08231, dd62ed3e, a9059cbb, 095ea7b3, 23b872dd\}$
function IsToken$_{\text{Sig}}(I)$ ◇ I is a list of instruction tuples $t \in (opcode \times arg)$
  $s_{constants} \leftarrow \emptyset$
  for $(i_{opcode}, i_{argument}) \in I$ **do**
    if $i_{opcode} = PUSH4$ **then**
      $s_{constants} \leftarrow s_{constants} \cup \{i_{argument}\}$
  return $|s_{constants} \cap s_{signatures}| \geq 5$

code of the token system to create, and thus contains push instructions of the required constants.[5] The common cause for these weaknesses is that the method considers neither data nor control flow.

Similar to the behavior-based method, the signature-based method is a local heuristic. This can result in false negatives. For example, if the smart contract does not implement the ERC-20 interface, but delegates calls to a suitable implementation. This form of delegation is common practice on the Ethereum platform because it makes deployments cheaper. Furthermore, it enables code updates by swapping the reference to the actual implementation [7, 8].

5 Measurements

5.1 Data and Procedure

To evaluate our two detection methods we study the Ethereum main chain from the day of its inception until 30 May 2018.[6] We extract all *unique runtime bytecode instances* and the addresses they are deployed on. With *runtime bytecode* we denote code that is executed when a transaction is sent to the contract after its deployment. This means we do not analyze initialization code.

In total we found 6 684 316 addresses that hosted bytecode at one point in time. From these addresses we extract 111 882 *unique runtime bytecode instances*, henceforth referred to as *bytecode instances* for brevity, unless stated otherwise. Observe that we do not double-count bytecode instances unlike it is often the case in headline statistics on smart contracts. We do not exclude contracts that were disabled by selfdestruct, i. e., we analyze all code ever deployed. Consequently, we also analyze bytecode instances that are barely used.

To evaluate that our detection results are not biased towards barely used or test code, we also define a subset of *active* instances. We define a bytecode instance as *active* if all hosting addresses combined handled a volume of at least 1000 transactions until 30 May 2018.

To build a *ground truth dataset* (GTD) for the evaluation, we downloaded 612 Top ERC-20 tokens[7] from Etherscan. Etherscan, a popular Ethereum block

[5] One such instance can be found at `0xbf209cd9f641363931f65c0e8ef44c79ca379301`
[6] Block number: 5 700 000
[7] Ranked by market cap, retrieved on 23 Aug. 18 from `https://etherscan.io/tokens`

Table 1: Recall of signature- and behavior-based detection methods against our GTD.

	Detected by Sig	**Not Detected by Sig**	
Detected by Behav	87.89% (508)	0.87% (5)	88.75% (513)
Not Detected by Behav	11.25% (65)	0.00% (0)	11.25% (65)
	99.13% (573)	0.87% (5)	100.00% (578)

explorer, curates its top list of ERC-20 tokens by only including systems that are popular, supported by at least one major exchange, compliant with ERC-20, and have a visible website. We exclude all token systems that were created after 30 May 2018, leaving us with a curated list of 595 ground truth token systems, of which we extract 578 bytecode instances.

We run both of detection methods over all bytecode instances and evaluate the results. The signature-based method (Sig) is able to process all of the input contracts. The behavior-based method (Behav) fails to analyze 1373 (1.23% of the total) instances. Failures occur if, for example, the bytecode contains syntactic errors not handled in the engine. We consider those 1373 instances as negative detection results. On the remaining 110 509 instances, our behavior-based method reaches a mean (median) code coverage of 71.9% (82.2%). Over 70% of the instances reach a coverage above 50%, supporting the claim that smart contracts are a very suitable for symbolic execution techniques.[8]

To confirm our restriction to the ERC-20 interface in the signature-based method, we adapted our method to count the number of ERC-777 tokens. We encounter only four systems implementing at least 4 of the 13 functions required by ERC-777. All of them also implement ERC-20 for backward compatibility.

5.2 Validation on Ground Truth Data and Error Analysis

Table 1 presents the detection results of both methods evaluated against our curated GTD. Observe that the signature-based method alone is pretty good at detecting tokens, reaching 99.13% recall. The behavior-based method performs visibly worse with a recall of 88.75% on our curated GTD. Since no token systems remained undetected by both methods, the combination of both (Behav $\vee$ Sig) gives us perfect 100% recall. Our GTD does not allow us to calculate the precision.

The behavior-based method is able to detect the exact five contracts that are missed by the signature-based approach (Behav $\wedge$ $\overline{\text{Sig}}$). Further manual investigation of these contracts shows that all of them do not implement ERC-20 up to our threshold. Fortunately, all of the five contracts published Solidity source code. Thus, we could confirm that they are missed by the signature-method because they implement only three of the six ERC-20 functions, namely

[8] 100% - #16 056, $\leq$ 75% - #50 874, $\leq$ 50% - #31 298, $\leq$ 25% - #5165, $\leq$ 10% - #1031

`totalSupply()`, `balanceOf(address)`, and `transfer(address,uint256)`. This suggests that our initial threshold is too high, or in other words that even major token systems handle standards laxer than expected. Table 8 in Appendix C lists those five contracts.

The signature-based method identified 65 token systems that were not found by the behavior-based method ($\overline{\text{Behav}} \wedge \text{Sig}$). We conjecture that either those tokens implement their internal state differently or they use libraries that implement the bookkeeping of storage values, thereby escaping our local behavior-based analysis. We try to answer why those tokens are not detected by manually inspecting a random sample of 20 (out of 65) bytecode instances (listed in Table 9 in Appendix C). We find that all of them are large and reasonably complex contracts.[9] We encountered three main causes for missed detection:

Delegation of Bookkeeping (6): We found 6 bytecode instances in our sample that do not implement any asset management logic in the contract itself. It is delegated to another contract. The front-end contract implements the ERC-20 interface, but many back-end bookkeeping contracts do not, e. g., the *Digix Gold Token*. Delegation patterns (or "hooks") like this one are often used to allow updates (by reference substitution) of the asset management logic.

Violation of Definition 1 (10): The second reason concerns mainly tokens that are derived from the popular MiniMeToken [5]. We found 9 of those in our sample. MiniMe uses a different storage layout. Instead of a plain integer that is updated over time, it writes a new checkpoint for every transfer into an array. This violates our detection assumption (1) in Definition 1, or, more specifically, fails our check that the field gets updated (self reference). Even though this already defeats our detection method, we find that `mythril` was not able to inspect the relevant paths in the transfer function. The average code coverage is as low as 34.8% on the 9 MiniMe-based tokens in our sample.

Also the *MakerDAO* instance is not detected for a violation of Definition 1, although it is not derived from MiniMe and reached high coverage (94.3%). It does not implement a balance check before the actual transfer as required in Definition 1. This can be fixed with an ad-hoc adjustment of the method, but we are concerned about the (not observable) false positive risk of a relaxed behavioral pattern.

Litations of Symbolic Execution and Taint Analysis (4): Four contracts in our sample neither delegate the bookkeeping work nor are derived from the MiniMeToken. All of them use a simple integer value to store the balance of the participants. *Storiqua* (42.9%), *LocalCoinSwap* (34.82%), and *LOCIcoin* (18.5%) suffer from low coverage. In the *Storiqua* instance, our method finds the relevant paths in the transfer function but does not find a matching store. In the case of *LocalCoinSwap*, we do not find a suitable constraint although the symbolic execution engine explores the relevant paths in transfer. *LOCIcoin* has the lowest coverage. The engine does not discover the relevant paths in the transfer method. Finally, *TrueUSD* reaches high coverage (72.9%), but the behavior-based method

[9] Mean (median) code size: all instances 3315.0 (2541), $\overline{\text{Behav}} \wedge \text{Sig}$ 8153.86 (7828)
Code coverage: 41.75% (40.25%)

did not find a suitable constraint in the transfer function. All of those cases are examples for known limitations of symbolic execution (reaching low coverage, missing relevant paths), taint analysis (failing to find matching stores), as well as our detection approach (missed constraints).

5.3 Generalization to All Smart Contracts on Ethereum

Table 2 reports detection statistics over all bytecode instances. We find that 33.17% of the bytecode instances on the Ethereum platform can be said to be token systems with high confidence because they are detected by both methods (Sig $\wedge$ Behav). The interesting part is where both methods disagree.

Recall from our manual ground truth analysis that all instances missed by the signature-based method but detected by the behavior-based method (Behav $\wedge \overline{\text{Sig}}$) are caused by our high threshold. So we re-run the analysis with a lower threshold of 3, as our manual inspection suggested. Table 4 and Table 5 (both in Appendix A) show the updated results of Table 1 and Table 2, respectively. With the lower threshold, the signature-based method detects 7193 more bytecode instances as tokens. 3232 of those newly detected token systems were already identified by the behavior-based method. The remaining tokens would have been missed otherwise. We conjecture that the 1772 bytecode instances only detected by the behavior-based method (Behav $\wedge \overline{\text{Sig}}$) are either non-ERC-20 bookkeeping contracts, as found in the *Digix Gold Token*, or token systems that do not implement ERC-20 for other reasons, such as obscuring their nature.

In the case of token systems detected by the signature-based but not by the behavior-based method ($\overline{\text{Behav}} \wedge$ Sig), we found mixed reasons in our GTD. First, we saw systems that implement the ERC-20 interface but delegate all bookkeeping tasks to other contracts. In order to study if this pattern generalizes to the whole dataset, we extract bytecode metrics, such as the number of call instructions. We find that contracts that are detected by the signature-based method contain an above-average number of call instructions. Table 7 (in Appendix A) presents the mean and median values of call-family instructions for different subsets of bytecode instances. The highlighted row stands out: $\overline{\text{Behav}} \wedge$ Sig instances have on average 2.2 times as many calls as the average bytecode instance. This indicates the use of delegation patterns as found in the *Digix Gold Token*. To further strengthen this interpretation, we us the Ethereum call graph (cf. Sect. 2.4) to find out if those instances have calls to other instances that are otherwise classified as token systems. For 920 of 10 472 instances ($\overline{\text{Behav}} \wedge$ Sig) we find static references. 563 have direct hardcoded calls to another instance classified as token system, suggesting that the detectable behavior is implemented in the callee. The second and third reason for missing tokens were inherent limitations of symbolic execution, which we could not further evaluate on the entire dataset.

Table 3 shows detection results for all *active* bytecode instances. The results are pretty comparable to Table 5. Note that the behavior-based method misses relatively more instances detected via signature than on the complete dataset. One interpretation is that high-profile tokens implement more complex logic, therefore evading detection by symbolic execution. This conjecture is supported

Table 2: Comparison of signature- and behavior-based detection methods on *all* bytecode instances. (Note that this is not a performance measurement. We cannot expect 100%.)

	Detected by Sig	**Not Detected by Sig**	
Detected by Behav	33.17% (37 114)	4.47% (5004)	37.65% (42 118)
Not Detected by Behav	5.82% (6512)	56.53% (63 252)	62.35% (69 764)
	38.99% (43 626)	61.01% (68 256)	100.00% (111 882)

Table 3: Comparison of signature- and behavior-based detection methods on all *active* bytecode instances (with signature threshold ≥ 3).

	Detected by Sig	**Not Detected by Sig**	
Detected by Behav	32.56% (2052)	0.73% (46)	33.29% (2098)
Not Detected by Behav	18.15% (1144)	48.56% (3060)	48.56% (4204)
	50.71% (3196)	49.29% (3106)	100.00% (6302)

by the observed bytecode sizes as well as code coverage reached: active bytecode instances are on average around 1.6 times as large as the average bytecode instance. Average code coverage also drops from 71.9 (82.2%) to 66.2 (69.6%).

5.4 Insights into the Token Ecosystem

The automatic detection of token systems allows us to shed more light into the token ecosystem. Looking at bytecode reuse, for instance, puts the headline numbers into perspective and informs us about the actual amount of innovation happening in the ICO community. To this end, Figure 1 connects our technical level of analysis (bytecode instances) to the publicly visible level of addresses hosting token systems. The most frequently deployed bytecode instance of a token system is a standard template by *ConsenSys*.[10] It has been deployed 8729 times to the Ethereum blockchain. 298 of these deployments have processed more than 100 transactions. Altogether 49 bytecode instances have been deployed more than 100 times, and 16 bytecode instances have 10 or more "busy" deployments.[11] These figures give some early intuition, but likely underestimate the extent of

[10] `https://github.com/ConsenSys/Token-Factory/blob/master/contracts/HumanStandardToken.sol`

[11] Note that "busy" is similar to our notion of *active*, however on the level of addresses rather than bytecode instances.

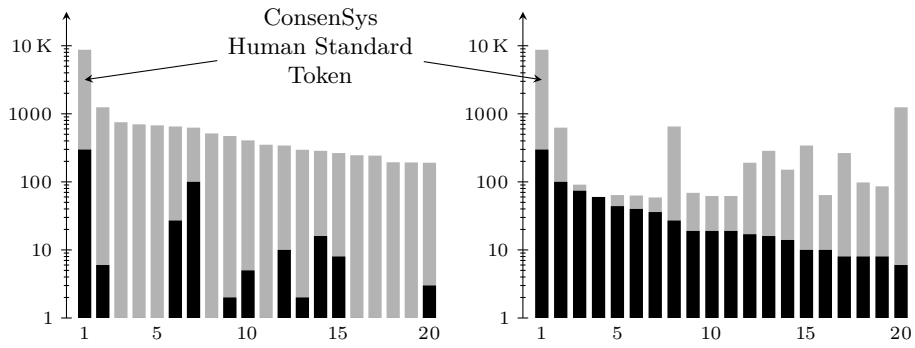

Fig. 1: Bytecode reuse of Ethereum token systems: number of addresses hosting a unique bytecode instance detected as token system. Top-20 ranked by total addresses (left) and "busy" addresses handling more than 100 transactions (right). Note the log scale.

code reuse as trivial modifications of template code (or the output of token factories that deploy polymorphic code) are not consolidated.

6 Related Work

As we are not the first to systematically analyze smart contracts on Ethereum or to study tokens on the Ethereum platform, we summarize prior art by topic area.

Mapping the Smart Contract Ecosystem: Using source code provided by Etherscan, Bartoletti and Pompianu [13] manually classify 811 smart contracts by application domain (e. g., financial, gaming, notary) and identify typical design patterns. Norvill et al. [21] propose unsupervised clustering to group 936 smart contracts on the Ethereum blockchain. Zhou et al. [30] develop Erays a Ethereum reverse engineering tool that lifts EVM bytecode to a human readable pseudocode representation, for futher inspection. They conduct four case studies to show the effectiveness of the approach.

Vulnerability Detection in Smart Contracts: Luu et al. [19] execute 19 366 smart contracts symbolically with the intention to uncover security vulnerabilities, which they find in 8833 cases. Tsankov et al. [27] build Securify, a symbolic execution framework to uncover security problems in smart contracts. Security patterns are specified in a domain-specific language based on Datalog. Nikolic et al. [20] study so-called trace vulnerabilities that manifest after multiple runs of a program. Introducing Maian, a symbolic execution framework to reason about trace properties, they identify 3686 vulnerable smart contracts. Brent et al. [14] present Vandal, a smart contract security analysis framework. It uses a Datalog-based language tailored to describe static analysis checks.

Token Systems: Somin et al. [26] study network properties of token trades and show that the degree distribution has power-law properties. They use a simple token detection method based on ERC-20 events generated at runtime,

therefore relying on the standard compliance of the contracts. Etherscan identifies token systems using a signature-based approach [9]. However, the details of the method are proprietary and thus not available for replicable science. Etherscan's headline numbers count addresses with code, not bytecode instances.

Symbolic Execution: Symbolic execution is a very mature discipline as witnessed by the number of literature surveys published. For instance, Baloni et al. [12] provide an overview of the main ideas and challenges in symbolic execution. Păsăreanu et al. [23] offer a survey of trends in symbolic execution research and applications with special focus on test generation and program analysis. Person et al. [24] introduce differential symbolic execution to calculate behavioral differences between versions of programs or methods.

Malware Detection: The prime application of symbolic execution in systems security is malware analysis. Luo et al. [18] use symbolic execution compare code based on behavior. Christodorescu et al. [15] have developed a semantics-aware malware detection framework that uses templates to specify malicious patterns.

Financial Regulation: We are not aware of symbolic execution in tools that support financial authorities in their monitoring and supervision tasks, although some applications stand to reason given the prevalence of algorithmic trading.

In contrast to the above-mentioned work on smart contracts and symbolic execution, we do not aim at generating test cases or show the absence of certain conditions in programs, e. g., integer overflows. We apply symbolic execution to explore all paths through a program and analyze whether that program can be classified based on a given structure, or the presence of certain behavior.

7 Conclusion and Future Work

The idea of this work is to detect Ethereum token systems based on behavioral patterns. We have presented a method and evaluated it as effective using curated ground truth data and a reference method based on signatures.

Both methods have specific advantages. The signature-based approach is simple, but limited to standard-compliant token systems. It is easy to defeat detection by slightly deviating from the standard. The method bears a false positive risk in case of factory contracts or dead code. Quantifying this risk is left as future work. The method can be improved by taking data flow into account.

The behavior-based method does not depend on standard-compliance. It is robust against reordering of parameters or renaming of functions. To which extent it can deal with sophisticated obfuscation is left for future work. The effectiveness of this method demonstrates that symbolic execution is practical on Ethereum.

Both methods fail if the detectable pattern spans over more than one address. If this limitation becomes problematic in practice, the use of concolic execution [12] in conjunction with the current blockchain state is a way to overcome the locality.

In particular the behavior-based method is hand-crafted to the application of token detection. A direction of future work is to generalize the approach by building a domain-specific language in which behavioral patterns can be specified on a high level of abstraction. This would facilitate extensions of our approach to

detect other kinds of behavior, such as smart contracts implementing non-fungible tokens, decentralized exchanges, or gambling services. Evaluating the transactions between the so-identified services would provide the necessary information to draw a map of the Ethereum ecosystem.

Acknowledgments

We like to thank ConsenSys for the work on `mythril`. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 740558.

References

1. Contract ABI Specification. https://solidity.readthedocs.io/en/develop/abi-spec.html, [Online; accessed 5 Sept 2018]
2. EIP 777: A New Advanced Token Standard. https://eips.ethereum.org/EIPS/eip-777, [Online; accessed 18 Sept 2018]
3. ERC-20 Token Standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, [Online; accessed 5 Sept 2018]
4. Manage several contracts with factories. https://ethereumdev.io/manage-several-contracts-with-factories/, [Online; accessed 5 Sept 2018]
5. Minime Token. ERC20 compatible clonable token. https://github.com/Giveth/minime, [Online; accessed 5 Sept 2018]
6. Mythril: Security analysis tool for Ethereum smart contracts. https://github.com/ConsenSys/mythril, [Online; accessed 31 July 2017]
7. Proxy Patterns. https://blog.zeppelinos.org/proxy-patterns/, [Online; accessed 13 Sept 2018]
8. The Parity Wallet Hack Explained. https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7, [Online; accessed 13 Sept 2018]
9. What is an ERC20 token and how to identify them on Etherscan? https://etherscancom.freshdesk.com/support/solutions/articles/35000081107-what-is-an-erc20-token-and-how-to-identify-them-on-etherscan-, [Online; accessed 18 Sept 2018]
10. Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO. No. 81207, Securities and Exchange Commission (July 2017)
11. Crypto-assets: Report to the G20 on work by the FSB and standard-setting bodies. Financial Stability Board (July 2018)
12. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 50 (2018)
13. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns. arXiv preprint arXiv:1703.06322 (2017)
14. Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B.: Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv preprint arXiv:1809.03981 (2018)

15. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy (S P'05). pp. 32–46 (May 2005). https://doi.org/10.1109/SP.2005.20
16. Fröwis, M., Böhme, R.: In Code We Trust? In: Data Privacy Management, Cryptocurrencies and Blockchain Technology, pp. 357–372. Springer (2017)
17. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7), 385–394 (1976)
18. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. IEEE Transactions on Software Engineering **43**(12), 1157–1177 (Dec 2017). https://doi.org/10.1109/TSE.2017.2655046
19. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. ACM (2016)
20. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. arXiv preprint arXiv:1802.06038 (2018)
21. Norvill, R., Awan, I.U., Pontiveros, B., Cullen, A.J., et al.: Automated labeling of unknown contracts in Ethereum (2017)
22. Parizi, R.M., Dehghantanha, A., Choo, K.K.R., Singh, A.: Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains. arXiv preprint arXiv:1809.02702 (2018)
23. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. International journal on software tools for technology transfer **11**(4), 339 (2009)
24. Person, S., Dwyer, M.B., Elbaum, S., Păsăreanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. pp. 226–237. ACM (2008)
25. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Security and privacy (SP), 2010 IEEE symposium on. pp. 317–331. IEEE (2010)
26. Somin, S., Gordon, G., Altshuler, Y.: Social Signals in the Ethereum Trading Network. arXiv preprint arXiv:1805.12097 (2018)
27. Tsankov, P., Dan, A., Cohen, D.D., Gervais, A., Buenzli, F., Vechev, M.: Securify: Practical Security Analysis of Smart Contracts.
https://arxiv.org/pdf/1806.01143.pdf (Aug 2018), [Online; accessed 5 Sept 2018]
28. Underwood, B.: Virtual Markets Integrity Initiative. Office of the New York State Attorney General (September 2018)
29. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (EIP-150 revision).
http://gavwood.com/paper.pdf (2017), [Online; accessed 18 June 2017]
30. Zhou, Y., Kumar, D., Bakshi, S., Mason, J., Miller, A., Bailey, M.: Erays: Reverse engineering ethereum's opaque smart contracts. In: USENIX Security

A Supplemental Result Tables

Table 4: Recall of signature- and behavior-based detection methods against our GTD with lower signature threshold (≥ 3).

	Detected by Sig	**Not Detected by Sig**	
Detected by Behav	88.75% (513)	0.00% (0)	88.75% (513)
Not Detected by Behav	11.25% (65)	0.00% (0)	11.25% (65)
	100.00% (578)	0.00% (0)	100.00% (578)

Table 5: Comparison of signature- and behavior-based detection methods on all bytecode instances with lower signature threshold (≥ 3).

	Detected by Sig	**Not Detected by Sig**	
Detected by Behav	36.06% (40 346)	1.58% (1772)	37.65% (42 118)
Not Detected by Behav	9.36% (10 473)	52.99% (59 291)	62.35% (69 764)
	45.42% (50 819)	54.58% (61 063)	100.00% (111 882)

Table 6: Comparison of signature- and behavior-based detection methods on all bytecode instances with lower signature threshold (≥ 3). Update until block 7 000 000.

	Detected by Sig	**Not Detected by Sig**	
Detected by Behav	35.44% (64 223)	1.28% (2319)	37.72% (66 542)
Not Detected by Behav	13.41% (24 296)	49.87% (90 369)	62.28% (114 665)
	48.85% (88 519)	51.15% (92 688)	100.00% (181 207)

Table 7: Call instructions statistics for different bytecode subsets (mean / median).

Subset	*CALLCODE*	*CALL*	*DELEGATECALL*
-	(0.05 / 0)	(4.08 / 1)	(0.56 / 0)
Behav $\vee$ Sig	(0.05 / 0)	(2.91 / 1)	(0.10 / 0)
Behav $\wedge$ Sig	(0.05 / 0)	(1.34 / 1)	(0.70 / 0)
$\overline{\text{Behav}} \wedge$ Sig	(0.05 / 0)	(9.06 / 6)	(0.20 / 0)
Behav $\wedge \overline{\text{Sig}}$	(0.01 / 0)	(2.20 / 0)	(0.04 / 0)

B ERC-20 Interface Specification

```
contract ERC20Interface {
    // Function Signatures
    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner)
        public constant returns (uint balance);
    function allowance(address tokenOwner, address spender)
        public constant returns (uint remaining);
    function transfer(address to, uint tokens)
        public returns (bool success);
    function approve(address spender, uint tokens)
        public returns (bool success);
    function transferFrom(address from, address to, uint tokens)
        public returns (bool success);
    // Events
    event Transfer(address indexed from,
                   address indexed to,
                   uint tokens);
    event Approval(address indexed tokenOwner,
                   address indexed spender,
                   uint tokens);
}
```

Listing 3: ERC-20 interface in Solidity.

C Documentation of Manual Inspections

Table 8: Five smart contracts of the GTD missed by the signature-based but found by the behavior-based method.

Name	Address	Code Hash	# ERC-20 Functions
LatiumX	0x2f85e502a988af76f7ee6d...	0xf30b6028435e...	3
Pylon	0x7703c35cffdc5cda8d27aa...	0x96858625adfa...	3
Minereum	0x1a95b271b0535d15fa499...	0x65d59c447f7c...	3
All Sports Coin	0x2d0e95bd4795d7ace0da...	0x1c57e11bbd6e7...	3
Golem	0xa74476443119a942de498...	0x35e72568bdaa...	3

Table 9: Random sample of 20 smart contracts in the GTD missed by the behavior-based but found by signature-based method.

Delegation of Bookkeeping

	Address	**Code Hash**
EmphyCoin	0x50ee674689d75c0f88e8f...	0x19780d1f0151fc...
Digix Gold Token	0x4f3afec4e5a3f2a6a1a411d...	0x941fab0f7c206...
FunFair	0x419d0d8bdd9af5e606ae2...	0xe29653f94e73...
Education	0x5b26c5d0772e5bbac8b31...	0xe359bf40848d...
Devery.io	0x923108a439c4e8c2315c4...	0x6b8bff0af6051...
UniBright	0x8400d94a5cb0fa0d041a3...	0x3058c20470fb...

Violation of Definition 1

	Address	**Code Hash**
Ethbits	0x1b9743f556d65e757c4c6...	0xd3f516225294...
Aston X	0x1a0f2ab46ec630f9fd6380...	0xc2b817789336...
Sharpe Platform Token	0xef2463099360a085f1f10b...	0xe0e29e2655db...
FundRequest	0x4df47b4969b2911c96650...	0x519dc5c0384b...
SwarmCity	0xb9e7f8568e08d5659f5d2...	0x88b20869ae32...
Mothership	0x68aa3f232da9bdc23434...	0x63e44909ce93...
Ethfinex Nectar Token	0xcc80c051057b774cd7506...	0x5c7c39e24430...
DaTa eXchange Token	0x765f0c16d1ddc279295c1a...	0xc4bfdc9026f14...
Swarm Fund	0x9e88613418cf03dca54d6...	0x56dd7cb818b4...
MakerDAO	0x9f8f72aa9304c8b593d55...	0xe69355035f77...

Limitations of Symbolic Execution and Taint Analysis

	Address	**Code Hash**
Storiqa	0x5c3a228510d246b78a37...	0x93be59026507...
LocalCoinSwap Cr.	0xaa19961b6b858d9f18a115...	0x88b9c793a727...
LOCIcoin	0x9c23d67aea7b95d80942e...	0x9488b89a5ee6...
TrueUSD	0x8dd5fbce2f6a956c3022b...	0xf447f893b44fd...