Universität des Saarlandes

Department of Computer Science

Master degree in Computer Science

Master Thesis

A Benchmark for Formal Verification of Ethereum Smart Contracts

Supervisor:
Prof. Dr. Matteo Maffei

Graduant:
István András Seres

Advisors:
Clara Schneidewind, Ilya Grishchenko

Reviewer:
Giancarlo Pellegrino

Aug, 2017

University of Trento

Department of Information Engineering and Computer Science

Master degree in Computer Science

Master Thesis

A Benchmark for Formal Verification of Ethereum Smart Contracts

Supervisor:
Prof. Fabio Massacci

Graduant:
István András Seres

Academic year 2016-2017

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Contents

List of Figures

List of Tables

Acknowledgements

In the first place I am grateful for the help, patience and support of Clara Schneidewind and Ilya Grishchenko. Without them this work would have been much harder to accomplish: they helped me to develop a better understanding of formal verification, static analysis, semantics and much more.

I am thankful for the availability of Matteo Maffei, for his supervision and for the opportunity to make this challenging and fascinating research and work together with his team.

Let me mention by name some of my fellow students from EIT Digital: Giovanni de Francesco, Ábris Nagy, Dorka Palotay and Jurian van Dalfsen. They were who inspired me by their endurance and passion and did not let me down on the way. Without their support and friendship this work would not be here.

Last but not least I am truly thankful to my parents, family and friends for their support.

1 Introduction

Although cryptocurrencies are quite a young branch of cryptography, more and more attention is payed on the development of such new digital assets [Coi]. The main research question of cryptocurrency as a field is that whether it is possible to create and maintain a secure, decentralized electronic cash system, which does not rely on any trusted third parties (e.g. banks, financial institutions). For a long time it was highly doubted by many researchers that it was even possible.

One of the challenges in designing cryptocurrencies is that we would like to retain the very same characteristics of classical cash systems also for electronic cash systems. However, this is more demanding to achieve without the involvement of trusted third parties. There had been dozens of unsuccessful proposals. Each and every of them failed because of various reasons. Some of them were not entirely decentralized (they relied on some trusted parties) just like B-money depends on a trusted timestamping service that sign off on the creation or transfer of money [Dai98]. Many proposals could not solve the double-spending problem: it is an error in a digital cash or cryptocurrency scheme in which the same single digital token is spent twice or more. GoofyCoin, a rudimentary proposal [Nar16] is one of many cryptocurrency proposals susceptible of double-spending attacks. However in case of fiat currencies double-spending is not even a problem, namely it is impossible to spend the very same coins more than once at the same time. Other challenges include that nobody should be able to create money out of thin air, there should also be measures for participants to check the validity of transactions taking place in the network.

The promise and vision of crypto currencies was a secure, publicly verifiable, decentralized digital cash system. In such a network one would have been able to send money with low transaction costs and with nearly instant settlement while retaining the security of classical financial systems. Benefits were charming for bankers, traders and more; according to a study, crypto currencies can potentially save banks tens of billions of dollars a year [You17]. However, this bright future was far in the mid 2000s when there were only broken proposals at hand.

In 2008, at the highest peak of the financial crisis, an alternative cash system, the celebrated Bitcoin was created and launched [Nak08]. Back then and also today a lot of people think of Bitcoin as an alternative cash system or digital asset where they can securely hold their investments. The great breakthrough was that Bitcoin could deliver a **secure** electronic cash system under minimal assumptions. Bitcoin remains secure if the majority of the network's hashpower belongs to benevolent people. It was proved in [GP17] that this assumption is indeed sufficient to solve the double-spending problem in the Bitcoin network. Other highly popular advantages are low transaction fees and speed in comparison with classical financial institution systems. Bitcoin also provided a purposefully non-Turing complete stack-based language to express the conditions of receiving Bitcoins. This means that one can determine the conditions which should hold if Bitcoins are released and sent to another party. These conditions can be considered as a simple form of smart contracts.

Smart contracts are computer protocols, which facilitate to execute the conditions of a contract. Smart contracts aim to reduce transaction costs and fraud loss by automatically executing contractual conditions and excluding - partially or totally- human intervention [Sza94]. Bitcoin allows writing only rudimentary smart contracts due to the non-Turing-completeness of the built-in scripting language.

In contrast with Bitcoin, Ethereum comes with a built-in quasi-Turing complete programming language, which enables anyone to write more expressive smart contracts [Woo14]. These smart contracts are computer programs with special characteristics. They are executed in a distributed fashion in a peer-to-peer (P2P) network. Anyone is able to see the bytecode of the contract which is posted on the immutable blockchain. Moreover, all the participants of the network can interact with the contract, more precisely calling functionalities of it or sending money to it. Unlike other computer programs, webapps or mobile applications, smart contracts operate on and deal with real money. This means that a compromise in a contract might cause severe loss of crypto money and/or real money. The most famous failure was the one of the DAO 1.0, which served as a decentralized crowdfunding platform until it got hacked in June, 2016. The attack exploited a programming bug in the contract and led to the loss of US$50-million worth of crypto token. Therefore the analysis of the security and correctness of these programs gained supreme importance.

Several tools were proposed to analyze the behaviour of Ethereum smart contracts ([Luu16], [Zik16]). These tools are also applied to catch and prevent bugs in contracts. It is extremely important to be sure that a contract is indeed secure, when a tool says so. If bad behaviour is not ruled out and contract is deployed, there is essentially no way to fix bugs due to the properties of the Ethereum platform (see Section 2). As new tools are emerging [Kar16] there is a need to compare and test these tools on a standard test suite. However the lack of a common benchmark makes assessing the performance of analysis tools and comparing them quite problematic. A common benchmark would make smart contract security transparent and help developers in choosing the most suitable analysis tool for their application scenario. Such a benchmark would also be able to point out the limitations of analysis tools and would assist in interpreting their results. Moreover developers would be more confident about what to expect and not from Ethereum smart contract analysis tools. Therefore this thesis contributes to the Ethereum community a benchmark for formal verification of Ethereum smart contracts.

2 Ethereum: the distributed world computer

As cryptography aims to design and develop techniques for secure communication in a trustless manner, that is, without relying on any (trusted) third parties, in the same way cryptocurrency as a subfield of cryptography seeks to design and provide decentralized, trustless financial/banking systems by applying many cryptographic protocols and tools. In contrast with cryptography, in cryptocurrency settings not only information has to be transmitted in a secure way, but also real money or other digital assets. The design and production of such systems is so challenging that many researchers thought it is not even possible [Nar16]. There were numerous proposals, however up to 2008 all of them failed either because they were not decentralized or because they were not cryptographically secure.

In 2008, at the highest peak of the financial crisis, there was a tremendous breakthrough in the cryptocurrency scene: the invention of Bitcoin [Nak08]. Bitcoin was the first (and still functioning) decentralized, cryptographically secure digital currency. It was a compelling example of the first decentralized financial system, which operated in a peer-to-peer network without the intervention of any central/federal bank or any other entity. In the Bitcoin network money (bitcoin, denominated as BTC) can be sent over the network in a secure way. Accounts and balances are stored in a publicly verifiable decentralized ledger, called blockchain. Shortly after the inception of Bitcoin, many Bitcoin-like cryptocurrencies were proposed using many of Bitcoin's technologies, especially the blockchain.

Ethereum is also one of these successors. However Ethereum is more than that in the sense that in the Ethereum network one can not only send money securely, but also can execute distributed programs called smart contracts. These decentralized programs are the essence of the invention Ethereum brought in the cryptocurrency community. At this point these notions and names might be vague for the reader, so in this section we give a detailed description of these important concepts.

2.1 The Ethereum blockchain

As of today, blockchain is the backbone of most of the decentralized cryptocurrencies (remarkable exception includes Ripple [Dav14]). First it was invented by Satoshi Nakamoto and described in the Bitcoin white paper [Nak08]. It is a distributed, immutable database which serves as a ledger for the cryptocurrency containing all the transactions which allow to determine the accounts and balances of every user. The distributed and open nature of it allows decentralized systems to be built on top it. More specifically, blockchain is a data structure, which is almost identical to linked lists with one crucial difference: blockchain is built with hash pointers instead of pointers. In Figure 1 "data" denotes transactions in case of Bitcoin and Ethereum as well. However in case of Ethereum transactions are more complex than that of Bitcoin in the sense that they do not only transfer value (bitcoin, ether or other tokens) but also cause code to be executed or new accounts to be created.

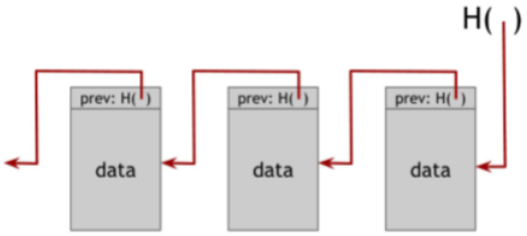

Figure 1: Blockchain: the data structure

The first block, which does not have an ancestor (implicitly does not have the *prev_hash* field) is called the genesis block. The genuine idea of using hash pointers instead of pointers enables the data structure to provide data integrity. A simple pointer in a linked list only tells us where to find the previous block, while a hash pointer also tells us whether the blockchain was modified or not. Each block built on top of the blockchain reinforces all the previous ones. In cryptocurrency applications of blockchain (see later), blocks which are further from the head of the blockchain are more likely to remain unchanged, since new branches of the blockchain might evolve during time. This property is called settlement finality [But16b]. The blockchain is typically stored at each member's hard disk in the P2P network. As new members join to the network they can download the blockchain from older members using the P2P network enabling them to verify the validity, authenticity and integrity of transactions. This way there is no single entity everyone should trust (like a central or federal bank), rather trust in the cryptocurrency is acquired by the possibility of public verifiability. Every member is able to commit a new block to the blockchain if she is able to solve the following inequality:

$$H(nonce||prev\_hash||data) < target,$$

where H is a cryptographically secure hash function, *nonce* is the value to be found, *prev_hash* is the hash of the previous block, *data* denotes the transactions put into the block and *target* is a dynamically changing value, which is set according predefined protocol rules as a function of previous *target* values. This general inequality is called a proof of work (PoW) puzzle, which is the core of PoW consensus algorithms. Consensus algorithms lets network participants to agree upon the actual state of the blockchain (the ledger of the cryptocurrency) in an organized way. Currently proof of work consensus algorithms are used by vast majority of cryptocurrencies such as Bitcoin and Ethereum. Other important alternative consensus algorithms include proof of stake algorithms [But16a].

If H is a "good" hash function (cryptographically secure), then for every fixed element y from the image space it is computationally hard to find x from the domain such that $H(x) = y$. Therefore the only way to solve this inequality is to bruteforce the *nonce* field of the block. Practically speaking one should solve a partial hash collision problem in order to create a new block on the blockchain, which in case of a cryptographically secure hash function is computationally hard. The action of finding new blocks is called mining. Mining is a probabilistic action where the success probability is proportional to the number of hashes one can perform at a given time; this is the so-called hash rate (hashes/second). The incentive behind mining is that new coins are created and given to the peer who finds the next block. This design choice of most cryptocurrencies keeps incentivized nodes to maintain the blockchain via finding new blocks and efficiently distributes coins to peers. In most cryptocurrencies mining is the only way to create money, therefore distributing coins

this way is also called minting.

In the Ethereum network, blocks are mined on average in every 15 seconds [Ethb]. In case that several blocks are found at almost the same time, the blockchain is split into different branches. This event is called a (regular) fork. Individual miners can decide in which branch they are continuing the mining. In most cryptocurrencies, they should choose the branch where there is the most accumulated difficulty (the sum of the leading zeros in target values) down to the genesis block. Due to this protocol rule splitting is - sooner or later - resolved, since it is an undesired behavior of the blockchain. A (regular) fork is especially unsafe and undermines the trust of the blockchain if it is done intentionally. An entity who owns more than the 50% of the hash rate of the network can potentially rewrite the blockchain, since finds blocks faster than the whole network together. This is clearly catastrophic since this entity could rewrite transactions and could decide which transactions she includes in the blocks. This attack is called the 51% attack. A remarkable paper showed that even if a party owns the 1/3 of the hash power of the network can mount a successful attack on the blockchain with a selfish mining strategy [ES13].

Forks can also be introduced willingly to the network as a mean of updating the protocol. In contrast with centralized computer systems upgrading a cryptocurrency protocol is more cumbersome, since in a decentralized architecture multiple nodes including miners, developers and users have to agree on protocol rules. There are two main type of protocol changes which can be carried out.

2.1.1 Soft fork

In case of a soft fork, changes made to the protocol are backward compatible. As a result of a soft fork, some previous valid transactions or blocks might be considered as invalid by upgraded nodes. If non-upgraded nodes continue to mine blocks they could potentially be rejected by the upgraded ones, causing upgraded and non-upgraded nodes to mine temporarily on two different branches of the blockchain. This is why soft forks need a majority of hash power in the network to resolve such forking events [Cas17].

2.1.2 Hard fork

A hard fork introduces new protocol rules which are not compatible with old protocol rules. One can think of a hard fork as an extension of rules [Cas17]. Non-upgraded nodes will consider all the new transactions and blocks invalid, therefore if they want to join to the new chain, they have to upgrade their node. During a hard fork a permanent divergence in the blockchain is taking place if non-upgraded nodes decide to stay and mine on the old chain. If a significant miner, developer and user base is behind both of the chains, then two independent and incompatible versions of the same coin are born. A perfect case study of how a community can split over rules was the hard fork in the Ethereum community taken place in summer, 2016 at block no.1920000 [Foub] when the Ethereum blockchain was forked and Ethereum Classic was emerged [Cal]. In the following our discussion will strictly stick to Ethereum.

Ethereum's blockchain is not just a distributed ledger, which only stores transactions going around in the P2P network, but there are also applications built on top of the Ethereum blockchain. These decentralized applications (dapps; one can think of dapps as web applications whose back end is written on the blockchain in the form

of smart contracts) are executed inside the Ethereum Virtual Machine (EVM). EVM describes how EVM bytecode gets executed. In order to simplify the development of dapps there are several programming languages which are designed to target this virtual machine. These high-level programming languages are used to develop these applications (i.e. smart contracts). There are currently 3 of them: LLL, Serpent and the most popular, Solidity. The above mentioned high-level languages are compiled down to bytecode which is stored on the blockchain as part of transactions and can be executed on demand. Due to the integrity of the blockchain, not only transactions, but also smart contracts which are encoded in transactions are immutable in Ethereum. This will have serious consequences regarding smart contract security.

2.2 Ethereum Virtual Machine

As we have just anticipated the Ethereum blockchain contains smart contracts and their bytecode encoded in transactions. These smart contract bytecodes are executed in the execution environment called Ethereum Virtual Machine (EVM) and they can alter the system state.

The EVM is a quasi-Turing-complete machine, where "quasi" comes from the fact that the number of computational steps is intrinsically bounded from above by a parameter called gas, which limits the total amount of computation done. For every block there is a block gas limit set by the miner but block gas limit is restricted to be in a certain range. Furthermore every transaction specifies a gas limit that needs to be below the block gas limit. Every Ethereum network participant executes bytecodes locally although miners are the ones who decide upon the blocks to be appended to the chain and reach consensus about the effect of the smart contracts execution on the global state. Indeed, if it would be possible for one to craft a smart contract which runs forever, then it would take also forever to calculate the effect of this contract. Therefore such a contract would act as a denial-of-service (DoS) attack, implying that quasi-Turing-completeness is inevitable.

EVM has a simple stack-based architecture with word size of 256-bit. The stack has a maximum size of 1024. Memory is a byte-array. Memory starts off zero-size, but can be expanded in 32-byte chunks by simply accessing or storing memory at indices greater than its current size. "Uninitialised" memory can be read as well and can therefore also be seen as initialised with all zeros. The motivation behind modelling memory as an array that gets extended dynamically rather than a mapping/store is that allocating new memory chunks consumes gas. In contrast with memory, storage is a key/value store, where keys and values are both 32 bytes. A crucial difference between storage and memory is that after contract's execution stops the content of memory is erased, however storage is permanent. Each created account has its own persistent storage, which serves as a storage for contract-related data. Storage is saved by the blockchain in the sense that on the execution model level executing the appropriate operation codes (in short: opcodes) influences the system state such that also next time if contract executes some code, it has access to all the data it previously stored into its 'virtual' storage area of the executing account. In both cases, initially all the values of memory and storage are set to zero.

Fees - denominated in gas - are charged under three circumstances:

1. for the computation of the operation,
2. to form the payment for a subordinate message call

3. for an increase in the usage of the memory.

The EVM bytecode set consists of numerous different opcodes. For example there are several arithmetic operations like addition, multiplication or stack operations for pushing values to the stack (PUSH) or popping them from the stack (POP) etc. After executing each opcodes one has to pay certain predefined amount of gas. Computationally less expensive opcodes (eg. arithmetics) of the EVM cost less, while more expensive ones (eg. calls, storing in memory or storage, loading from memory or storage) cost more. Just to put things into perspective: for the execution of opcode division (DIV) one has to pay 5 gas, for duplicating a specific value on the stack (DUP) costs 3 gas, while for writing into the persistent storage of an account (storage) (SSTORE) one has to pay 20,000 gas.

Gas can be purchased by paying Ether on a rate called **gasPrice**, which is specified by the sender of the transaction. The sender also specifies an upper bound on the gas, called **gasLimit**, specifying the amount of gas she is willing to spend on running the contract. This means that the sender assigns the transaction along with **gasPrice** $*$ **gasLimit** Ether. If the transaction or contract runs out of gas (OOG) – namely the contract would still run although there is no gas left – an exception is raised, all the gas is subtracted from the sender and all the system changes are reverted. If there is some gas left, then it is sent back to the sender. In every case the consumed gas is sent to the miner, who mined that specific block, which contains the transaction. Sending gas means more precisely that the exact amount of ether which was spent to buy that amount of gas is sent. Obviously miners tend to execute those contracts which offer higher **gasPrice** for a unit of gas, therefore senders are also trying to specify a viable **gasPrice**, which is likely that will be picked up by some miners.

Each block has a value called **blockGasLimit**, which denotes the maximum number of gas the block can consume. The value **blockGasLimit** is dynamically changing according to a predefined function of previous values of **blockGasLimit**. This means that **blockGasLimit** can only be increased or decreased by a certain amount compared to the last block. This prevents a malicious miner setting the **blockGasLimit** extremely high. This feature of Ethereum is crucial since this is an upper bound for computations and serves as another necessary measure to deter DoS attacks in the network.

On the other hand **blockGasLimit** is also a limitation of Ethereum which restricts the processing capacity of transactions of the overall Ethereum network. In case if numerous transactions (say hundreds of thousands) are waiting to be selected and mined in a block, but their publishing on the blockchain is delayed, while continuously full blocks are mined indicating that the network is overwhelmed, such a scenario is called a DoS attack (Denial of Service) [Jam17]. Such DoS attacks can be either malicious or non-malicious.

Non-malicious DoS occurs if simply there are so many pending transactions that they cause unusally long time to process a transaction. Such an event can be caused by the growing popularity of Ethereum or by miners setting the **blockGasLimit** too low.

So far, the most successful malicious DoS attack happened in fall 2016. In the attack a person or group of people were executing opcodes which were cheap to add to the network but for clients it was computationally difficult to process them. During the attack, miners were asked to lower the **blockGasLimit**. This issue was resolved by increasing the gas cost of certain underpriced opcodes. Such changes required a

hardfork of the blockchain and every node in the network to upgrade their Ethereum client software [Jam16].

2.3 Accounts, transactions and contracts

An account is identified by an address, which is a 160-bit code. Accounts have several fields like its balance, a nonce, which serves as a transaction count. They also contain some (possibly empty) EVM Code and a (possibly empty) Storage State associated with them. If the account does not have code associated with it, it implies that the account balance is controlled by some external entity and it is called external account, while when it does have some EVM code associated with it, it is called an Autonomous Object or contract account. An Autonomous Object might be only referred as contract later on, however contract might also mean a piece of EVM code.

A transaction is a cryptographically-signed instruction constructed by an external actor (network participant). Transactions can either be message calls or contract creations (a transaction which results in creating an account with associated code). Both of these two types have the following common fields: **nonce** (the number of transactions sent by the sender), **gasPrice**, **gasLimit**, **to** (address of the recipient), **value** (amount of Ether to be transferred to the recipient) and other values which facilitate to authenticate the sender via elliptic-curve signatures. Additionally, a contract creation transaction contains **init**: an unlimited size byte array specifying the EVM-code for the account initialisation procedure. In contrast, a message call transaction contains **data**: an unlimited size byte array specifying the input data to the message call.

2.3.1 Interacting contracts

The EVM enables accounts and contracts to create other accounts with associated code via the CREATE opcode. It takes 3 operands from the stack determining the initial balance of the account (ie. the value of Ether sent along with the create transaction) and the place of the associated code in memory and its size. If the create transaction had been successful the address of the newly created account is pushed to the top of the stack, otherwise a 0 is pushed indicating that the transaction failed either because it ran out of gas, call stack limit was reached, machine stack limit was exceeded during execution or the balance of the sender was too low on Ether to carry out the value transfer.

EVM also makes it possible for contracts to call each other in different ways by providing corresponding opcodes. These opcodes are extremely useful if one would like to use a functionality of another contract.

- **CALL**: the CALL opcode takes seven arguments from the stack and pushes back after execution a single value of zero (if the call failed) or one (if it was successful). The seven arguments specify the amount of Ether that is sent along with the call, the amount of gas thath the call provides, the address of the callee and memory locations for the input and output data of the call.

- **CALLCODE**: takes the same number of operands from the stack, however in this case the callee address is the same as the one of the caller, while the code is overwritten. This means that if contract A makes a CALLCODE to contract B, a piece of code of B will run in the context of the caller A. This also means

that whenever the code accesses or modifies the storage, it will change that of contract A and not of contract B.

- **DELEGATECALL**: is similar to CALLCODE, however a remarkable difference is that in this case the context is almost entirely identical. If contract A invokes contract B who does DELEGATECALL to contract C, the message sender in the DELEGATECALL is contract A (whereas if CALLCODE was used the message sender would be B).

2.4 Loops and conditionals

As quasi-Turing completeness suggests, EVM should be able to handle and provide loops and conditionals. To that end there are the opcodes JUMP, JUMPI and JUMPDEST. The opcode JUMP simply takes the first element from the stack and sets the program counter to that element. Similarly JUMPI alters the program counter for the first element of the stack if the second element is not 0, otherwise program counter is incremented by 1. JUMPDEST opcodes mark the valid destinations for jumps. They do not have effect on machine state during execution apart from incrementing the program counter and that they cost a bit of gas. If a jump lands on an invalid jump destination, state is reverted and no gas is refunded.

It is not necessary that every time a PUSH opcode would precede the JUMP(I) opcodes. This would essentially imply that all the jump destinations are known statically. However, this is not the case. There are indeed JUMP(I) opcodes, where the jump destination is taken from the stack as a result of some computation. This highly dynamic nature of the EVM (ie. the lack of static jumps) makes harder to determine jump destinations statically.

2.5 Blocks

Ethereum has a blockchain, which contains blocks of data: transactions. The blocks are mined by some participants and distributed to other participants who validate them. The Ethereum blocks contain three types of relevant information:

1. the block header, H,
2. information corresponding to the comprised transactions, T and
3. a set of other block headers U that are known to have a parent equal to the present block's parent's parent (known as ommers).

Now, let us shortly introduce those block header fields, which are relevant to our argumentation:

- **parentHash**: the hash of the parent block's header.
- **beneficiary**: the 160-bit address to which all fees collected from the successful mining of this block shall be transferred
- **difficulty**: an integer corresponding to the difficulty of this block. This can be calculated from the previous block's difficulty and the timestamp.
- **number**: an integer denoting the number of ancestor block i.e. the length of the blockchain at the current block. The genesis block has a number of zero.

- **gasLimit**: an integer denoting the current limit of gas expenditure per block.
- **gasUsed**: an integer denoting the total gas used in transactions in this block.
- **timestamp**: an integer in Unix's time() format denoting the time of this block's inception.
- **mixHash**: a hash which together with **nonce** proves that a sufficient amount of computation has been carried on this block.
- **nonce**: a hash which together with **mixHash** proves that a sufficient amount of computation has been carried on this block.

Some of these block header fields are also accessible for contracts from the EVM via certain opcodes. Such opcodes include BLOCKHASH, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT opcodes. These opcodes are quite useful when contracts have to access blockchain information. Even more interesting and hot topic in the Ethereum community is the application of oracles. Oracles enable smart contracts to access data from the outside world. This is extremely useful, since the EVM is only capable to provide on-chain data to contracts, while oracles are able to send information about stock prices, flight delays or weather conditions. Remarkable oracles are developed by Fan Zhang et al. [Fan16] and Thomas Bertani et al. [Ora17].

2.6 Events and logs

Each transaction has an attached receipt which contains zero or more log entries. Log entries represent the result of events having fired from a smart contract. Events occur during execution and they are intentionally inserted by the contract developer.

Events are dispatched signals the smart contracts can fire. DApps, or anything connected to Ethereum JSON-RPC API, can listen to these events and act accordingly. Event can be indexed, so that the event history is searchable later.

Events/logs are the result of LOG opcodes being executed in the EVM.

Logs are not part of the blockchain itself per se, since they are not required for consensus (they are just historical data), however they are verified by the blockchain as the transaction receipt hashes are stored inside the blocks. The hash of the event logs are recorded in the block header. If one has the header, she can simply gather all logs from that block, hash them and compare to the value stored in the header. If they match, no manipulation took place [Cona].

For more detailed description of the Ethereum protocol, the astute reader is referred to [Woo14].

As a summary, Ethereum could be described as a distributed, decentralized world computer, which is open to anyone. Furthermore anyone can use this computer anytime and all the executions of these distributed programs are verifiable by all the participants.

3 Solidity, a language for smart contracts

Solidity is the most popular programming language used to write smart contracts for the Ethereum blockchain. It is developed by Gavin Wood, Christian Reitweissner, Alex Beregszaszi, Yoichi Hirai and other Ethereum core contributors [Riz]. It has a Javascript-like syntax and it is designed to target the EVM. It is not only used in Ethereum, but other platforms also offer it as a language for developing decentralized applications in such as Countarparty [Der] and Tendermint [Kwo]. Although there are two other high-level languages one can write smart contracts in, the usage of Solidity is almost exclusive among Ethereum peers. Therefore, in the continuation we will focus on presenting the most important and relevant features of the Solidity language.

3.1 The structure of contracts in Solidity

Contracts in Solidity are similar to classes in object-oriented languages. To illustrate this, let us consider the following simple contract as a guiding example.The contract named `MyToken` implements a simple token with basic functionalities: registering token holders' balance and the possibility for sending tokens around.

```
1  contract MyToken {
2      mapping (address => uint256) public balanceOf;
3
4      function MyToken(uint256 initialSupply){
5          balanceOf[msg.sender] = initialSupply;
6      }
7
8      function transfer(address _to, uint256 _value) {
9          if (balanceOf[msg.sender] < _value) throw;
10         if (balanceOf[_to] + _value < balanceOf[_to])
               throw;
11         balanceOf[msg.sender] -= _value;
12         balanceOf[_to] += _value;
13     }
14 }
```

Figure 1: A minimum viable token contract

In general, contracts might have state variables (fields in object-oriented languages) stored in the persistent data storage of the contract: `balanceOf` in the example at Figure 1 it is an array which keeps track of token holders' balances. Contracts might also have functions (methods in object-oriented languages), which can operate on state variables and might possibly change the global state (`transfer` and `MyToken` function). One of the most important function of contracts is the constructor function.

3.1.1 Constructor function

Contracts can be created from outside or from Solidity contracts. When a contract is created, its constructor (a function with the same name as the contract) is executed

once. A constructor is optional. Only one constructor is allowed, therefore overloading is not possible [Foue].

If a contract wants to create another contract, the source code (and the binary) of the created contract has to be known to the creator. Therefore this implies that cyclic creation dependencies are impossible.

The basic syntax of the language (conditionals, loops etc.) are almost identical to the syntax of Javascript. This is not by chance; this was a design choice of the Ethereum core developers to make it as easy as possible to start developing using the Solidity language. Therefore, in the rest of the section we will highlight the main differences in syntax and functionalities between Solidity and Javascript.

3.2 Fallback function

A crucial subtlety of Solidity is a feature called fallback function: a function which does not take arguments, neither return type, nor name. Every time a contract receives a call with none of the functions match the given function identifier (or if no data is supplied at all) or receives money, the fallback function is executed. In case of the fallback function being called and the contract not implementing it then the call throws an exception sending back Ether, making the contract unable to receive money. Therefore every contract has to implement the fallback function if it wants to receive money. An essential protocol level rule for the fallback function is that it cannot use more than 2,300 gas, if it was invoked by the `.send()` function. The `.send()` function has the general form of `x.send(y)`, which means that the contract sends `y` wei ether to the address `x`. The `x.send(y)` function invokes the fallback function of `x`. If the fallback function of `x` uses more than 2,300 gas, an out of gas exception is raised and the call is reverted, furthermore left without any effect [Rei16]. Therefore one should only put minimal computation into the fallback function, if one wants the contract to be able to receive money.

If a contract receives an incoming function call, it extracts the called function's hashed signature via the CALLDATALOAD opcode. If the hashed function signature does not match any of the contract's function signatures then the program counter is set to that of the fallback function and then the code of the fallback function is executed.

3.3 Function calls, visibility and getters

As it was discussed earlier, functions are essential parts of contracts to make them modular. Functions can interact with each other via function calls. In Solidity there are two main types of function calls:

1. Internal function calls: A function can be called within the contract itself. It maintains the context and the caller's gas and calldata (`msg.sender` (sender of the call), `msg.gas` and calldata). It is called a function call, however, it is compiled down to the JUMP bytecode [Hes], therefore it is much cheaper than external function calls.

2. External function calls: this type of function call occurs when a transaction from an externally-owned account is sent to the function with calldata or via the CALL opcode. External function calls are more expensive in terms of gas

consumption. These kinds of call potentialy change the sender, call data and value.

These two types of function calls give four possibilities for a state variable or function in terms of visibility: external, public, internal and private.

The four visibility levels are the following:

- **external**: these functions can be called from other contracts and via transactions. It can not be called internally simply writing `f()`, although `this.f()` will work.
- **public**: these functions can either be called internally or via messages. For public state variables, an automatic getter function is generated.
- **internal**: these functions and state variables can only be called, accessed internally from the current contract or from contracts deriving from the current one without using the keyword `this`.
- **private**: private functions and state variables are only visible in the contract they are defined in, they are not even visible or accessible from derived contracts.

For functions public is the default, while for state variables external is not possible and internal is the default visibility level.

A frequent semantic misunderstanding of smart contract developers is to think that making a function or state variable private prevents the whole world outside of the blockchain observing them. It only prevents other contracts to access or modify them, however on the blockchain these pieces of information are still visible and observable.

3.3.1 External calls in detail

Contracts are not isolated: various opcodes are provided for this purpose as they were presented in Section 2.3.1. They can call and interact with each other via multiple mechanisms: on Solidity level there are several ways to call a function of another contract: `.call()`, `.send()`, `.transfer()` and `.delegatecall()`. They are also called low-level functions since they only return a boolean value on the Solidity level to indicate whether the call was successful or not. These are the most important external calls which work as follows:

- **call**: is used to call a function of another contract or within the contract and sending Ether along with the call. The general syntax of calling on Solidity level has the form of (where `.gas()` and `.value()` are optional)
  `r.call.gas(gasV).value(ethV)(bytes4(sha3("func(uint256)")),param)` where `r` is the address of the recipient of the call, with `gasV` one can specifiy the amount of gas the call is allowed to consume, `ethV` denotes the amount of Ether denominated in wei (a denomination of ether, see Section 3.6) to be transferred to recipient `r`. The called function is identified by the first 4 byte of its hashed signature, if in the `recipient` contract there is no function matching the given signature, the fallback function is executed; finally `param` is the parameter passed to the function call.

- **send**: is used to transfer Ether from the running contract to a recipient `r` in the general syntax of `r.send(ethV)`, where `ethV` denotes the amount of Ether denominated in wei to be transferred to recipient `r`. The send function is also compiled to the CALL opcode (with an empty function signature), however in case of `send()` one can not allocate the amount of Ether oneself; that is 2300 gas is sent along with the send call and this value can not be altered. After the Ether has been transferred, the fallback function of the recipient is executed. In case if the fallback function consumes more gas than 2300, an OOG exception is raised and the Ether transfer is reverted, however consumed gas is not refunded.

- **transfer**: is essentially the same as **send**, however when sending Ether with the transfer function the entire available gas is provided as stipend to the call, it is not set to 2300. This implies that potentially computationally more expensive fallback functions might also not throw an exception when invoked by `transfer()`, although one still has to be really careful what operations one puts in the fallback function.

- **delegatecall**: is quite similar to **call**, with the difference that the invocation of the called function is run in the caller environment [Nic16], i.e. code is delegated into the environment of the caller.

3.4 Types

Solidity is statically typed, supports inheritance, libraries and complex user-defined types. Statically typed means that variable types have to be specified or known at compile-time. Solidity provides a few elementary types which can be combined to form more complex, user defined types. Variables with the right types can also interact with each other in expressions containing operators. A quick overview of this machinery is given hereby.

3.4.1 Value types

Value types is an umbrella term for all those types which are passed by value. In Solidity such types include booleans, (unsigned) integers, fixed-size byte arrays, dynamically-sized byte arrays and various literals (rational, integer, hexadecimal, string) which are well known from other languages. Enumerations are also provided for creating user-defined types in Solidity.

An important platform specific type is the **address** type. It holds a 160 bits value, which represents the address of an account. A useful state variable of addresses is the balance state variable, which holds the amount of Ether denominated in wei held by the account. Sending Ether to accounts is possible by invoking the `.transfer()` function on addresses. The `.transfer()` function also executes the recipient's fallback function just like `.send()`. All contracts inherit the members of address, so it is possible to query the balance of the current contract using `this.balance`.

3.4.2 Reference types

Types, called complex types which do not necessarily fit into a single word (256-bits) can cause headache to developers whether to store them in memory or storage. In every context there is a default, such as for function parameters the memory, or for

state variables the storage, however this can be overwritten by keywords `storage` or `memory`. These are crucial design choices since copying such variables from storage or memory burns a lot of gas.

Essential reference types are (dynamic) arrays and mappings. For user-defined types there are structs that work just like in Javascript or C++.

For an exhaustive list of types and language specification the reader is referred to [Foue].

3.5 Exceptions

In Solidity there are some cases where exceptions are thrown automatically and they can also be thrown manually by using the `throw` keyword. The way Solidity handles exceptions depends how contracts call each other. In general, if there is a chain of nested calls, exceptions are handled in two different ways:

- if all the calls in the chain are internal calls, then the execution stops and all the side effects are reverted. All the gas allocated by the originating transaction is consumed.
- If there is at least one external call in the chain, then the exception is propagated along the chain, reverting all the side effects in the called contracts, until it reaches a call (callcode, delegatecall etc.). From that point the execution is resumed and a false boolean value is returned by the external function call. Additionally all the gas allocated by the exception throwing external function call is consumed [Nic16].

Catching exceptions in Solidity is not possible yet. On the bytecode level exceptions are implemented as jumping to invalid jump destinations.

Solidity automatically generates a runtime exception in the following situations:

1. accessing an array at a too large or at a negative-index,
2. performing an external function call targeting a contract that contains no code,
3. if a message call does not finish properly (runs out-of-gas, no matching function, throws an exception itself), except when a low-level operation (send, call, callcode, delegatecall) is used,
4. divide or modulo by zero etc.

For the full list of automatically generated exceptions the reader is referred to [Foue].

3.6 Units and Globally Available Variables

Useful units are the measures of time and Ether; Subdenominations of Ether such as wei, szabo, finney and ether are supported in Solidity. The ratio of these are the following: 1 ether is worth 1000 finney, 1 finney is 1000 szabo and 1 szabo is 10^{12} wei. Units of time are also built-in and supported by the language such as seconds, minutes, hours, days, weeks and years.

The more interesting parts are the special variables and functions of Solidity. Some of them are straightforward to decode such as `msg.gas` (remaining gas), `block.difficulty`

or `msg.value` (number of wei sent with the message). The function `block.blockhash(uint blockNumber)` returns the hash of a block, where `blockNumber` denotes the blockNumber-th recent block. For scalability reasons `blockNumber<256`, older blocks are not accessible via this function. Solidity has rich support for cryptographic functions like SHA-3, Keccak-256 or elliptic curve cryptography related functions. Address related attributes and functions are also accessible through variables like `<address>.balance` or `<address>.callcode()`.

3.7 Development platforms and availability

It is made easy to start developing in Solidity since it is well supported and maintained on various platforms. There are docker builds for the compiler and in addition, one can just try Solidity in the browser (using the Remix IDE [Foud]), most probably this is the easiest to start with, since it does not require any installation to try out. Building from source is also possible on all major operating systems.

4 Attacks on contracts

Contracts are popular targets of attacks, because they are storing and manipulating real assets, eventually real money. Unlike for web apps or mobile applications, if one manages to hack a contract, the attacker is most likely able to directly steal money. Therefore, there is a direct attractive compensation. High profile contracts which hold hundreds of Ethers are real honeypots for crooks. As of 19th May, 2017 there are more than 378,000 contracts on the blockchain, which all together hold more than 12,000,000 Ether which is worth around US$1.4 billion [Foua].

One also has to keep in mind the two fundamentally different attack scenarios which might occur in the wild. The source of this distinction lies in the different nature of transactions i.e. there are two transaction types: message calls and contract creations. In message calls one would like to ensure that the contract one is interacting with can not fool one in any possible way. One could think about a banking contract and a customer who can perform various tasks via message calls, for example asking the balance, sending money etc. On the other hand also the developer and creator of the banking contract wants to have guarantees that after she deploys the contract on the blockchain certain properties will always hold. For instance, customers of the banking contract should not be able to see other balances than their own or should not be able to steal money from other accounts under any circumstances. These two different positions carry different opportunities for attacks and defenses alike.

4.1 Transaction-ordering dependence

As it was introduced in section 2.5 blocks in Ethereum contain transactions which are executed in a sequential manner, therefore the global state is updated several times in each epoch. A regular node does not have any influence, but solely the miners, on the order in which transactions are included in blocks. Therefore miners can choose an ordering which is beneficial to them, causing harm to the users of the contract. The following contract at Figure 2 is an example for a contract being transaction-ordering dependent (TOD or Tx dependent).

```
1  contract Puzzle {
2          address public owner ;
3          bool public locked ;
4          uint public reward ;
5          bytes32 public diff ;
6          bytes public solution ;
7
8          function Puzzle(){
9                  owner=msg.sender;
10                 reward=msg.value;
11                 locked=false;
12                 diff=bytes32(11111);
13         }
14
15         function (){
16                 if(msg.sender==owner){
17                         if(locked)
```

```
18                                  throw ;
19                          owner.send(reward);
20                          reward=msg.value;
21                  }
22                  else
23                          if(msg.data.length>0){
24                                  if(locked) throw ;
25                                  if(sha256(msg.data)<diff){
26                                          msg.sender.send(
                                                reward);
27                                          solution=msg.data
                                                ;
28                                          locked=true ;
29                                  }
30                          }
31          }
32  }
```

Figure 2: Puzzle: a transaction-ordering dependent contract

In this Puzzle contract shown in Figure 2 the goal of the game is to find a value x, such that $H(x) < target$, where H is the cryptographically secure sha256 hash function and $target$ is a pre-defined value of difficulty. If users are able to submit such a value to the Puzzle contract they can unlock the prize and win some Ether. However a smart (rather malicious) miner who observes transactions going around and sees that a valid solution is found can immediately issue a transaction on her own with the correct value of x and include her own transaction first in the next block and only later the transaction which in reality was created way earlier than hers. This way, the miner is able to get the prize and lock the prize from the honest player. In this case, a possible remedy could be to use commitment-schemes rather than sending solutions in plain-text in the transactions.

The TOD bug recently contributed to the second biggest heist in the Ethereum community causing the loss of US$30 million dollars [Pal17]. A common effort in the smart contract community is to establish programming patterns and standard contracts which can serve as libraries. One could use these functionalities already deployed on the blockchain. This saves gas and development efforts for contract creators and they can additionally be confident that they are using high standard rigorously tested code.

However, if the calling of such contracts happens in an irresponsible way, it can lead to vulnerabilities (in our case the TOD bug). The bug appeared in a multisignature wallet. A wallet is multisignature if it requires more than one private key to spend funds from that wallet. In the most general scenario any k number of private keys out of n private keys are needed to send funds out of the wallet. This gives more security and granularity for users to control their wallet, since it is less likely that numerous private keys are compromised at the same time than only one private key.

At Figure 3 one can observe the simplified version of the used library contract, which implemented all the functionalities required for the multisignature wallet, highlighting the `initWallet` function. `initWallet` function set the limit for daily withdrawals and most importantly the owners of the multisignature wallet. Due to the

nature of the `WalletLibrary` contract, only the owners of the multisig wallet were able to send out funds from the wallet. This function should have been only callable by the constructor function of the `Wallet` contract at Figure 4, however due to a bug this was not guaranteed.

```
contract WalletLibrary{
// constructor - just pass on the owner array to the
   multiowned and
// the limit to daylimit
        function initWallet(address[] _owners, uint
           _required, uint _daylimit) {
                initDaylimit(_daylimit);
                initMultiowned(_owners, _required);
        }
//more functions from the library
}
```

Figure 3: WalletLibrary: a library contract implementing multisig functions

Anytime the `Wallet` contract was called with the `initWallet` function identifier, the fallback function of the `Wallet` contract was executed since there was no match for the `initWallet` function identifier. Unfortunately, all the `msg.data` of the call was forwarded and the delegated call of `WalletLibrary` executed the `initWallet` function in the context of the `Wallet` contract since DELEGATE call was used. This way one could reinitialize the `Wallet` contract, assigning new owners to the `Wallet` contract at will. The attackers could then drain the contract since they became the "legitimate" owners of the multisig wallet [Lor17].

The vulnerability of letting the `Wallet` contract reinitialize itself via the fallback function made the contract transaction-ordering dependent. A slicker miner can influence who will be the owner of the multisig wallet by the ordering of the transactions in a block. The sender of the transaction executed later will be the new owner of the `Wallet` contract.

This vulnerability could have been escaped if double initialization had been checked by the `initWallet` function in the `WalletLibrary` contract. Another simple way to deter this attack would have been to mark the `initWallet` function as `internal`.

```
contract Wallet{
        function() payable {
                // just being sent some cash?
                if (msg.value > 0)
                        Deposit(msg.sender, msg.value);
                else if (msg.data.length > 0)
                        _walletLibrary.delegatecall(msg.
                           data);
        }
        //more Wallet functions
}
```

Figure 4: Wallet: a transaction ordering dependent contract

According to [Sir17] various multisig wallets are affected by the above mentioned bug. Luckily a group of white hat hackers drained the vulnerable contracts in time (ie. before the bad guys) saving the community from a US$200 million dollars loss of Ether. Afterwards they refunded the affected nodes to other safe wallets and addresses [Jba].

4.2 Timestamp dependency

Since everything is posted on the blockchain which is written in the code – even local variables and state variables are marked private – it is a tricky and hard task to generate random numbers. Additionally, the blockchain is not a good source of randomness, especially if one uses a value which is set by the miners. A common error made by developers is to use the TIMESTAMP opcode (which pushes the block timestamp to the stack) in a random number generator function as a seed or to depend in any way on timestamp values. The timestamp of a block marks the time of the block's inception, however this value is set by miners within a reasonable range: it has to be larger than the previous block's timestamp, and should be within 900 seconds from the timestamp on the miners' local system [Luu16]. Just imagine a lottery contract where the selection of the winner depends on a random number function, which uses the block timestamp as a seed. By precomputing seed values, a miner can choose a timestamp value which is beneficial for her, making herself the winner of the game.

```
function(){
        if(timestamp>0 && now-timestamp>24 hours){
                msg.sender.send(msg.value);
                if(this.balance>0){
                        leader.send(this.balance);
                }
        }
        //...//
}
```

Figure 5: Lottopolo: a timestamp dependent contract

In Figure 5 a general example for timestamp dependency can be found. The flow of Ether directly depends on timestamp values as the block timestamp occurs in the conditional of the if-then construct [exa]. Thereby it is clear that the contract presented in Figure 5 is timestamp dependent.

Another important example of the timestamp dependency bug can be observed in Figure 6. This is a code snippet from a contract called `theRun` from the Ethereum blockchain. For sake of clarity, only the relevant function is shown here, while the rest of the code can be seen at [con]. The flow of Ether i.e. the selection of the winner in this lottery contract is chosen by the aid of the home brewed `random(uint Max)` function. Furthermore the seed of the random function is the block timestamp value making the random function timestamp dependent. In contrast with the Lottopolo contract, theRun contract is indirectly timestamp dependent through the usage of timestamp as a seed.

```
uint256 constant private salt =  block.timestamp;

```

```
3     function random(uint Max) constant private returns (
          uint256 result){
4         //get the best seed for randomness
5         uint256 x = salt * 100 / Max;
6         uint256 y = salt * block.number / (salt % 5) ;
7         uint256 seed = block.number/3 + (salt % 300) +
              Last_Payout +y;
8         uint256 h = uint256(block.blockhash(seed));
9
10        return uint256((h / x)) % Max + 1; //random number
              between 1 and Max
11    }
12
13 }
```

Figure 6: theRun: another timestamp dependent contract

To avoid this problem one could potentially use oracles to get pseudo-randomness [Ora17]. Another viable solution could be to use block hashes to generate pseudo-random numbers [Har16] or by querying other verified contract, which serves as a pseudo-random function. An example implementation of the later one relies on the following scheme: the general idea is to create a decentralized trustless psuedo-random number generator, which is implemented in contract, let's say, **C**. Each participant sends a number s_i to contract **C** using a commitment scheme implying that actually $sha3(s_i)$ is sent to the contract. After the revealing phase, only the revealed s_i values are used in function $f(s_1, \ldots, s_n)$, where f is a pseudo-random number generator function. The first and widely used implementation of this idea is the `RANDAO` contract [Zha].

4.3 Mishandled exceptions

One of the major weaknesses of Solidity from a developer's point of view is the poor design of exception handling. In Ethereum, uncaught exceptions might cause serious losses and since there is not given a proper exception handling mechanism from the language side, one has to be truly careful designing contracts when it comes to unexpected program execution or errors.

This problem is extremely prevalent in case of contracts calling each other via some low-level functions (`send`, `callcode`, `delegatecall`) as it was presented in Section 3.5. These calls can fail due to several reasons: out-of-gas or the call stack limit is reached. If an exception is raised in the callee, all changes caused while executing the call are reverted and the callee returns a false boolean value to the caller. Unless the caller does not check explicitly on the Solidity-level the returned boolean value from the call, exceptions and unexpected behaviours remain unnoticed.

```
1 contract KingOfTheEtherThrone{
2         address monarch;
3
4         function claim(){
5                 // calculate the ruler's compensation
6                 monarch.send(compensation);
7                 monarch=msg.sender;
```

```
8           }
9           //...//
10  }
```

Figure 7: KingOfTheEtherThrone: a mishandled exception example

An aching example for mishandling exceptions is the King of The Ether Throne (KoET) contract [exc]. KoET implements a simple ponzi scheme: the operator of the game generates returns for older investors through revenue paid by new investors. In KoET usurpers can claim the Ether Throne by paying the ether-price for it. The leaving king gets this aforementioned Ether-price, which should be bigger than the price which was paid by the former king. The difference is the profit of the leaving king. In the simplified code snippet in Figure 7, the compensation to the former king is sent by the `.send()` function. The `.send()` function is compiled down to the CALL opcode, which can fail due to many reasons. A common mistake is that the fallback function consumes more than 2,300 gas, therefore making the callee contract incapable of receiving Ether. If such a situation occurs, the send function returns `false`, however this scenario is not handled by any mean in the KoET contract. Even if the send function returns false there is no measure in the contract to get this situation over. It simply ignores it by continuing its execution, implying that in this case the compensation of the former king is stuck in the KoET contract forever. Due to this bug, KoET had to shut down in February, 2016 [Etha].

The very same bug in KoET can also be exploited by deliberately exceeding the call stack limit. To be more precise, such attacks – also referred to as call depth attacks in the literature [Zik16] – only affect contracts before block no. 2,463,000, since after this block protocol changes [But] made practically impossible call stack attacks by increasing gas costs of I/O heavy operations such as SLOAD, CALL, DELEGATECALL etc. No matter how the vulnerability is exploited, the takeaway message is, that it might be harmful to keep going believing we have sent out funds when we have not.

Best practice for those who want to receive Ether is to be very concerned what instructions are put in the fallback function. Under any circumstances the fallback function should not consume more than 2,300 gas. There is already a tool, which can check gas consumption of functions [Fouc]. Those, who actually want to send out Ether from their contracts should avoid using directly the send function. They should rather isolate the effects of the send only for one party at a time by issuing a bail out, meaning that they should create a pattern where the recipient herself withdraws the money [Zik16]. A more ambitious solution would be to upgrade the EVM such that it could offer proper exception handling mechanisms also from the language side.

4.4 Reentrancy bugs

A program (contract) or a function is reentrant if it can be interrupted in the middle of its execution and can be called from another program or method. However this general definition of reentrancy in the context of Ethereum applies in a slightly different way as the execution of contracts can not be influenced externally, therefore interrupting a method is not possible. The only way the execution of a contract is interrupted if it calls another one, this scenario can be dangerous if the caller method is reentrant, which means that the callee could potentially reenter the original method again. This bug could be extremely dangerous since if a function is reentrant and could be

reentered several times while the function sends out money, this money is sent out multiple times although the internal logic of the contract assumes that the function is executed only once. Potentially,an attacker can drain all the money from a reentrant contract by exploiting this vulnerability [Ves].

Let us imagine a general, simple, rudimentary banking contract, called the EtherBank [Ves]. It has three basic functionalities: getting the balance, adding to the balance and withdrawing money implemented by `getBalance`, `addToBalance` and `withdrawBalance` functions, respectively. The problematic function is `withdrawBalance`. Every time `withdrawBalance` is called from a customer contract, the `.call()` function is executed which by design executes the customer's fallback function (see Section 3.2). This is a potential attack surface, since we can not assume anything about the code in the customer's fallback function.

```
contract EtherBank{
        function getBalance(address user) constant returns
            (uint) {
                return userBalances[user];
        }

        function addToBalance() {
                userBalances[msg.sender] += msg.amount;
        }

        function withdrawBalance() {
                amountToWithdraw = userBalances[msg.sender
                    ];
                if (!(msg.sender.call.value(
                    amountToWithdraw)())) { throw; }
                userBalances[msg.sender] = 0;
        }
}
```

Figure 8: A banking contract vulnerable to reentrancy bug

Moreover, the `withdrawBalance` is a reentrant function since it fails to modify customer balances before the `.call()` function is executed. This means just before `.call()` is executed the contract's internal state is exactly the same as right after `.call()` is executed and the fallback function is called. This is a crucial mistake, since a malicious fallback function could potentially reenter the `withdrawBalance` repeatedly, until **gasLimit** is reached or the balance of the entire EtherBank is drained to the attacker's contract.

The attacker's fallback function could be the same in essence as the one shown in Figure 9.

```
function () {
  vulnerableContract v;
  uint times;
  if (times == 0 && attackModeIsOn) {
    times = 1;
    v.withdraw();
   } else { times = 0; }
}
```

Figure 9: A malicious fallback function

This specific fallback function would cause to withdraw the attacker's balance twice. The attacker has to be careful not to run out of gas, otherwise the effect of the whole attack is reverted. One possible way to fix this bug is to use mutexes or simply subtracting the to-be-withdrawn amount from the balance before the actual money sending is executed.

An influential exploitation of this bug has taken place on June 17th, 2016. An attacker [Unk] was able to steal 3,641,694.241898506 Ether (worth US$59,578,117.80 at the time of the attack) from The DAO 1.0 contract [Slo]. It is still, as of today, one of the biggest heists of all time.

It was one of the first cautionary signs for smart contract developers to test mission critical high-profile contracts with more scrutiny. The heist drew more attention to smart contract security, more specifically to formal methods and formal verification [But16c].

5 Foundations of static analysis

The failure of numerous high profile smart contracts presented in Section 4 made it necessary for the Ethereum community to start preventing and fixing bugs occurring in smart contracts. Moreover, these failures were perfect calls for methods from the realm of program analysis and formal verification. Such works are continually emerging from academia and industry side as well, targeting for making smart contracts safer, see Section 8. In this section, the foundations of static analysis are presented.

5.1 Static vs Dynamic analysis in the blockchain environment

In safety-critical computer systems, like the Ethereum blockchain, detecting and locating vulnerable code is essential. To find these locations in an automated, rigorous and scalable manner, program analysis is required. Computer programs can be analyzed statically and dynamically. These two branches of program analysis are called static and dynamic program (or code) analysis, or just short static and dynamic analysis [Dav08].

The term static analysis covers analysis techniques that perform analysis either on the source code level or object code level without actually executing these analyzed programs. A great advantage of static analysis is that they are potentially able to cover all the execution paths of a program and additionally prove formally the absence or presence of a bug.

In contrast to this notion, dynamic analysis tools perform analysis by running the application and observing its behaviour for certain inputs or attack vectors. Typically, dynamic analysis is not capable to cover all the execution paths, therefore they can not give guarantees about the absence of a bug. However they allow analyzing programs where the source or the object code is not known or available, but the most important advantage of dynamic analysis is that they report only real bugs [DuP13].

These two approaches as one can see are fundamentally different; they perfectly complement each other, hence in most cases they are used in tandem. Often static analysis is used in early development cycles, and before shipping the code one might use dynamic analysis to validate the findings of static analysis [Jon13].

5.2 Soundness and completeness

Soundness and completeness are two key properties of static analyzers.

The term **soundness** is originated in mathematical logic: a deductive system is sound with respect to a semantic if it *only* proves valid arguments [Yic12]. This notion naturally extends to static analysis techniques, where soundness means the preservation of program semantics, which is the principal requirement of a correct, useful static analysis. In the context of bug detection soundness implies the ability to catch all possible occurences of a bug. This means that sound static analyzers do not produce any false negative errors. A sound static analyzer achieves this property by overapproximating the behaviours of the program. Therefore a static analyzer is guaranteed to identify all violations of a property, but also might report some false positives, or violations of the property that can not actually occur.

Similarly **completeness** in mathematical logic means that a deductive system is complete with respect to a semantic if it proves *every* valid argument. Completeness of a static analyzer implies that it does not produce false positives, although there might

potentially be uncaught false negatives. Such static analyzer obtains correctness by underapproximating the behaviors of the program. All violations of a property reported by the analyzer corresponds to an actual violation of the property, although there is no certainty that all violations will be reported.

5.3 Sensitivities

Unfortunately, due to decidability problems, total precision (absence of false positives) cannot be achieved while maintaining soundness [Chr09]. It is a driving force in the design of analysis tools to achieve the highest precision possible even if it carries another trade-off formal verification engineers have to deal with. Precision significantly influences the computational complexity of an analysis, namely its scalability. As a rule of thumb one can say that generally better precision implies worse scalability while fast algorithms are usually imprecise. Hence in the field of program verification a large number of techniques were developed from fast/imprecise methods ranging to slow/more precise ones so that engineers, developers could choose the approach which fits the most to their application scenario and purpose of analysis [Chr09].

In static analysis it is a major task to minimize the number of false positives. It is achieved by deploying relevant features which could possibly affect analysis results. An analysis that captures such relevant features is said to be sensitive. These features (sensitivities) can be flow-, value-, context-sensitivity among others. Empirical results and decades of research history show that the deployment of appropriate sensitivities can dramatically decrease the number of false positives, however on the other hand increases computational complexity of the analyzer [La05].

5.3.1 Flow-sensitivity

Flow-sensitivity is able to substantially decrease the number of false positives in certain settings [CH95]. Flow-sensitivity for a static analysis is the capability to take into account the control flow order in which statements may execute in a sequential program. Conversely, flow-insensitive analysis is not capable of having different approximations at different program points i.e. there is only one global solution for the program. One could also highlight the difference between flow-sensitive and flow-insensitive analysis that the former applies strong updates (updates state over-approximations), while the later applies weak updates (accumulate state over-approximations). A strong update overwrites the old content of an abstract memory location l with a new value, whereas a weak update adds new values to the existing set of values associated with l. Whenever safe, it is preferable to apply strong updates to achieve better precision [Isi10].

5.3.2 Value-sensitivity

Value-sensitivity is the ability of a static analysis to approximate runtime values, e.g., by skipping unreachable program branches [Ste16].

5.3.2.1 Gas-sensitivity

Gas-sensitivity can be considered as a subclass of value-sensitivity. It is a sensitivity specific to Ethereum denoting the ability of a static analysis to approximate runtime gas values and use this information to improve precision. Gas is an intrinsic notion of

the EVM, hence it should also be reflected in the static analysis to filter out out-of-gas related exceptions and throws.

5.3.3 Context-sensitivity

Context-sensitivity is the ability of the analysis to compute different static approximations upon different method calls. Contrarily, context insensitive analysis conflates call sites, allowing one call to return its value to a different call site where it was not even called. This can cause serious loss in analysis precision. Context sensitivity solves this problem by distinguishing call sites.

5.4 Bug finding as a classification problem

One of the main applications of program analysis is to catch specific bugs in the analyzed code [Dil]. Usually, analysis methods are used and implemented in tools which are subsequently applied to analyze programs. Multiple tools could use the same analysis technique and because tools are the ultimate layer of program analysis which yields results, the performance of the tool has to be assessed. From the perspective of tools, program analysis can be considered as a binary classification problem, where the question is whether an analyzed program contains a bug or not. The tools return an answer regarding a program being buggy which might be in line with reality or not. Hereafter, the following general notions will be relevant in our discussion (illustrative comments are tailored to our specific application scenario of smart contracts).

Definition 5.1. Terminology of binary classification: Generally a program, in our case specifically a contract can belong to one of these sets depending on whether it contains the analyzed bug or not and whether it was detected or not. As a remark: hereby notions of a contract being secure or buggy is defined with respect to a certain property.

- **Positive (P)**: the number of positive cases in the data, i.e. the buggy contracts.
- **Negative (N)**: the number of negative cases in the data i.e. the secure contracts.
- **True Positive (TP)**: the number of hit or true positives i.e. those contracts which are indeed buggy and identified as being buggy as well.
- **True Negative (TN)**: the number of correct rejection or true negatives i.e. secure contracts which are identified as not being buggy.
- **False Positive (FP)**: number of false alarm or false positives i.e. a contract which is flagged as buggy although it is secure.
- **False Negative (FN)**: number of miss or false negatives i.e. those contracts which have bugs although are not found by the tool and flagged as secure.

Applying the just defined notions one can introduce other useful measures to assess the performance of static analytic tools.

Definition 5.2. Derivations from the confusion matrix

- **Sensitivity (recall, hit rate or true positive rate (TPR)**: $TPR = \frac{TP}{P}$

- **Specificity (true negative rate (TNR))**: $TNR = \frac{TN}{N}$
- **Precision (positive predictive value (PPV))**: $PPV = \frac{TP}{TP+FP}$
- **Negative predictive value (NPV)**: $NPV = \frac{TN}{TN+FN}$
- **False negative rate (FNR)**: $FNR = \frac{FN}{P}$
- **False positive rate (FPR)**: $FPR = \frac{FP}{N}$

6 Benchmark for smart contract verification

As formal verification of smart contracts is evolving and developing, there is a crucial need to establish a common platform for comparing the performance of various tools and evaluate their properties. Such benchmarks already exist in other fields of formal verification, like Android application verification, such as the open test suite called DroidBench [Arz]. These benchmarks provide a direct measure to evaluate the performance of proposed formal verification tools for Android applications. Indeed, in the Ethereum community such a benchmark is heavily needed, since currently all the quantitative analysis for individual tools were made in different slices of the Ethereum blockchain making these results incomparable ([Luu16], [Zik16]). Even if contracts were marked as vulnerable with some specific bug, the only way to check this was manual investigation which is tedious and error-prone.

Therefore, we propose a benchmark of Ethereum smart contracts, which enables formal verification researchers to test their tools on. We would like to place this database to the hands of the Ethereum community and invite researchers as well as developers to propose and add contracts to this open test suite.

The proposed benchmark seeks to be a tool which is used to assess the sensitivity and specificity of static analytic tools regarding certain types of bugs by proposing smart contracts which can potentially be difficult to detect for certain types of static analytic tools. This helps users in making informed decisions about what tools to use, how to use them, and what results to expect. The goal of this benchmark is to help making smart contract security more visible. Enclosed contracts are written in Solidity solely for presentation purposes, although also the benchmark and the majority of tools are targeting the bytecode level of contracts. For better understanding, the precise structure of the benchmark is depicted in Table 1. Benchmark files are text files containing contracts in bytecode format. From the file names extensions are omitted for sake of clarity. Most of them are compiled from Solidity, except the files in the Serpent row and those from the manually optimized row. For sake of transparency also source code files are enlcosed from which benchmark contracts are compiled. Solidity files have the .sol extension, while Serpent files have .se extension. In case of manually optimized contracts, Solidity contracts are attached from which they are further optimized manually.

Table 1 Benchmark skeleton

	Bugs			
	Timestamp dependency	Mishandled exceptions	Reentrancy	Tx ordering
Flow-sensitive	fSTime	fSMis	fSRe	fSTx
Value-sensitive	vSTime	vSMis	vSRe	vSTx
Gas-sensitive	gSTime	gSMis	gSRe	gSTx
Context-sensitive	cSTime	cSMis	cSRe	cSTx
Simple	time	mis	re	tx
Tricky bytecodes	trickTime	trickMis	trickRe	trickTx
Serpent	timeSe	misSe	reSe	txSe
Manually optimized	manOptTime	manOptMis	manOptRe	manOptTx

The benchmark challenges tools to detect four different kinds of bugs — namely

timestamp dependency, mishandled exceptions, reentrancy bugs and transaction-ordering dependence — in various settings. In the first part of the benchmark, tools are tested for different sensitivities. In the second part of the benchmark, tricky contracts are added to test whether analyzers are able to catch difficult-to-detect false negatives; test coverage ranges from simple buggy contracts to buggy contracts having difficult-to-model bytecodes (only referred to as tricky bytecodes). Contracts containing tricky bytecodes aim at testing support of uncommon opcodes. Still, in this part of the benchmark one can find contracts written in other Ethereum language (Serpent) and manually optimized contracts. Serpent and manually optimized contracts are designed to test whether tools are solely focusing on valid compilations of Solidity code or whether they can handle bytecodes compiled from Serpent and manually crafted bytecodes as well, since on the Ethereum blockchain contracts are not necessarily written and compiled from Solidity.

6.1 False positives and sensitivities

In the first part of the benchmark, tools are tested for whether they have a certain sensitivity or lack them. These part of the test suit can possibly inform us about what kind of false positives one can expect from the static analyzer.

6.1.1 Flow-sensitivity

The lack of flow-sensitivity can potentially lead to false positives by not being able to calculate different approximations at different program points (see Section 5.3.1) as the following examples indicate.

6.1.1.1 Reentrancy bug false positive

In order to identify reentrancy bugs an analyzer should be able to detect state changes within functions. Reentrancy is especially dangerous if it takes place in a function which pays out Ether. If such a function could be executed multiple times externally via a reentrancy bug, one can deplete all the Ether of the vulnerable contract (see Section 4.4).

```
function withdrawBalance() {
    uint amountToWithdraw=userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    if(amountToWithdraw>0){
        if (!(msg.sender.call.value(amountToWithdraw)())){
            throw;
        }
    }
}
```

Figure 10: A flow-insensitive false positive for the reentrancy bug

State changes are hard to follow for a flow-insensitive analyzer what can cause several false positives. Let us take as an example the banking contract, already seen in Section 4.4, in Figure 10. This code snippet is a highlight from a simple banking contract which consists of multiple functions (asking the balance, adding money to balance), however, here only the reentrant function is shown.

It is essential that all the users should only be able to withdraw their balance once. For this reason, it is essential to avoid reentrancy bugs in such a function. Since money is sent out at line 5 by the `.call.value()` function, another contract's fallback function is executed opening up a potential place for the reentrancy bug. To circumvent an exploit, one should make sure that reentering the `withdrawBalance()` function and executing the `.call.value()` command at line 5 is not possible once again. This demand is successfully satisfied by deploying a new variable, called `amountToWithdraw`, which efficiently guards outgoing payments.

To make sure that the reentrancy bug can not happen, contract developers should verify that function `withdrawBalance()` can not be reentered at line 1. For the first time `withdrawBalance()` is called, the `amountToWithdraw` variable equals the balance of the message's sender (line 2). Right after this point as a result of line 3, the same balance is zeroed out. If `amountToWithdraw` is bigger than zero, then the balance is sent out to the caller. Even if there was a malicious fallback function which attempted reentering the `withdrawBalance` function, it would simply be not able to execute the `.call.value()` function once again, since the balance is zeroed out, hence the `amountToWithdraw` is zero this time, implying that one can not reenter again to the `if` conditional. For this reason the balance is only sent out once and only once.

However, a flow-insensitive analyzer would flag this contract as buggy. The reason is that due to weak updates at line 3, the analyzer overapproximates the `amountToWithdraw` variable with values of 0 and the balance of the message sender. Since a flow-insensitive analyzer does not update state approximations, but rather accumulates them (see Section 5.3.1), `amountToWithdraw` stays there as a possible value for the balance of the message sender even for the second entering of the function, suggesting that a reentrancy attack is possible. Even though this is prevented by setting the balance to zero, the analyzer fails to catch this since it is flow-insensitive.

6.1.1.2 Timestamp dependency bug false positive

Relying on random numbers that can be influenced by actors external, such as miners, is a serious security risk. It is extremely critical, if the flow of Ether is influenced by such random numbers (see Section 4.2).

```
1 function draw(){
2         uint rand;
3         rand=TimestampDependentRand();
4         rand=TrueRand();
5         Payout(rand);
6 }
```

Figure 11: A flow-insensitive false positive for the timestamp dependency bug

There is enclosed a sweepstake contract at Figure 11. In this case, the winner is selected by using random numbers. Therefore, implicitly the redistribution of Ether is also relying on random numbers. At first glance, one might think that the payout function is timestamp dependant, since it comes from the flawed `TimestampDependentRand()` function, however, the later random number is used at line 4 coming from a pseudo random function called `TrueRand()`, that is not timestamp dependant (see Section 4.2).

On the other hand, this contract is deemed to have a timestamp dependant Ether flow by a flow-insensitive analyzer. Due to weak updates (value accumulation to variables), the flow-insensitive analyzer considers variable rand having timestamp dependant and true random values. Since the former case can also happen according to the analyzer, the flow of Ether is flagged as timestamp dependant resulting in a false positive error.

6.1.1.3 Mishandled exception bug false positive

The following example in Figure 12 most probably would raise a false alarm for a flow-insensitive analyzer, since at line 5, variable `i` is over-approximated by holding values of `0` and `100`. Therefore, according to a flow-insensitive analyzer, it might be that the contract enters the conditional at line 6 which implies that line 9 can be reached where there is a mishandled exception. However, in fact this is not the case here, since `i` is set to `100` at line 5 causing the contract not to enter the conditional. This implies that line 9 is never reached and that there is no outgoing Ether at all, i.e. the contract is not vulnerable to the mishandled exception bug.

```
1  contract SendBack {
2      mapping (address => uint) userBalances;
3      function withdrawBalance() {
4          uint i = 0;
5          i = i + 100;
6          if (i == 0){
7                      uint amountToWithdraw = userBalances[
                           msg.sender];
8                      userBalances[msg.sender] = 0;
9                  msg.sender.send(amountToWithdraw);
10         }
11         }
12 }
```

Figure 12: A flow-insensitive false positive for the mishandled exception bug

6.1.1.4 TOD bug false positive

To avoid repetition we skip introducing in depth the TOD bug false positive example for flow insensitive analyzers, since it follows the very same logic of the contract already introduced in Section 6.1.1.3: a TOD vulnerable piece of code (in our case the fallback function of TOD contract from Section 4.1 in Figure 2) is placed into a conditional where the guard is assigned and reassigned right before the conditional. Due to the flow-insensitive nature of the analyzer, the analysis would falsely deem that the body of the conditional might potentially be entered and executed, however it is not.

6.1.2 Value-sensitivity

To fool value-insensitive static analyzers one can apply the following tactics: a simple way to go is that one puts vulnerable code into an unreachable branch. One can make a branch uncreachable by testing a variable in the guard which has a specific value.

A value-insensitive analyzer would overapproximate by assuming that the branch is reachable, since it does not handle the precise value placed in the guard of the conditional statement. Therefore, it would raise a false alarm. This pattern is easily applicable for all the covered types of bugs, therefore it is only shown for one of the attacks to avoid redundancy.

6.1.2.1 Mishandled exception bug false positive

A fragment of a contract which causes false positive for a value-insensitive analyzer might look like as the one depicted in Figure 13.

Although the body of the conditional is never reached, a value-insensitive tool considers it as reachable therefore raising a false alarm. One could have just assigned two different values to variable `x`, although applying a loop was a design choice of the benchmark to also test whether tools are capable of handling and unfolding loops properly.

```
function (){
        uint x=0;
        for(int y=0;y<=10;y++){x++;}
        if(x==0){
                msg.sender.send(msg.value);
        }
}
```

Figure 13: A value-insensitive false positive for the mishandled exceptions bug

6.1.2.2 Gas-sensitivity

The fragment shown in Figure 14 is intended to show a simple relay contract, which sends back all the money it receives via the `.send()` command in the fallback function (as a very similar contract to this is implemented and deployed on the Ethereum blockchain, cf. [Conb]). A gas-insensitive analyzer would flag this contract as one with a mishandled exception bug, although this is not the case, since every time the fallback function is executed, it will run out of gas due to the massive gas consumption in the loop body. Therefore all the state changes are rolled back and in this case the buggy `.send()` function will do no harm.

```
contract SendBack {
    function() {
                uint counter=0;
        msg.sender.send(msg.value);
                for(uint i=0;i<1000;i++){
                        counter++;
                }
    }
}
```

Figure 14: A gas-insensitive false positive for the mishandled exception bug

Again to avoid redundancy, we do not show the remaining three false positives for the gas-insensitive case, since they follow the same tactics as presented in Figure 14:

due to massive gas consumption in the fallback function, the effect of vulnerable code is reverted.

6.1.3 Context-sensitivity

Conflating call sites causes serious loss of precision in modelling function calls properly.

6.1.3.1 Timestamp dependency bug false positive

Imprecision introduced by weak updates and state over-approximations make context-insensitive analyzers incapable of catching the following bugs.

```
contract lottopollo {
  address leader;
  uint    timestamp;
  Dummy dummy;
  //payOut is omitted
  function randomGen() constant returns (uint randomNumber
     ) {
       return block.timestamp;
    }
        function draw(uint seed){
                bool isZero;
        isZero = dummy.isZero(0);
        isZero = dummy.isZero(1);
        if (isZero){
            uint randomNumber=randomGen();
            payOut(randomNumber);
        }
  }
}

contract Dummy {
    function isZero(uint num) returns (bool){
        if (num == 0)
            return true;
            else
            return false;
    }
}
```

Figure 15: A context-insensitive false positive for the timestamp dependency bug

By conflating call sites, the analysis will consider an execution path which does not exist in the program. More precisely, it will report that it is possible to return to `isZero` the value of `dummy.isZero(0)` right before the guard of the conditional. If this is possible, then the body of the conditional gets executed and obviously function `Payout` would rely on a timestamp dependent random number, however, it is clear from Figure 15 that function `payOut` is not even executed since the guard always evaluates to `false`, therefore this contract is indeed a false alarm.

The same tactic can be applied to functions having different vulnerabilities.

6.2 False negatives

It is essential to test tools not only for false positives, but also for false negatives; to increase coverage we test tools for whether they are able to support the full EVM bytecode set or whether they are only able to support contract bytecodes compiled from Solidity and therefore can not support other languages such as Serpent.

6.2.1 Mishandled exceptions: a pattern matching technique

An interesting tool is the static analyzer used for scanning the live Ethereum blockchain to detect mishandled exceptions caused by the `.send()` function [Zik16]. Due to its simple pattern matching logic to detect these vulnerabilities, it is quite uncomplicated to fool the analyzer by false negatives.

The `.send()` function, which ignores the return value is compiled down by the Solidity compiler for these opcodes:

"CALL SWAP4 POP POP POP POP POP"

When the CALL opcode completes, it places five values on the stack, and the first of them is the return code. In the bytecode fragment above, one can observe that the five sequential POP opcodes clear these values off the stack, even before they could be used at all, essentially resulting in a mishandled exception. The SWAP4 command here is redundant, since it just swaps the first and the fifth element of the stack, although it does not change the semantics of the program, since all these 5 elements are wiped off.

If the return value is stored/checked immediately after the function call, the corresponding bytecode representation will be the following:

"CALL SWAP4 POP POP POP POP SWAP1 POP"

The additional SWAP1 opcode makes it possible to push the return value even further into the stack: this bytecode pattern leaves the return value available to the subsequent code.

The analyzer matches these two patterns against bytecodes and decides whether a contract has mishandled exceptions or not. This is the simple tactic deployed by the static analyzer [Wen] to detect contracts not having the mishandled exception bug, which clearly leaves space for false negatives as the following snippet proves.

```
1 function payBack(){
2         bool returnValue;
3         returnValue=msg.sender.send(msg.value);
4 }
```

Figure 16: A mishandled exception failed to detect

The fragment of the contract presented in contract snippet 16 is flagged as not buggy by the tool, however it obviously is. Even though the return code of the `.send()` function is stored in the variable `returnValue`, it is not used in any reasonable way to handle exceptions. Therefore it is quite likely that the tool produces a large number of false negatives.

6.2.2 Difficulties of modelling the global state

It is quite cumbersome for a static analysis tool to model the blockchain environment, especially the possibility of reverting the global state in some special cases. When modelling the global state one needs to take care of subtleties of the semantics, namely that the global state might get reverted: it might happen that a message call or function call runs out of gas, exceeds the call stack limit or by other reasons throws an exception. Either way, if an exception is thrown, the global state is reverted, although the consumed gas is not refunded. However, these state changes are kept and not reverted in Oyente [Luu16] (See Section 8). They are unable to discard changes introduced by calls which are failed. This logical flaw in modelling correctly the global state opens up the possibility to construct false negatives.

In the example shown in Figure 17, the variable `secureMode` is set to false by default, which implies that every time function `dangerous()` is invoked, it will use a a `Payout()` function which is based on a timestamp dependent function. Therefore by default this contract is vulnerable to the timestamp dependency bug. The only way to set `secureMode` to `true` is to call the fallback function somehow. Some tools, like [Luu16], consider this and assert that the contract is timestamp dependent bug free, however it is not true. Whenever the fallback function is called and executed it will fail due to the massive gas consumption taken place in the loop. Since the fallback function fails, also variable `secureMode` is set back to `false`, however many tools keep changes in the global state and the local state of the caller, therefore would deem that `secureMode` is successfully set to `true`. This contract is a buggy one, however many tools fail to detect it, therefore it is a valid false negative of the tool.

```
contract oyenteFalseNegative{
        bool secureMode=false;
        uint counter=0;
        function dangerous()){
                if(secureMode==false){
                        rand=timestampDependentRand();
                        Payout(rand);
                }
        }
        function(){
                secureMode=true;
                for(uint i=0;i<1000;i++;){
                        counter++;
                }
        }
}
```

Figure 17: An Oyente false negative contract

6.2.3 Tricky bytecodes

All of the static analysis tools provide support only for a subset of Solidity or for a subset of the EVM opcodes depending whether they are targeting the bytecode-level or the Solidity-level. The missing subset usually is not modelled or considered at all. This defect causes severe loss in precision. Tools either crash or give unreasonable

results when they encounter opcodes they can not handle (see Section 7). For that end it is needed to test which critical opcodes tools are able to handle and analyze correctly.

6.2.3.1 The CALLCODE opcode

Several contracts use libraries to leverage the power of already deployed contracts [Sir17]. These contracts usually call libraries via `.delegate()` or `.callcode()` functions which are compiled to DELEGATECALL and CALLCODE opcodes respectively.

The following contract at Figure 18 is a general banking contract, however one of the functionalities is not implemented by `contract A` itself, rather it uses a callcode to another contract to have the `withdrawBalance` function. To do so one should provide the address of the contract she wants to call, this can be seen at line 6. It is clearly a reentrant contract, since it invokes the reentrant `withdrawBalance` function of `contract reentrant`, however this remains undetected, since tools are incapable of treating callcodes and they simply just crash or give false negative results. Such tools which can not handle callcodes include [Luu16] and [Kar16]. This contract pair could be also held up as an example for false negative mishandled exceptions as the return value of the low-level `callcode` is not used.

```
1  pragma solidity ^0.4.0;
2  contract A{
3      mapping (address => uint) userBalances;
4
5      function withdrawBalance_A(){
6          var c=reentrant(0
               x6bE7ab2FE5a48c86065f3c96563e1E0fdF7eA520);
7          bool returnValue=c.callcode('withdrawBalance');
8      }
9      function getBalance(address user) constant returns(
           uint) {
10         return userBalances[user];
11     }
12
13     function addToBalance() {
14         userBalances[msg.sender] += msg.value;
15     }
16 }
17
18 contract reentrant{
19     mapping (address => uint) userBalances;
20
21     function withdrawBalance() {
22     uint amountToWithdraw=userBalances[msg.sender];
23     if (!(msg.sender.call.value(amountToWithdraw)())){
           throw; }
24         userBalances[msg.sender]=0;
25 }
26 }
```

Figure 18: A false negative reentrant with a tricky bytecode

The hereby introduced pattern is also extendable for the other bug types. A contract which makes a callcode to a buggy contract will itself be affected by the introduced bug. Since these bugs are introduced via mechanisms which can not be modeled by the analyzer, the bugs remain undetected.

6.2.3.2 The CREATE opcode

New contracts can not only be created by external actors but also by other contracts themselves. Usually this is not modelled by analysis tools due to the computational complexity required to do so.

```
contract SendBack {
    mapping (address => uint) userBalances;
    function withdrawBalance() {
        D d = new D(4);
                uint amountToWithdraw = userBalances[msg.
                    sender];
                userBalances[msg.sender] = 0;
                msg.sender.send(amountToWithdraw);
        }
}

contract D {
    uint x;
    function D(uint a) payable {
        x = a;
    }
}
```

Figure 19: A contract exhibiting the mishandled exception bug along with the CREATE opcode

The contract at Figure 19 is a contract exhibiting a mishandled exception bug additionally creating another contract. Creating a new contract in Solidity can be done via the `new` keyword, which is compiled to the CREATE opcode. Some analysis tools crash when they encounter an opcode they can not model such as CREATE (see Section 7).

For the remaining bug types one can easily generate the test cases for the CREATE opcode case by inserting the `new` contract creating pattern in the contract code.

6.2.3.3 The SELFDESTRUCT opcode

Similarly to CREATE, the SELFDESTRUCT opcode (`suicide(address)` in Solidity level) also modifies the global state by zeroing out the account and sending the balance of it to the function argument `address`. SELFDESTRUCT is especially useful if one would like to get rid of an unused or buggy contract. It is quite non-trivial to model the effect of SELFDESTRUCT on the global state, since unlike all the other opcodes SELFDESTRUCT incurs negative gas costs since the operation frees up space on the

blockchain by clearing all of the contract's data. This negative gas deducts from the total gas cost of the transaction, thus if one is doing some clean-up operations first, SELFDESTRUCT can reduce gas costs [Woo14].

```
contract SendBack {
    mapping (address => uint) userBalances;
    function withdrawBalance() {
                uint amountToWithdraw = userBalances[msg.
                    sender];
                userBalances[msg.sender] = 0;
                msg.sender.send(amountToWithdraw);
                var c=SendBalanceHere(0
                    x6bE7ab2FE5a48c86065f3c96563e1E0fdF7eA520
                    );
                suicide(c);
        }
}

contract SendBalanceHere{

}
```

Figure 20: A contract exhibiting the mishandled exception bug along with the SELFDESTRUCT opcode

Similarly to the CREATE case one can generate test cases for the remaining vulnerabilities in a straightforward manner by adding the selfdestruct pattern to the vulnerable part of the contract.

6.3 Serpent contracts

An essential part of the benchmark is to test not only Solidity contracts. This part contains contracts that are essentially the same ones as presented in Section 4, the only difference is that they are implemented in Serpent. 4 different contracts exhibit the four different bugs, however here only one is shown to give a little glimpse into Serpent. It has a very similar syntax as Python, but it is also quite similar to Solidity.

```
data userBalances[][]

def getBalance():
        return userBalances[msg.sender][0]

def addToBalance():
        self.userBalances[msg.sender][0]=self.userBalances
            [msg.sender][0]+msg.value

def withdrawBalance():
        amountToWithdraw=self.userBalances[msg.sender][0]
        send(msg.sender,amountToWithdraw)
        self.userBalances[msg.sender][0]=0
```

Figure 21: A reentrant contract written in Serpent

6.3.1 Manual optimizations

As [Tin16] points out the Solidity optimizer under-optimizes contracts with certain patterns. For instance the outcome of a loop might be constant, which can be calculated at compile time, however Solidity compiler does calculate the whole loop even though it returns a constant value. This behaviour wastes gas. Most of the patterns described in [Tin16] can be optimized on the source code level with one exception. This exception is shown in Figure 22.

```
pragma solidity ^0.4.0;
contract manOpt {
    uint sum=0;
    function gas(uint x){
        for (uint i = 0 ; i < x ; i++){
            sum += i;
        }
    }
}
```

Figure 22: A contract which can be further optimized with respect to gas consumption

```
    POP
    POP
    POP
    POP
  tag 7
    JUMPDEST
    DUP3
    DUP2
    LT
    ISZERO
    PUSH [tag] 8
    JUMPI
    PUSH 1
    DUP1
    SLOAD
    DUP3
    ADD
    SWAP1
    SSTORE
  tag 9
    JUMPDEST
    PUSH 1
    ADD
    PUSH [tag] 7
    JUMP
  tag 8
    JUMPDEST
```

Figure 2: A bytecode pattern which can be optimized

The problem with the contract above is that at every iteration of the loop, variable `sum` is loaded from storage and stored. This means that opcodes SLOAD and SSTORE are executed `x` times. This is unwanted since these opcodes are really expensive and they should normally be executed once each: SLOAD before the loop and SSTORE after the loop.

Therefore contract developers are incentivized to manually optimize their own contract bytecodes in order to save a signifacant amount of gas and eventually ether. Such pattern is compiled down to bytecode similar to that of Figure 2. The loop starts at tag 7, where ISZERO pushes `0` to the stack if `i<x`, otherwise `1` is pushed. If the conditional does not hold the execution jumps out from the body of the loop to tag 8. Otherwise `sum` is loaded from storage and `i` is added to it. Variable `i` is incremented in tag 9, which jumps back to the evaluation of the conditional.

One could load variable `sum` before tag 7, and place it on the stack and treating it as a local variable inside the loop. All the necessary computations are done to `sum` variable and eventually one can store it in the permanent storage of the contract right after tag 8. After these modifications one should not forget to modify jump destinations accordingly since they may vary due to inserted or deleted opcodes.

These considerations could save a lot of gas for contract developers, hence the appearance of such manually optimized

contracts are not far from reality, therefore tools also should be able to handle them. For each vulnerability a manually optimized contract is created by adding the above described for loop pattern to the corresponding vulnerable contract.

6.4 Benchmark validity

Benchmark tests are not exactly like real contracts. The tests are derived from coding patterns observed in real contracts deployed on the Ethereum blockchain, but the majority of them are considerably simpler than real contracts. That is most real world contracts will be considerably harder to successfully analyze than this Ethereum benchmark test suite. On the other hand bytecode lengths are not far from realistic ones due to the overhead introduced by the Solidity compiler. Additionally benchmark contracts covers the vast majority of EVM opcodes. We think that the benchmark should serve for debugging as well as it should make testing easier by making bugs more transparent. Although the tests are based on real code, it is possible that some tests may have coding patterns that do not occur frequently in real code.

7 Results

We tested Oyente on our benchmark to investigate how well it manages to identify the various bugs on the test suite. Oyente is able to catch 4 different bugs (see Section 8), the same bugs the benchmark is tailored to test: timestamp dependency, reentrancy bug, concurrency attack and the callstack attack. Hereby we give an overview about the performance of Oyente on the benchmark.

7.1 Timestamp dependency

Oyente flags contracts timestamp dependent if they have feasible timestamp dependent execution paths which sends out money/ether [Luu]. They are correctly able to not raise false alarms in contracts not having the vulnerability (with the exception of vSTx (crash), gSMis and gSRe (segfault for both)). Oyente detects timestamp dependency bug correctly in Time.sol and trickSELFDESTRUCTTime.sol, however surprisingly in the Serpent equivalent of Time.sol timestamp dependency bug remains undetected. It gives a false positive for the context-sensitive cSTime, while for the rest of the contracts it gives a crash or a segfault.

7.2 Reentrancy bug

Solely focusing on contracts tailored for the detection of the reentrancy bug, we found that the only contract which is flagged vulnerable to the reentrancy bug is the simple Solidity contract (Re.sol), however surprisingly its counterpart written in Serpent (Re.se) is flagged as safe. All the other contracts testing for reentrancy are not tagged as vulnerable, except cSRe.sol and trickSELFDESTRUCTRe.sol.

7.3 Concurrency bug

Concurrency bug is a synonym for the TOD bug. Once again Oyente fails to detect the bug in its purest form written in Serpent, however the Solidity version is flagged being vulnerable properly. For the contract testing flow-sensitivity Oyente crashes, and for the rest of the contracts it assures the user about the absence of the bug. trickSELFDESTRUCTTX, fSTx and cSTx give true, while the rest either result in false or Oyente crashes.

7.4 Callstack attack

Due to the protocol changes made in October,2016 [But], callstack attack is not practically reachable anymore, therefore there is no need to include it anymore in the investigated attack list from the side of Oyente.

7.5 Results summary

Results in more detailed can be observed in Table 2. False indicates that Oyente flagged that contract being not vulnerable to a specific bug, while true shows that Oyente deems that the contract is vulnerable to a specific bug. In some cases (crash, segfault) Oyente was not able to provide meaningful output.

As the results indicate one can conclude that Oyente is flow- and value-sensitive, however it is not gas- nor context-sensitive. Moreover Oyente is not able to model and handle contracts creating other contracts (Oyente outputs: "unknown instruction: create"), or calling libraries via delegate call or callcode. On the other hand it works well with contracts exhibiting the SELFDESTRUCT opcode. Oyente seems to give incorrect outcomes for non-Solidity contracts (Serpent and manually optimized contracts as well).

Table 2 Oyente's performance on the benchmark

	Bugs			
	Callstack attack	Timestamp depen-dency	Concurrency	Reentrancy
Time.sol	true	true	false	false
Mis.sol	true	false	true	true
Re.sol	false	false	true	true
Tx.sol	false	false	false	true
Time.se	true	false	false	false
Mis.se	true	false	false	false
Re.se	true	false	false	false
Tx.se	true	false	false	false
fSTime	true	crash	crash	crash
fSMis	false	false	false	false
fSRe	false	false	false	false
fSTx	false	false	false	true
vSTime	true	crash	crash	crash
vSMis	true	false	false	false
vSRe	false	false	false	false
vSTx	false	crash	crash	crash
gSTime	true	segfault	segfault	segfault
gSMis	true	segfault	segfault	segfault
gSRe	false	segfault	segfault	segfault
gSTx	false	false	false	false
cSTime	true	true	false	true
cSMis	true	false	true	true
cSRe	false	false	true	true
cSTx	false	false	false	true
trickCALLCODETime	true	false	false	false
trickCALLCODEMis	true	false	false	false
trickCALLCODERe	true	false	false	false
trickCALLCODETx	true	false	false	false
trickCREATETime	true	false	false	false
trickCREATEMis	true	false	false	false
trickCREATERe	false	false	false	false
trickCREATETx	false	false	false	false
trickSELFDESTRUCTTime	true	true	false	false
trickSELFDESTRUCTMis	true	false	true	true
trickSELFDESTRUCTRe	false	false	true	true
trickSELFDESTRUCTTx	false	false	false	true
manOptTime	false	false	false	false
manOptMis	false	false	false	false
manOptRe	false	false	false	false
manOptTx	false	false	false	false

8 Related work

After June 17, 2016, when the DAO 1.0 heist happened, formal verification became one of the hottest topics not just in the Ethereum community, but also in other cryptocurrency communities [Rei16]. Suddenly developers, cryptocurrency enthusiasts, industry and academia realized that formal verification of smart contracts is an inevitable step in the process of writing secure contracts. Since then remarkable works were done in the field of formal verification of Ethereum smart contracts.

To catch up with the abundance of the possible attacks and bugs in Ethereum smart contracts, we recommend to look at [Nic16] by Nicola Atzei and Massimo Bartoletti. It gives a comprehensive, detailed survey of attacks and bugs, also providing descriptions of real exploits happened out in the wild. Moreover a taxonomy of vulnerabilities is provided as a reference for developers of smart contracts, to know and avoid common pitfalls. For a formal verification engineer or researcher it is a great place to start to see programming mistakes where smart contracts can go wrong.

The most important static analysis tool for Ethereum smart contracts is the one called Oyente from reasearchers Luu, Chu, Olickel, Saxena and Hobor [Luu16]. This work was a great inspiration for us to start our research in formally verifying smart contracts as it provided the first automated and scalable tool to analyze Ethereum smart contracts. Oyente focuses on detecting 4 common vulnerabilities in smart contracts, namely Transaction-Ordering Dependence, Timestamp Dependence, Mishandled Excepions and Reentrancy bugs, which eventually caused the fall of the DAO. Using the bytecode of the contracts, Oyente builds the control flow graph (CFG) of the contract. Afterwards infeasible execution paths are excluded using the Z3 SMT solver. They analyzed the Ethereum blockchain regarding these 4 bugs, and they could identify these problems in multiple high-profile smart contracts. Oyente could show that King of Ether is indeed susceptible to Mishandled Exceptions and that the DAO has reentrancy bugs. To present the applicability and scalability of Oyente, researchers conducted the first large scale static analysis of real Ethereum smart contracts by analyzing the first 1,459,999 blocks of the Ethereum blockchain. Their quantitative analysis clearly showed that how prevalent these vulnerabilities are among smart contracts: 5411 were flagged as having the mishandled exception bug, 3056 having the TOD vulnerability and 340 have the reentrancy bug out of 19,366 contracts. On the other hand these results are limited by the fact, that the tool is not sound, which implies that there could be numerous contracts, which are vulnerable, still flagged as secure. Also on the negative side Oyente is only capable of verifying 4 properties, so it can not be used as a general purpose contract verifier.

Another promising approach is to translate EVM bytecodes into F* [Kar16]. The advantage of this translation is to leverage the power and rich machinery of F* in verifying various properties about smart contracts. According to [Kar16] the translation was only done for a subset of the EVM bytecodes and also there is no proof of soundness for the EVM bytecode-F* translation. Unfortunately as of now there is still no available implementation of this approach.

Formal verification efforts for Ethereum have been done not only in academia, but also in the Ethereum community. The main people here are Yoichi Hirai and Sami Mäkelä. They together defined the EVM for interactive theorem provers (Isabelle) [Yoi16]. They are aiming manual, machine-checked proofs, not automized ones, therefore their approach seems to be less scalable than the others. Using this

theorem proving environment they were able to prove some properties of the Deed contract [Hir16]. The Deed contract is part of the Ethereum Name Service (ENS) where people can bid for their own Ethereum domains in an auction process in order to use shorter (eg. accountOfJohnSmith.eth) domains rather than the 160-bit not human-friendly addresses. Hirai and Mäkelä were able to prove one property about the Deed contract, namely that at one invocation of the contract only the registrar is able to decrease the balance. The approach is also applicable for other contracts, however lots of manual work is required to redefine contract representation, moreover currently the approach is not able to handle contracts with loops.

A loosely related work is the Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug by Zikai Alex Wen and Andrew Miller [Zik16]. This work uses bytecode level heuristics to identify contracts on the live Ethereum blockchain which contain the unchecked-send bug (in other works this bug is referenced as mishandled exception). This work clearly showed that developers are unaware of this weakness of Solidity, therefore soon after the publication of this article core Ethereum developers updated the compiler, such that it gives a warning to developers in case if they don't check the return values of the send function [thr16].

To get the full picture one has to mention the only existing dynamic analysis tool in the Ethereum community [Tea]. TestRPC is a tool to simulate full client behaviour and make developing Ethereum applications much faster by enabling developers to test the behaviour of their contract in the blockchain environment. It is extremely hard to simulate correctly the execution environment, not to mention the uncertainty and complexity of the blockchain behaviours such as interleaving execution paths. Since such behaviours are hard to simulate dynamically, it is better to stick to static analysis which is able to overapproximate such difficult-to-simulate behaviours.

References

[Sza94] Nick Szabó. “Smart Contracts”. In: (1994).
URL: https://web.archive.org/web/20011102030833/http://szabo.best.vwh.net:80/smart.contracts.html.

[CH95] Paul R. Carini and Michael Hind.
“Flow-Sensitive Interprocedural Constant Propagation”. In: (1995).
URL: https://pdfs.semanticscholar.org/6a71/a6cb31bc19d55b0828e1b16f89c3aa10d4f4.pdf.

[Dai98] Wei Dai. “B-money”. In: (1998).
URL: http://www.weidai.com/bmoney.txt.

[La05] Monica S. Lam and al.
“Context-Sensitive Program Analysis as Database Queries”. In: (2005).
URL: https://people.csail.mit.edu/mcarbin/papers/pods05.pdf.

[Dav08] Nenad Jovanovic Davide Balzarotti Marco Cova. “Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications”.
In: (2008). DOI: 10.1109/SP.2008.22.

[Nak08] Satoshi Nakamoto. “Bitcoin: A Peer-to-Peer Electronic Cash System”.
In: (2008). URL: https://bitcoin.org/bitcoin.pdf.

[Chr09] Gregor Snelting Christian Hammer.
“Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs”. In: (2009).
URL: https://pp.info.uni-karlsruhe.de/uploads/publikationen/hammer09ijis.pdf.

[Isi10] Alex Aiken Isil Dillig Thomas Dillig.
“Fluid Updates: Beyond Strong vs. Weak Updates”. In: (2010).
URL: https://www.cs.utexas.edu/~isil/esop-extended.pdf.

[Yic12] Mayur Naik Yichen Xie.
“Soundness and its Role in Bug Detection Systems”. In: (2012). URL: https://www.cs.umd.edu/~pugh/BugWorkshop05/papers/12-xie.pdf.

[DuP13] Neil DuPaul. “Static Testing vs. Dynamic Testing”. In: (2013).
URL: https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing.

[ES13] Ittay Eyal and Emin Gün Sirer.
“Majority is not Enough: Bitcoin Mining is Vulnerable”. In: (2013). URL: https://www.cs.cornell.edu/~ie53/publications/btcProcFC.pdf.

[Jon13] M. Jones.
“Static and dynamic testing in the software development life cycle”.
In: (2013).
URL: https://www.ibm.com/developerworks/library/se-static/.

[Dav14] Arthur Britto David Schwartz Noah Youngs.
“The Ripple Protocol Consensus Algorithm”. In: (2014).
URL: https://ripple.com/files/ripple_consensus_whitepaper.pdf.

[Woo14] Gavin Wood.
"Ethereum: A secure decentralised generalised transaction ledger".
In: (2014). URL: http://paper.gavwood.com/.

[But16a] Vitalik Buterin. "A Proof of Stake Design Philosophy". In: (2016).
URL: https://medium.com/@VitalikButerin/a-proof-of-stake-design-philosophy-506585978d51.

[But16b] Vitalik Buterin. "On Settlement Finality". In: (2016). URL:
https://blog.ethereum.org/2016/05/09/on-settlement-finality/.

[But16c] Vitalik Buterin. "Thinking About Smart Contract Security". In: (2016).
URL: https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/.

[Fan16] Kyle Croman Fan Zhang Ethan Cecchetti.
"Town Crier: An Authenticated Data Feed for Smart Contracts".
In: (2016). URL: https://eprint.iacr.org/2016/168.pdf.

[Har16] Saianeesh Keshav Haridas. "Random in Ethereum". In: (2016).
URL: http://otlw.co/2016/06/29/random-in-ethereum.html.

[Hir16] Yoichi Hirai.
"Formal Verification of Deed Contract in Ethereum Name Service".
In: (2016). URL: https://yoichihirai.com/deed.pdf.

[Jam16] Hudson Jameson. "FAQ: Upcoming Ethereum Hard Fork". In: (2016).
URL: http://hudsonjameson.com/2017-06-27-accounts-transactions-gas-ethereum/.

[Kar16] Antoine Delignat-Lavaud Karthikeyan Bhargavan.
"Short Paper: Formal Verification of Smart Contracts". In: (2016).
URL: http://www.cs.umd.edu/~aseem/solidetherplas.pdf.

[Luu16] Loi Luu. "Making Smart Contracts Smarter". In: (2016).
URL: https://eprint.iacr.org/2016/633.pdf.

[Nar16] Arvind Narayanan. *Bitcoin and Cryptocurrency Technologies*.
Princeton University Press, 2016.

[Nic16] Massimo Bartoletti Nicola Atzei.
"A survey of attacks on Ethereum smart contracts". In: (2016).
URL: https://eprint.iacr.org/2016/1007.pdf.

[Rei16] Christian Reitwiessner. "Smart Contract Security". In: (2016).
URL: https://blog.ethereum.org/2016/06/10/smart-contract-security/.

[Ste16] Matteo Maffei Stefano Calzavara Ilya Grishchenko.
"HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving". In: (2016).
URL: http://www.dais.unive.it/~calzavara/papers/eurosp16.pdf.

[thr16] Reddit thread. "Warning updates in Solidity compiler". In: (2016).
URL: https://www.reddit.com/r/ethereum/comments/4q65i8/nice_to_see_solidity_online_compiler_already/?st=j2ovztmw&sh=57677f88.

[Tin16] Xiaoqi Li et al. Ting Chen.
"Under-Optimized Smart Contracts Devour Your Money". In: (2016).
URL: https://arxiv.org/pdf/1703.03994.pdf.

[Yoi16] Sami Mäkelä Yoichi Hirai. "EVM defined for Isabelle". In: (2016).
URL: https://github.com/pirapira/eth-isabelle/blob/master/lem/evm.lem.

[Zik16] Andrew Miller Zikai Alex Wen.
"Scanning Live Ethereum Contracts for the "Unchecked-Send" Bug".
In: (2016).
URL: http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/.

[Cas17] Amy Castor. "A Short Guide to Bitcoin Forks". In: (2017). URL: https://www.coindesk.com/short-guide-bitcoin-forks-explained/.

[GP17] Cyril Grunspan and Ricardo Perez-Marco. "Double spend races".
In: (2017). URL: https://arxiv.org/pdf/1702.02867.pdf.

[Jam17] Hudson Jameson.
"Accounts, Transactions, Gas, and Block Gas Limits in Ethereum".
In: (2017). URL: http://hudsonjameson.com/2017-06-27-accounts-transactions-gas-ethereum/.

[Lor17] Ari Juels Lorenz Breidenbach Phil Daian.
"An In-Depth Look at the Parity Multisig Bug". In: (2017). URL: http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/.

[Ora17] Oraclize. "The random datasource: "A Scalable Architecture for On-Demand, Untrusted Delivery of Entropy"". In: (2017).
URL: https://blog.oraclize.it/the-random-datasource-a-scalable-architecture-for-on-demand-untrusted-delivery-of-entropy-7dbae6536322.

[Pal17] Santiago Palladino. "The Parity Wallet Hack Explained". In: (2017).
URL: https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

[Sir17] Emin Gün Sirer. "Parity's Wallet Bug is not Alone". In: (2017).
URL: http://hackingdistributed.com/2017/07/20/parity-wallet-not-alone/.

[You17] Joseph Young.
"Blockchain Can Save Banks Tens of Billions of Dollars a Year: BoE".
In: (2017). URL: https://cointelegraph.com/news/blockchain-can-save-banks-tens-of-billions-of-dollars-a-year-boe.

[Arz] Steven Arzt. *DroidBench 2.0*. URL:
https://github.com/secure-software-engineering/DroidBench.

[But] Vitalik Buterin. *EIP 150 protocol changes*.
URL: https://github.com/ethereum/EIPs/issues/150.

[Cal] Eric Somdahl Caleb James DeLisle. *Ethereum Classic Blog*.
URL: http://www.ethereumclassic.com/.

[Coi] CoinMKTCap. *CryptoCurrency Market Capitalizations*.
URL: https://coinmarketcap.com/.

[Cona] Consensys. *Technical Introduction to Events and Logs in Ethereum.* URL: https://media.consensys.net/technical-introduction-to-events-and-logs-in-ethereum-a074d65dd61e.

[con] A blockchain contract: *theRun contract.* URL: https://etherscan.io/address/0xcac337492149bdb66b088bf5914bedfbf78ccc18#code.

[Conb] Ethereum Blockchain Contract. *SendBack Contract.* URL: https://etherscan.io/address/0xc6b13d41866cba306fe9ddfafd580ac8d92bfb75#code.

[Der] Robby Dermody. *Counterparty official website.* URL: https://counterparty.io/.

[Dil] I¸sıl Dillig. *A Gentle Introduction to Program Analysis.* URL: https://www.cs.utexas.edu/~isil/dillig-plmw14.pdf.

[Etha] King of the Ether Throne. *Post-Mortem Investigation.* URL: https://www.kingoftheether.com/postmortem.html.

[Ethb] Etherscan. *Ethereum Average BlockTime Chart.* URL: https://etherscan.io/chart/blocktime.

[exa] Timestamp dependent example: *Lottopolo contract.* URL: https://etherchain.org/account/0x0155ce35fe73249fa5d6a29f3b4b7b98732eb2ed#code.

[exc] Mishandled exception: *King of The Ether Throne contract.* URL: https://etherscan.io/address/0x2464d1d97f8d0180cfad67bdb19bc30cca69dda0.

[Foua] Enterprise Foundation. *Ethereum Contract Accounts.* URL: https://etherscan.io/accounts/c.

[Foub] Ethereum Foundation. *DAO fork block.* URL: https://etherscan.io/block/1920000.

[Fouc] Ethereum Foundation. *Remix.* URL: https://github.com/ethereum/remix.

[Foud] Ethereum Foundation. *Remix development environment.* URL: https://remix.ethereum.org/.

[Foue] Ethereum Foundation. *Solidity Documentation.* URL: http://solidity.readthedocs.io/en/develop/.

[Hes] Tjaden Hess. *Internal calls and the JUMP opcode.* URL: https://ethereum.stackexchange.com/questions/15663/cannot-call-internal-function-via-call.

[Jba] Jbaylina. *The WHG has Returned 95% of the Funds and Now Hold Less Than $10 Million Worth of Rescued Funds.* URL: https://www.reddit.com/r/ethereum/comments/6povrc/the_whg_has_returned_95_of_the_funds_and_now_hold/?st=j5nwj0gy&sh=ba1921d9.

[Kwo] Jae Kwon. *Tendermint official website.* URL: https://tendermint.com/.

[Luu] Loi Luu. *Time Dependence Detection Error.* URL: https://github.com/melonproject/oyente/issues/132.

[Riz] Pete Rizzo.
In Formal Verification Push, Ethereum Seeks Smart Contract Certainty.
URL: http://www.coindesk.com/ethereum-formal-verification-smart-contracts/.

[Slo] Slock.it. *The DAO Token.* URL: https://etherscan.io/token/TheDAO.

[Tea] Ethereum JS Team. *A dynamic analysis tool for Ethereum.*
URL: https://github.com/ethereumjs/testrpc.

[Unk] Unknown. *Dao Hacker Contract on the Ethereum blockchain.*
URL: https://etherscan.io/address/0xc0ee9db1a9e07ca63e4ff0d5fb6f86bf68d47b89#internaltx.

[Ves] Peter Vessenes. *More Ethereum Attacks: Race-To-Empty is the Real Deal.*
URL: http://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/.

[Wen] Zikai Alex Wen. *Appendix A: Details on how we analyze the blockchain.*
URL: https://docs.google.com/document/d/1En0DjqmSpqVVxsdGAcJs4Rkw5IjxnUeTJMUVMhWum28/edit.

[Zha] YaNing Zhang. *RANDAO contract.*
URL: https://github.com/randao/randao.