In Code We Trust ?
Measuring the Control Flow Immutability of All Smart Contracts Deployed on Ethereum

Michael Fröwis and Rainer Böhme

Department of Computer Science, Universität Innsbruck, Austria

Abstract. Program code stored on the Ethereum blockchain is considered immutable, but this does not imply that its control flow cannot be modified. This bears the risk of loopholes whenever parties encode binding agreements in smart contracts. In order to quantify the issue, we define a heuristic indicator of control flow immutability, evaluate it based on a call graph of all smart contracts deployed on Ethereum, and find that two out of five smart contracts require trust in at least one third party. Besides, the analysis reveals that significant parts of the Ethereum blockchain are interspersed with debris from past attacks against the platform. We leverage the call graph to develop a method for data cleanup, which allows for less biased statistics of Ethereum use in practice.

Keywords: Smart Contract, Trustless, Code Analysis, Call Graph, Ethereum

1 Introduction

Smart contracts are computer programs that encode agreements between parties. They can be settled in virtual currency by decentralized systems of networked nodes. This is advantageous in situations where conventional means of contract enforcement are prohibitively costly, or the parties have no access to a common arbiter or juridical system.

Like for conventional natural-language contracts, a number of conditions must be fulfilled before a party can accept being bound by the terms: the party must understand the content of the contract with the same semantic applied by a potential judge, the integrity of the contract must be guaranteed over its entire lifetime, and the contract must not contain or refer to any terms that can be changed unilaterally after the contract is signed. These three conditions can be mapped to technical requirements (in the same order): access to verifiable source code, immutability of compiled code, and control flow immutability.

If any of these conditions is violated, the party accepting contract terms must trust at least one third party in that the enforcement does not thwart the contract's designated objectives. Ethereum presents itself as a platform for *trustless* smart contracts, and provides means to meet the above-mentioned technical requirements. However, users are free to write smart contracts in a

Turing complete language, so the extent to which smart contracts meet the requirements in practice remains an empirical question.

We set out to answer this question with special emphasis on control flow immutability. We apply abstract interpretation techniques to all bytecode deployed on the public Ethereum blockchain, and synthesize the information in a complete call graph of static dependencies between all smart contracts.

We are not the first to systematically analyze smart contracts on Ethereum. Luu et al. [13] execute 19 366 smart contracts symbolically with the intention to uncover security vulnerabilities, which they find in about 8833 cases. Using source code provided by Etherscan, Bartoletti and Livio [5] manually classify 811 smart contracts by application domain (e. g., financial, gaming, notary) and identify typical design patterns. Norvill et al. [16] propose unsupervised clustering to group 936 smart contracts on the Ethereum blockchain. We are not aware of any prior work that builds or analyzes a call graph of dependencies between smart contracts on Ethereum or a similar platform.

Systemic analyses of all smart contracts on the Ethereum blockchain are impeded by the presence of a significant number of smart contracts originating from attacks against the platform. Therefore, as a second contribution, we propose a cleanup method to pre-process the smart contracts. This is necessary before any meaningful and generalizable measurements of legitimate[1] use can be made.

The paper is organized as follows. Section 2 recalls the vision of *trustless* smart contracts and derives necessary technical requirements for the trustlessness property. Section 3 briefly describes the Ethereum platform and documents our data extraction and analysis methods. Section 4 motivates the cleanup and describes how we accomplished it. Results are presented in Section 5. Finally, Section 6 concludes the paper with a discussion of limitations and implications.

2 Trustless Smart Contracts

We provide some background by reflecting on the notion of trustlessness. Section 2.1 reviews the vision of smart contracts as defined by Szabo [18]. Section 2.2 defines the technical requirements to reach the vision of trustless smart contracts.

2.1 The Vision

Smart contracts are not a very new concept. Szabo introduced the term in 1997. The idea of smart contracts is that many kinds of contractual clauses, in fact every computable clause, can be encoded in logic. That means we can encode contract clauses in computer programs and let the program decide what happens in the course of the contract's lifetime.

This automation of contracts has many advantages, such as reduced transaction cost, less subjectivity, easier auditing, etc. It also facilitates machines

[1] In a slight abuse of legal terminology, this notion of legitimacy includes *everything* except attacks against the platform as a whole.

to enforce in contractual agreements. Think of a car that only starts when the insurance premium is paid. The vision also comprises scenarios where machines enter contractual agreements as partners. Think of autonomous trading agents.

As long as programs encoding contracts are run on local *trusted hard- and software*, and the source code (or some human readable representation) is available for verification, no trust in other parties is needed. This rationale has been around for long. It serves, e. g., as a philosophical pillar of the free software movement.

In the real world, contracts regulate relationships between different parties who may have different interests and objectives, and do not necessarily trust each other. This raises the question of who executes the program encoding a contract? If only one party executes it, the others have to trust in its honesty. If many execute it, what happens if they disagree about the output? A simple approach, also known from paper-based contracts, is the involvement of an impartial *trusted third party*. This can happen in two ways. First, the third party computes the outcome of the contract. Second, whenever a conflict arises, the trusted third party acts as an arbiter. In both cases, all other parties must trust that the third party is fair and abide to its decisions.

Trusted third parties only shift the problem to another hopefully trustworthy party. The declared vision of smart contracts is a system where nobody has to trust a central party. Szabo argues that every algorithmic intermediary can be replaced by a *trustworthy virtual computer*. He contemplates a trustless system using "post-unforgeable transaction logs" and "mutually confidential computation" [18].

Although, in theory, it was known in 1997 that is is possible to build a trustless system based on cryptographic multiparty computation, practical universal systems remained out of reach for lack of efficiency. The advent of blockchain-based systems has demonstrated the existence of a sweet spot that offers more efficient solutions by combining off-the-shelf cryptography with probabilistic distributed consensus protocols. This has led to a renaissance of the ideas behind trustless smart contracts as well as practical freely programmable systems.

2.2 Technical Requirements of Trustless Smart Contracts

Blockchains, or more specifically their underlying consensus protocols, allow to resolve conflicts between parties over a public network without a trusted third party [15]. With the ideas behind Bitcoin, it was possible to build a more efficient version of the *trustworthy virtual computer*, called *consensus computer* [14].

Consensus computers carry out and verify computations over a public network as if they ran on local trusted hardware. Individual parties do not need to trust in any other single party in the system. It is sufficient to make behavioral assumptions about collectives of parties, e. g., that the majority follows the protocol. This brings us back to the desirable situation where we do not have to trust anyone as long as we can verify a program's source code. But now we can run programs that affect many parties, not just one.

If the programs encode contractual clauses, access to the source code is only one of several necessary requirements for trustless smart contracts. We also need *immutability of control flow*. Once we send a smart contract to a consensus computer, it is not supposed to change anymore; just akin to conventional contracts should not be altered after signing. More specifically, if a smart contract has dependencies to other smart contracts on the consensus computer (e. g., by following the common practice of code reuse through libraries, which has been adopted on practical consensus computers), those references should be hard coded in the smart contract. Especially, they should not be determined by (later) input or state of the consensus computer. In other words, once the program is deployed, the control flow must not be changed.

Observe that code immutability is necessary but not sufficient for control flow immutability; which again is necessary but not sufficient for trustlessness, because a smart contract's outcome may also depend on data, which can be unknown at the time of the deployment. In this work, we make first steps towards measuring trustlessness in practice using a heuristic indicator of control flow immutability.

3 Method

Now we describe the data collection and analysis process. Section 3.1 introduces specifics of the Ethereum platform and the terminology needed to understand the analysis (For details, we refer to [3,19].). Section 3.2 explains how we extract smart contract code and build a call graph. Section 3.3 describes how we measure trustlessness of the smart contracts deployed on the public Ethereum blockchain.

3.1 Ethereum in a Nutshell

Ethereum [3,19] can be seen as a generalization of the ideas behind Bitcoin. It is a decentralized system that updates a global state stored in an authenticated data structure called *blockchain*. Besides transferring virtual currency tokens, the Ethereum platform enables users to create smart contracts. Smart contracts are implemented as a special kind of account, which is controlled by program code. More specifically, the program in such a *code account* represents an encoding for arbitrarily complex state transitions. Those state transitions are triggered by sending transactions to the address of the code account. Parameters can be passed in the transaction's `data` field. Code accounts can hold private[2] state in *state variables*. All state variables are persisted in the blockchain and can be modified only by the code of the corresponding code account. Besides code accounts, there exist *user accounts* that are controlled by external parties (i. e., private keys belonging to public keys that define the account). User accounts are best comparable to standard Bitcoin accounts. Both account types can create arbitrary transactions and thus interact with other code accounts, create new code accounts, or transfer virtual currency tokens.

The program in a code account is executed by the *Ethereum Virtual Machine* (EVM), a stack-based virtual machine that executes bytecode. Users typically create smart contracts using a high-level programming language that compiles to EVM bytecode. A popular smart contract language is Solidity [1].

[2] Private refers to scope and write access. It does not imply any confidentiality.

Once a smart contract is compiled to EVM bytecode, it can be deployed to the Ethereum blockchain and thus made available to others. This is done by sending a transaction without a specified recipient to the Ethereum network. The code is sent within the `init` field of the transaction. When the transaction is included in a valid block, every node that processes and verifies the block sees the transaction without recipient. If a node[3] encounters such a transaction, it passes the payload contained in the `init` field to the EVM, which executes it. The output is saved as code of the newly created code account. The address assigned to the code account is determined by the rightmost 160 bits of the Keccak hash of the *recursive length prefix* (RLP) encoded creator address and the account's nonce. The nonce of a code account is incremented as smart contracts are created.[4]

The typical payload of a smart contract creation looks as follows: `(initialization code` $\|$ `code` $\|$ `initialization parameters)`, where $\|$ denotes concatenation. The EVM starts by executing the payload, thus the `initialization code` is executed. The `initialization code` is responsible for setting up initial values of state variables, if needed. The `initialization code` returns the code that will be stored at the newly created address. By default, the `initialization code` loads the `code` part from the payload into memory and returns the memory address and length. But this is just a convention. The `initialization code` could also dynamically build code in memory, or even return garbage. We found that almost all smart contract creations follow the default deployment convention (see Table 1 on page 12).

When a code account is created, its code is part of the blockchain, inheriting the property that it gets harder to modify as more blocks are added to the chain. After several confirmations, the possibility of modification is negligible, therefore the code becomes practically *immutable* over time. (Recall that immutable code does not imply control flow immutability. Measuring the latter is our objective.)

Although code is practically immutable, there is a possibility to disable code accounts. The EVM has an instruction that indicates that the current smart contract should be disabled and the space used by state variables and the account itself can be freed. This operation is called self destruction.[5] After a code account has self destructed, it can still be called, but it behaves as if there is no code available. This means a call to a self destructed smart contract returns without any effect [4]. We call smart contracts that have called self destruct at some point in time *dead smart contracts*. By contrast, all smart contracts that have not called self destruct are called *active smart contracts*.

As mentioned in Section 2.2, source code availability is critical to smart contracts. Whenever the semantic of a smart contract on the blockchain shall be evaluated based on higher-level source code, a verifiable mapping between bytecode, source code, and addresses is needed. We are aware of two relevant services that aim to provide this mapping on a larger scale.

[3] A node that follows the protocol. We make this assumption throughout the paper.
[4] For completeness: also user accounts have nonces. The nonce of an user account is the number of transactions sent by that account.
[5] EVM instruction SELFDESTRUCT

Etherscan[6] is a closed source web application. Users can upload source code that runs on a certain address. The user must provide the exact compiler version and flags used to generate the bytecode at the address. Etherscan then checks if the compiled source matches the bytecode at the supplied address. If it matches, Etherscan saves the corresponding source code and considers this code account a verified smart contract. At the time of writing,[7] Etherscan hosts 1728 verified smart contracts, which is less than 1% of all active smart contracts.

Swarm [2], the other service, is a decentralized peer-to-peer system built as storage service for the Ethereum development stack. It is linked to Ethereum's virtual currency to incentivize honest participation. The idea is as follows: whenever a smart contract is deployed to the blockchain, at the same time one deploys to Swarm a metadata file containing compiler version, flags, and source code. The address of the metadata in Swarm is the hash of its content. This hash is added to the compiled bytecode, hence the bytecode itself refers to its metadata. The verification of the mapping involves the same steps as done by Etherscan, but all metadata is available and can be automatically gathered from Swarm. Although this system sounds promising, and the Solidity compiler already generates metadata and adds hashes to the compiled bytecode, we found that Swarm is barely used to host metadata at the time of writing (see "Swarm metadata" and "Swarm hashes" in Table 1).

3.2 Parsing, Data Extraction and Call Graph Creation

Our goal is to analyze smart contracts and especially the call relationships between them. We extract the call relationship information directly from bytecode because source code is barely availability, as discussed in the previous section.

To extract bytecode from the Ethereum blockchain, we built upon an existing open source blockchain parser project.[8] The project uses the JSON–RPC API, which is part of all major Ethereum node implementations,[9] to extract data from the blockchain. Although this approach is probably slower than parsing the on-disk blockchain format directly, it is more convenient and less error prone.

As already said, code accounts are created by transactions without recipient. To extract the code of all smart contracts, we iterate over all transactions and select those which have no recipient. The code of the smart contract can be found in the `init` field of the transaction [19]. Unfortunately, this approach is limited to smart contracts created by user accounts. Smart contracts created by code accounts are not manifested as transactions in the blockchain itself, but are side effects of the transaction that invoked the contract execution. Interactions between smart contracts (calls) or creations of new smart contracts by other smart contracts are done by so called *internal transactions*. To make internal transaction visible, the execution of the smart contract code needs to

[6] https://etherscan.io/

[7] Accessed on 19 June 2017.

[8] https://github.com/alex-miller-0/Ethereum_Blockchain_Parser

[9] https://github.com/ethereum/wiki/wiki/JSON-RPC

be instrumented. Fortunately, the *parity* client supports a tracing mode[10] that instruments the EVM for this purpose. The tracing API also allows to analyze whether a contract self destructed during an invocation.

All smart contract creations found are written to a MongoDB[11] instance for further processing. We store the bytecode, creation block number, and destruction block number of every smart contract we found. This enables us to filter for active smart contracts in a given time range.

To analyze if a smart contract is trustless, we need to know its call relationships to other smart contracts. We want to distinguish calls to hard coded addresses from calls to addresses provided as input parameter or read from state variables. If a call destination is hard coded, we want to be able to extract it. Our target is to build a call graph of all smart contracts we parsed. To do so, we analyze the bytecode of all smart contracts to extract calls to other smart contracts.

We start by disassembling the bytecode. For that purpose, we use the *evmdis*[12] project as a starting point. Evmdis supports data flow analysis[13] on EVM bytecode. Of interest to us is the *reaching definition* analysis. Informally speaking, reaching definition means that instructions are annotated with a set of variables that are visible to this instruction. For every such variable, evmdis stores the position in code where the variable was assigned last before the instruction. Therefore, evmdis annotates every instruction with the EVM's stack layout before the execution of the instruction. Instead of actual values, this stack layout contains references to instructions that could have produced this stack entry.

We use the reaching definition annotations to find the source of call addresses. To do so, we first search the bytecode for call instructions.[14] A call instruction on the EVM consumes seven stack entries. We are interested in the address, which is stored in the second entry. What we obtain from the annotation is either a static value or a set of n instructions that could have generated the relevant stack entry. If we reach a static value, we are done. Otherwise we follow the links to the instructions that potentially produced the stack entry. We recursively follow all stack positions consumed by the instruction until we either find evidence (in the form of *indicator instructions*) that the address is derived from input parameters,[15] state variables,[16] or we find a constant pushed to the stack.[17]

The fact that we do not evaluate address calculations besides length padding sounds over-simplifying. However, this does not matter in practice for two reasons. First, if the address is not hard coded, we are not interested in its computation because it has no effect on our measurement. Second, we are not aware of any instance where smart contracts use hard coded addresses that are modified before

[10] https://github.com/paritytech/parity/wiki/JSONRPC-trace-module
[11] https://www.mongodb.com/
[12] https://github.com/Arachnid/evmdis/
[13] Via abstract interpretation.
[14] Specifically: CALL, DELEGATECALL, CALLCODE
[15] Reaching a CALLDATALOAD instruction.
[16] Reaching a SLOAD instruction.
[17] Reaching a PUSH20: an address is 20 bytes long.

the call. Address arithmetic is impractical on the EVM because addresses cannot be systematically assigned. Moreover, the most popular high-level language Solidity disallows address computation. Nevertheless, there remains a small risk of wrongly extracted addresses, which we handle later in the analysis.

Another thing to consider are smart contracts that cannot be analyzed at all. This may happen if a code account hosts invalid bytecode. The code provided to the EVM upon creation of a smart contract is not necessarily valid EVM bytecode. We also have to consider the time needed for the extraction of call data. Although smart contracts tend to be rather small (see "Bytecode size" in Table 1), it can take some time to follow all code paths to extract the address origin. To keep the time to extract data manageable over all code accounts, we limit the runtime of the extraction process to 30 seconds per smart contract. A total of 539 smart contracts where not included in our dataset because of this restriction.

With the calls extracted from the bytecode of all code accounts, we can build the desired call graph. The graph is generated by iterating over all active smart contracts in our database.[18] Vertices represent smart contracts and are annotated with the address and the block number of the smart contract creation. Edges are directed and represent call relationships (from caller to callee).

Our graph contains five special nodes. Not all active smart contracts have corresponding transactions in the blockchain: the EVM supports four[19] hard coded smart contracts. Our fifth special node is UNKNOWN. It is used whenever a smart contract has calls where no hard coded address could be extracted. In the later analysis, we treat UNKNOWN as *not* trustless because the code is not known, whereas the hard coded addresses of the EVM are considered trustless.

Another possible point of failure is the extraction of hard coded addresses, especially if arithmetic on addresses is involved. To prevent wrong addresses, we check before the creation of new edges if the called address belongs to an active smart contract. If so, we insert the edge. Otherwise we check if the address is in the set of dead smart contracts. If so, we can be almost certain that the smart contract is no longer active. If not, we either extracted a wrong address or the user deployed a smart contract with a wrong address. We ignore calls to wrong addresses as well as calls to dead smart contracts for our analysis of trustlessness. As calls to addresses that do not host code return without effect, we consider such calls as trustless. Even though we do distinguish between dead and wrong to build our call graph, we learned how many smart contracts refer to smart contracts that are not longer *active* (see "Calls to dead addresses" in Table 1).

3.3 Measuring Trustlessness

The call graph is the basis for analyzing the trustlessness of smart contracts.

[18] Using the *networkX* graph library, `https://networkx.github.io/`
[19] at the addresses 0x1, 0x2, 0x3, 0x4

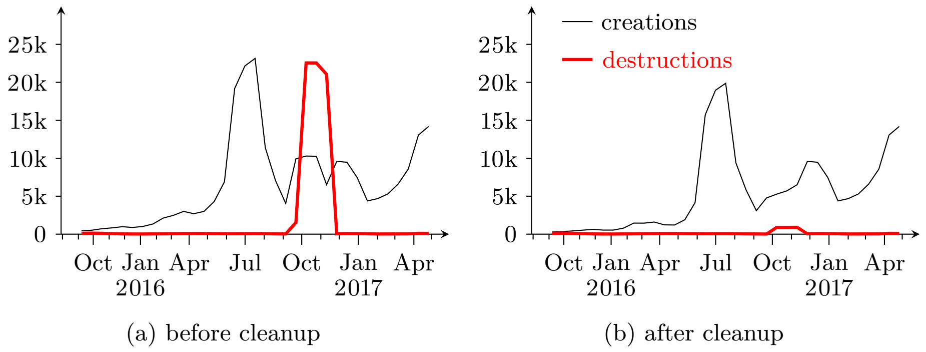


Fig. 1: Creation and destruction rate (moving window over 100k blocks)

A simple indicator of trustlessness can be defined as follows: Let $G = (V, E)$ be the directed call graph and $\text{succ}(v) = \{\ w \mid (v, w) \in E\}$ with $v, w \in V$. Now,

$$\text{trustless}(v) = \begin{cases} \text{true}, & \text{if } \text{succ}(v) = \emptyset \\ \text{false}, & \text{if } \textsc{Unknown} \in \text{succ}(v) \\ \bigwedge_{s \in \text{succ}(v)} \text{trustless}(s), & \text{otherwise.} \end{cases}$$

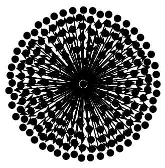
Informally, a smart contract is trustless if and only if all calls in its dependency tree have hard coded addresses, hence all code that a smart contract can execute is fixed upon deployment of the smart contract.

A disadvantage of this recursive indicator is that it does not terminate on cyclic graphs. Note that Ethereum makes it difficult (but not impossible) to produce cycles. Every smart contract is deployed in its own transaction. The address is returned after the smart contract is deployed. To introduce cyclical dependencies, one has to deploy a smart contract with a reference to a smart contract that is not yet deployed. Thus, one must be able to predict the address a smart contract is deployed to. This is possible because the address creation is deterministic. Therefore, with some effort, it is possible to deliberately create dependency loops. We found that loops do exist (see Table 1). To handle cyclic dependencies, we use a set that tracks already visited vertices when calculating the trustlessness indicator and stop the recursion. A vertex encountered twice signals that there is a cycle in the dependency graph. We consider smart contracts with cyclic dependencies as not trustless in our analysis.

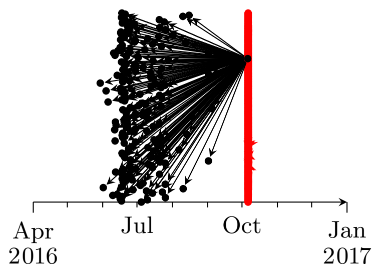
4 Call Graph Cleanup

Extracting statistics from the raw Ethereum call graph can be misleading as the data is interspersed with attack debris. Here we report our cleanup procedure.

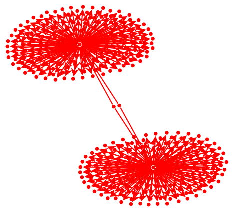
The Ethereum platform has been target of multiple attacks in the last couple of months [9,10]. Two attacks are especially notable because they led to hard forks of the Ethereum blockchain [8,12]. One of them is the infamous DAO



(a) One of 45 "death stars": subgraphs of 171 active smart contracts attributed to the attacks



(b) Creation process of the components



(c) One of two components deployed in October 2016

Fig. 2: Call graph perspective on the 2016 DoS attack (selected components)

Attack. The DAO (*Decentralized Autonomous Organisation*) is a blockchain-based venture capital fund, designated to fund new Ethereum projects. Due to a bug, an attacker was able to steal coins worth roughly 60 million USD, at the time of the attack [9]. Besides the ominious fork, the DAO Attack has left no obvious traces in the call graph of smart contracts.

This cannot be said of the other major attack on Ethereum, a DoS (*Denial of Service*) attack which unfolded in October 2016. An attacker flooded the Ethereum network with transaction spam, using various strategies to overload and slow down the network [17]. To prepare the attack, the attacker deployed thousands of smart contracts that called other smart contracts in a tree structure. The addresses of the called smart contracts are hard coded in the calling smart contract. The leaves of the tree carried out the actual attack, e. g., by cheap contract creation via self-destruct [7].

One example of such an attack can be seen in block 2 416 461,[20] where one invocation of a smart contract caused 15 000 others to call self-destruct. The attack is easily observable in our dataset. In Figure 1a, the spike in contract destructions in October 2016 as well as the spike in contract creations July and October 2016 are indicators of the attack. We also observed many smart contracts created and destructed in the same block, which is atypical for non-malicious uses (see "With zero lifetime" in Table 1).

The volume and patterns of smart contracts involved in the DoS attack present a significant bias for our analysis. To clean our data, we asked the Ethereum community for help and were provided with a set of 99 addresses that where *directly involved* in the attacks. 34 of them are code accounts. Directly involved means they were used for the actual attack on the network. We know that the attacker created far more code accounts than he actually used in the attack. Ideally, we want to filter *all* smart contracts created in preparation of the attack. Therefore we started to look for patterns to identify suspect code accounts.

[20] See TxHash:
0xf435a354924097686ea88dab3aac1dd464e6a3b387c77aeee94145b0fa5a63d2

One pattern we filter are code accounts created and destroyed in the same block around the time of the attacks[21] (see "With zero lifetime" in Table 1).

By looking at connected components in the call graph, we found 45 star-shaped subgraphs with 171 vertices each, sharing a very similar structure: one master code account deployed in October called 170 sub-code accounts deployed earlier. Figure 2 illustrates this behavior for one selected *death star*. All of them were created between July and October 2016 by the same address.[22] This address is in the set provided by the Ethereum community. As illustrated in Figure 2, the attackers also deployed at least two large connected components in October 2016.

Our final set of smart contracts to be excluded from the analysis is composed of all 34 smart contracts flagged by the Ethereum community, all code accounts identified by our heuristics as well as their direct and indirect neighbors in the directed call graph. We obtain a total of 95 791 code accounts that are potentially related to the attack, of which 30 668 are still active on 01 May 2017.

Observe from Figure 1b that the cleanup largely removed the spikes in smart contract creation and destruction. There remains a suspicious spike in smart contract creations in July 2016. We conjecture that most of the smart contracts created in July 2016 are also related to the attacks. But due to a lack of evidence and the risk of false positives, we decided to not filter our data further.

5 Results

We study the Ethereum main chain from the day of its inception until 01 May 2017.[25] We report results *before* and *after the cleanup*, as described in Section 4. We have 225 000 active smart contracts before cleanup and 194 332 after cleanup.

5.1 Stylized Facts

Table 1 summarizes our quantitative results. General statistics include the mean and median of bytecode sizes ("Bytecode size"). Observe that the cleanup reduced both mean and median bytecode sizes. This means that the smart contracts used for the attacks were exceptionally large. We also measured the mean and median lifetime of smart contracts ("Lifetime"). It is easy to see the bias introduced by the attacks in the *before cleanup* column. The median smart contract lifetime before cleanup is 0. This is a consequence of the 52 689 smart contracts created and self destructed in the same block ("With zero lifetime"). This is a significant bias in a set of 69 875 destructed smart contracts in total. If we look at the results after cleanup, we see that mean an median are about the same, approximately five

[21] From 01 May 2016 until the hard fork on 18 Oct 2016.

[22] Address: 0x1fa0e1dfa88b371fcedf6225b3d8ad4e3bacef0e

[23] Created and self destruct in the same block.

[24] Address: 0x938162cc5d6f4fc5d3f9edec18c93c5379d56062

[25] Block number: 3 633 433

Table 1: Summary of active smart contracts in Ethereum until 01 May 2017

Concept	Statistic	Analysis mode	
		before cleanup	after cleanup
Smart Contracts			
Total active smart contracts	#	225 000	194 332
Bytecode size (bytes)	mean	1078	775
	median	578	542
With zero lifetime[23]	#	52 689	65
Lifetime of dead contracts (blocks)	mean	10 061	40 687
	median	0	39 001
Violate deployment convention	#	6	6
Source Code Availability			
Swarm hashes	#	29 496	29 480
Swarm metadata	#	14	14
Dependencies			
Smart contracts with calls	#	196 176	167 110
Smart contracts without calls	#	25 456	24 430
Could not analyze dependencies	#	3368	2792
Smart contracts with self loops[24]	#	1	1
Smart contracts with loops	#	30	7
Calls to dead addresses	#	14 196	1712
Calls to wrong addresses	#	6983	5487
Trustlessness			
Trustless smart contracts	#	122 375	119 493
	% of total	54.4	61.5

and a half days.[26] As mentioned in Section 3.1, we found only 6 smart contracts that violated the default deployment convention.

In terms of source code availability, we find that about 13 to 15 percent of all active smart contracts contain references to Swarm metadata (*Swarm hashes*). But only 14 (in absolute terms!) actually host metadata (*Swarm content*). We conclude that Swarm is not a reliable source of source code for the study period.

When it comes to dependencies, we find that most smart contracts have calls (dependencies) to other smart contracts ("smart contracts with calls","Smart contracts without calls"). Our cleanup filtered very few smart contracts without calls. Many of the smart contracts used in the attacks create big dependency trees to amplify the attack, thus the result is not surprising. The row "Could not analyze dependencies" reports the number of smart contracts we were not able to extract dependency information from. Both values are around 1.5 percent of the total active smart contracts. As mentioned in Section 3.3, our call graph contains cyclic dependencies. Interestingly, most of the loops we found are related to the attacks, only 7 are left after cleanup. In Section 3.2 we described how we deal

[26] Assuming 12 seconds block time.

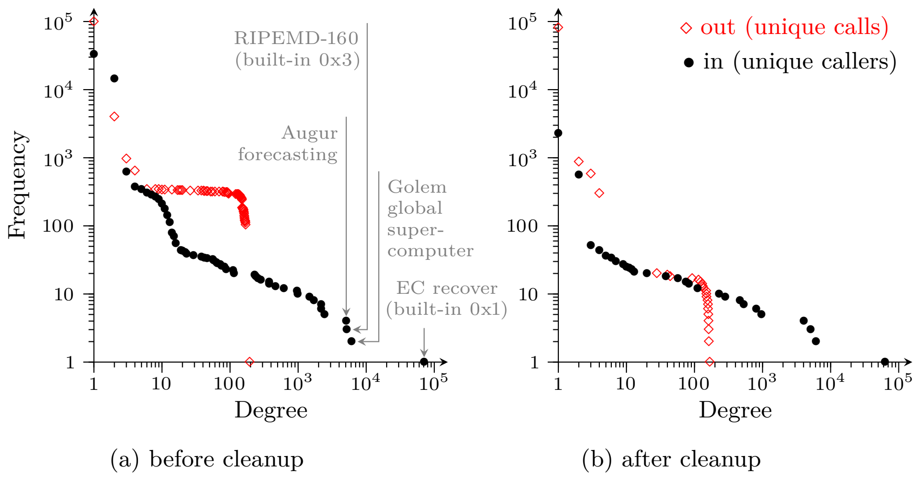


Fig. 3: Cumulative degree distributions of the directed call graph

with calls to wrong and dead smart contracts. It is interesting to see that after cleanup, the references to self destructed smart contracts decrease significantly (“Call to dead addresses”).

Another way of looking at dependencies is the degree distribution of the call graph depicted in Figure 3. The in-degrees follow a typical Pareto shape in the cumulative log-scaled representation both before and after cleanup. We annotate the smart contract that are called from the highest number of other smart contract in Figure 3a. Two of them are hard coded smart contracts (cf. Sect. 3.2). The distribution of out-degrees is visibly more affected by the attacks. The singularity around degree 170 can be attributed to the death stars. It disappears largely, but not completely, after cleanup. We conjecture that we might have missed 10–20 suspicious smart contracts after successfully removing several hundreds. Comparing both distributions highlights the importance of removing attack debris from the Ethereum call graph.

5.2 Trustlessness

In Section 2.2 we described the requirements for trustless smart contracts. Some of the requirements are supported by the design of the Ethereum platform, such as distribution, consensus, fairness, and determinism. The extent to which these requirements are met depends largely on behavioral assumptions about the participating nodes, which are beyond the scope of this paper. Here we concentrate on the immutability of the control flow, a necessary requirement for trustlessness and a property of individual smart contracts.

Figure 4 compares active smart contracts to trustless active smart contracts using the trustlessness indicator presented in Section 3.3 over time. We find that, before cleanup, 54 percent of all active smart contracts in our sample are trustless in principle. The ratio raises to 62 percent after cleanup.

In other words, two out of five smart contracts deployed on Ethereum do require trust in at least one third party who, in principle, can alter the control flow of the program that enforces an agreement after it is committed to the blockchain. This is not necessarily concerning, but a remarkable observation against the backdrop of trustlessness being framed as the key benefit of smart contracts and blockchain-based systems in general over conventional (centralized) infrastructures. In simple terms, there remains a gap between vision and practice.

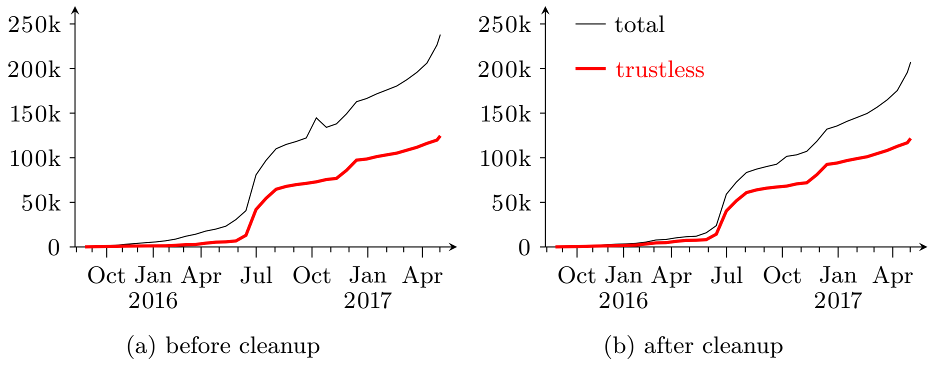


Fig. 4: Active smart contracts compared to active trustless smart contracts

6 Discussion and Conclusion

We have developed a measurement approach for the trustlessness of smart contracts and applied it to all the smart contracts on Ethereum. Two out of five smart contracts we found on Ethereum are not trustless according to our call graph-based indicator. This means it is hard or even impossible for users to verify these smart contracts. We also motivated the need for data cleanup when analyzing smart contract properties in order to avoid biases introduced by the large scale attacks against the Ethereum platform. Accordingly, we propose a cleanup strategy that leverages the call graph. This allows us to produce unbiased summary statistics of legitimate use of Ethereum, including indicators of bytecode size, smart contract lifetime, and source code availability.

Our approach has some limitations. It is based on the extraction of hard coded addresses from bytecode. Although it seems to be robust in practice, it is heuristic in nature with the possibility of extracting wrong information. The apparent robustness also depends on the usage conventions on the Ethereum platform. For example, if languages that allow address arithmetic gain popularity, the current approach will resolve fewer dependencies. Other limitations persist independent of the extraction of code dependencies. Currently, our approach is blind to data dependencies. Those can range from simple deactivation flags, which differ from self destruct only in the gas impact, to emulations of Turing equivalent machines inside the smart contract. This means that even if a smart contract is trustless according to our indicator, it can still encode agreements where trust in individual parties is needed. Tackling data dependencies is hard, because many use cases of smart contract need them.

Furthermore, we do not consider gas restrictions at the moment. Callers can limit the amount of work a callee can do by restricting the gas supply of the callee, this directly influences the amount of trust needed between parties.

Let us conclude with a broader outlook: Ethereum promises to fulfill the vision of trustless smart contracts. However, trustlessness is not only a property of the platform, but also of every individual smart contract. Our measurements show that many smart contracts violate necessary conditions for trustlessness in practice. We assume that many of these violations are the result of a lack of awareness rather than intentional. This raises the need for tooling that helps to avoid such mistakes, or at least increases awareness for the subject. Static analysis of source code (for example Solidity) could be used to prevent the most common trustlessness violations, such as calling addresses obtained from parameters or state variables. Furthermore, there is a relation to the verifiability of smart contracts: existing formal verifiers for Ethereum [6,11] need certain assumptions, which seem to be implied in our notion of trustlessness. Therefore, the subset of trustless smart contracts is more amenable to formal verification than general code for the Ethereum platform.

Acknowledgments

We like to thank Dr. Christian Reitwießner for answering many Ethereum related questions, Dr. Arthur Gervais for pointing out the existence of the parity tracing API and the excellent parsing script, Nick Johnson for the creation of the *evmdis* disassembler, Martin Holst Swende for the additional material on the Ethereum DoS attacks, and Clemens Brunner for proofreading and discussions. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 740558.

References

1. Contracts - Solidity 0.4.12 documentation.
   http://solidity.readthedocs.io/en/develop/, [Online; accessed 12 June 2017]
2. Contracts - Solidity 0.4.12 documentation - Swarm.
   http://solidity.readthedocs.io/en/develop/miscellaneous.html#contract-metadata, [Online; accessed 12 June 2017]
3. Ethereum Homestead Documentation.
   http://ethdocs.org/en/latest/, [Online; accessed 19 June 2017]
4. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts. Tech. rep., Cryptology ePrint Archive: Report 2016/1007 (2016)

5. Bartoletti, M., Pompianu, L.: An empirical analysis of smart contracts: platforms, applications, and design patterns. arXiv preprint arXiv:1703.06322 (2017)
6. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Short Paper: Formal Verification of Smart Contracts
7. Buterin, V.: A state clearing FAQ. https://www.reddit.com/r/ethereum/comments/5es5g4/a_state_clearing_faq/ (Nov 2016), [Online; accessed 18 June 2017]
8. Buterin, V.: Hard Fork Completed. https://blog.ethereum.org/2016/07/20/hard-fork-completed/ (Jul 2016), [Online; accessed 18 June 2017]
9. del Castillo, M.: The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. http://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/ (Jun 2016), [Online; accessed 18 June 2017]
10. Hertig, A.: So, Ethereum's Blockchain is Still Under Attack.... http://www.coindesk.com/so-ethereums-blockchain-is-still-under-attack/ (Oct 2016), [Online; accessed 18 June 2017]
11. Hirai, Y.: Formal verification of deed contract in ethereum name service (2016)
12. Jameson, H.: FAQ: Upcoming Ethereum Hard Fork. https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/ (Oct 2016), [Online; accessed 18 June 2017]
13. 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)
14. Luu, L., Teutsch, J., Kulkarni, R., Saxena, P.: Demystifying incentives in the consensus computer. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 706–719. ACM (2015)
15. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
16. Norvill, R., Awan, I.U., Pontiveros, B., Cullen, A.J., et al.: Automated labeling of unknown contracts in Ethereum (2017)
17. Swende, M.H.: The Shanghai Attacks. https://edcon.io/ppt/one/Martin%20Holst%20Swende_The%20'Shanghai%20'Attacks_EDCON.pdf, [Online; accessed 19 June 2017]
18. Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2(9) (1997)
19. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (EIP-150 revision). http://gavwood.com/paper.pdf (2017), [Online; accessed 18 June 2017]