**A survey of attacks on Ethereum smart contracts**

**以太坊智能合约攻击调查**

**Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli**

**Universita` degli Studi di Cagliari, Cagliari, Italy**

**[{atzeinicola,bart,t.cimoli}@unica.it](mailto:{atzeinicola,bart,t.cimoli}@unica.it)**

# Abstract

# 摘要

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.

智能合约是一种计算机程序，它可以由相互不信任的节点组成的网络正确执行，而不需要外部的可信机构。由于智能合约处理和转移具有相当价值的资产，因此除了正确执行这些合约外，还必须确保合约的执行安全，以防针对窃取或篡改资产的攻击。我们在以太坊中研究这个问题，以太坊是迄今为止最著名和最常用的智能合约框架。我们分析了以太坊智能合约的安全漏洞，提供了可能导致漏洞的常见编程陷阱的分类。我们展示了一系列攻击，这些攻击利用这些漏洞，允许对手窃取金钱或造成其他损害。

# 1 Introduction

# 1简介

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.

比特币（Bitcoin）是一种分散的加密货币，自2009年推出以来，其市值已达100亿美元，它的成功引起了业界和学术界的极大兴趣。各行各业以及各国政府都被加密货币底层技术区块链的“颠覆性”潜力所吸引。基本上，区块链是由对等网络的节点维护的附加数据结构。加密货币使用区块链作为公共账本，记录所有货币的转移，以避免双重支付。

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.

尽管比特币是区块链技术最典型的应用，但除了加密货币之外，还有其他应用：如金融产品和服务、跟踪各种财产的所有权、数字身份验证、投票等。一个热门话题是如何利用区块链技术来实现智能合约[34,54]。非常抽象地说，智能合约是相互不信任的参与者之间的协议，由区块链的共识机制自动执行，而不依赖可信的权威机构。

The most prominent framework for smart contracts is Ethereum [32], whose capitalisation has reached 1 billion dollars since its launch in July 2015. 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 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].[[1]](https://fanyi.baidu.com/" \l "_ftn1" \o ")

智能合约最突出的框架是以太坊[32]，自2015年7月推出以来，其市值已达10亿美元。在以太坊中，智能合约以计算机程序的形式呈现，用图灵完备语言编写。以太坊的共识协议规定了点对点网络的节点如何扩展区块链，其目标是确保合约的正确执行。该协议的关键见解之一是，要将新的数据块附加到区块链，节点必须参与抽奖，其中奖的概率与节点的计算能力成正比。这是一种激励机制确保，即使中奖的恶意节点试图附加一个执行合约不正确的区块，该区块最终也会从区块链中移除。尽管对协商一致协议的有效性提出了一些批评[37,44]，但最近的理论研究表明，只要诚实节点控制了网络的大部分计算能力，协商一致协议就具有安全性[39,52]。[[1]](https://fanyi.baidu.com/" \l "_ftn1" \o ")

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 ∼ $60M from a contract, but its effects were cancelled after an harshly debated revision of the blockchain.

以太坊智能合约的正确执行是其有效性的一个必要条件：否则，对手可能会篡改执行，例如将合法参与者的部分资金转移给自己。然而，仅仅执行的正确性并不足以使智能合约安全。实际上，以太坊智能合约中的几个安全漏洞是通过实际开发经验[35]和以太坊区块链上所有合约的静态分析[43]发现的。这些漏洞已被一些针对以太坊合约的实际攻击所利用，造成了金钱损失。这些攻击中最成功的一次成功地从一份合约中窃取了6000万美元，但在经过激烈辩论的区块链修订后，其后果被中和。

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.

有几个原因使得智能合约的实现在以太坊中特别容易出错。其中很大一部分与Solidity有关，Solidity是以太坊支持的高级编程语言。许多漏洞似乎是由Solidity的语义和程序员的直觉之间的不一致造成的。问题在于，Solidity虽然看起来像一种类型化的Javascript语言（除了异常和函数），但却以一种特殊的方式实现了其中的一些特性。同时，该语言不引入构造来处理特定方面，例如计算步骤被记录在公共区块链上的事实，其中它们可以被不可预测地重新排序或延迟。

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.

不安全智能合约泛滥的另一个主要原因是，已知漏洞的文档分散在多个来源，包括官方文档[8,22]、研究论文[24,35,43]以及互联网论坛[7]。对以太坊智能合约的漏洞和攻击进行全面、独立和更新的调查仍然缺乏。

**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]( co2.unica.it/ethereum).

**贡献**。本文首次系统地阐述了以太坊的安全漏洞及其高级编程语言Solidity。我们在分类法中组织漏洞的原因，其目的有两个：（i）作为智能合约开发人员的参考，以了解和避免常见的漏洞；（ii）作为研究人员的指南，以促进智能合约分析和验证技术的发展。对于分类法中造成漏洞的大多数原因，我们提出了一种利用漏洞的实际攻击（通常在实际合约中进行）。我们所有的攻击都在以太坊测试网上测试过，他们的代码可以在网上获得。<co2.unica.it/ethereum>

# 2 Background on Ethereum smart contracts

# 2以太坊智能合约背景

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.

以太坊[32]是一个分散的虚拟机，它根据用户的请求运行称为合约的程序。合约是用图灵完备字节码语言编写的，称为EVM字节码[56]。粗略地说，合约是一组函数，每个函数由一系列字节码指令定义。合约的一个显著特点是，它们可以将以太币（一种类似比特币的加密货币[46]）转移到用户和其他合约之间。

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.

用户将事务发送到以太坊网络，以便：（i）创建新合约；（ii）调用合约的功能；（iii）将以太币转移到合约或其他用户。所有的交易都记录在一个公共的、仅附加的数据结构上，称为区块链。区块链上的交易顺序决定了每个合约的状态，以及每个用户的余额。

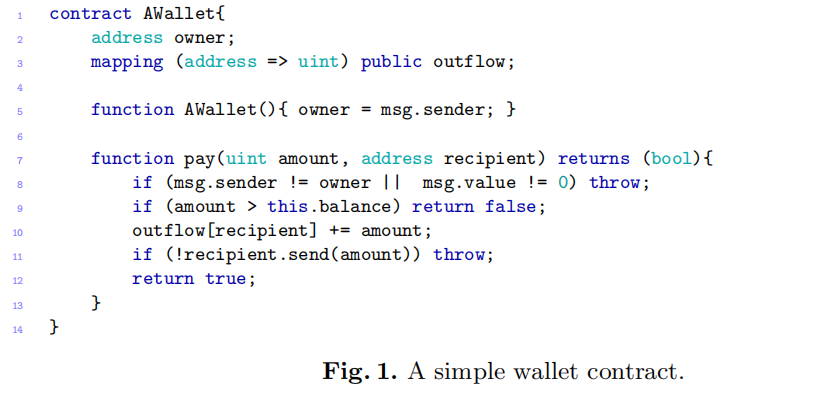
Since contracts have an economic value, it is crucial to guarantee that their execution is performed correctly. To this purpose, Ethereum does not rely on a trusted central authority: rather, each transaction is processed by a large network of mutually untrusted peers — called miners. Potential conflicts 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.

由于合约具有经济价值，因此保证合约的正确执行至关重要。为此，以太坊并不依赖于一个可信的中央机构：相反，每个事务都由一个由相互不信任的对等方（称为矿工）组成的大型网络来处理。合约执行中的潜在冲突（例如，由于失败或攻击）通过基于“工作证明”难题的协商一致协议解决。理想情况下，当对手不能控制网络的大部分计算能力时，合约的执行是正确的。

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.

共识协议的安全性依赖于诚实矿工是理性的假设，即矿工遵守协议比试图攻击协议更有利可图。为了使这一假设成立，矿工们在进行协议要求的（耗时的）计算时得到了一些经济奖励。这些激励措施的一部分是由用户在每次交易时支付的执行费提供的。这些费用限制了事务的执行步骤，因此可以防止拒绝服务攻击，即用户试图用耗时的计算压倒网络。

**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. 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 addressesto 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]](https://fanyi.baidu.com/" \l "_ftn2" \o ")

**编程智能合约**。我们通过一个小示例（图1中的AWallet）来说明合约，该示例实现了与所有者关联的个人钱包。我们使用Solidity（一种类似Javascript的编程语言）编译成EVM字节码，而不是直接将其编程为EVM字节码。直观地说，合约可以从其他用户那里接收以太币，其所有者可以通过pay功能将以太币（部分）发送给其他用户。哈希表outflow记录它将钱发送到的所有地址，并将总转移金额关联到每个地址。所有收到的以太币都由合约持有。它的数量自动记录在余额中：这是一个特殊的字段，程序员不能更改。[[2]](https://fanyi.baidu.com/" \l "_ftn2" \o ")

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.

合约由字段和函数组成。用户可以通过向以太坊节点发送适当的事务来调用函数。交易必须包括执行费（对于矿工），并且可能包括从调用者到合约的以太币转移。Solidity也有例外，但有一种特殊的行为。当抛出异常时，它将无法被捕获：执行停止，费用丢失，所有负面影响（包括以太币传输）都将恢复。

The function AWallet at line 5 is a constructor, run only once when the contract is created. The function pay sends amount (1wei = 10−18ether) 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).

第5行的AWallet函数是一个构造函数，在创建合约时只运行一次。函数pay将合约中的金额（1wei=10−18ether）发送给接收方。在第8行，如果调用方(msg.sender)不是所有者，或者是一些以太币(msg.value)附加到调用并转移到合约。由于异常会还原，所以这个以太会返回给调用者（但是调用者会损失费用）。在第9行，如果所需的以太币量不可用，则调用终止；在这种情况下，不需要异常地还原状态。在第10行，合约在将以太币转移给接收者之前更新登记。第11行使用的send函数有一些奇怪之处，例如，如果收件人是合约的话，它可能会失败（见第3节）。

**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.

**执行费**。理想情况下，每个函数调用都由以太坊网络中的所有矿工执行。矿工被激励去做这样的工作，由调用函数的用户支付执行费。除了用作奖励，执行费还可以防止拒绝服务攻击，即对手通过请求耗时的计算来降低网络速度。

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.

执行费用是根据gas和gas价格定义的，其产品代表用户执行代码所支付的成本。更具体地说，触发调用的事务指定用户愿意支付的gas限额，以及每单位gas的价格。大致来说，单位价格越高，矿工选择执行交易的可能性就越大。每个EVM作业消耗一定量的gas[56]，总费用取决于矿工执行的整个作业顺序。

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. 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.[[4]](https://fanyi.baidu.com/" \l "_ftn4" \o ")

矿工执行事务直到事务正常终止，除非抛出异常。如果事务成功终止，剩余的gas将返回给调用者，否则为事务分配的所有gas都将丢失。如果一个计算消耗了所有分配的gas，它将以“out of gas”异常终止-因此调用者将丢失所有gas。希望尝试拒绝服务攻击（例如，通过调用耗时函数）的对手应分配大量gas，并支付相应的费用。如果对手选择与市场一致的gas价格，矿工将执行交易，但攻击成本太高；否则，如果价格太低，矿工将不会执行交易。[[4]](https://fanyi.baidu.com/" \l "_ftn4" \o ")

**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, 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.[[5]](https://fanyi.baidu.com/" \l "_ftn5" \o ")

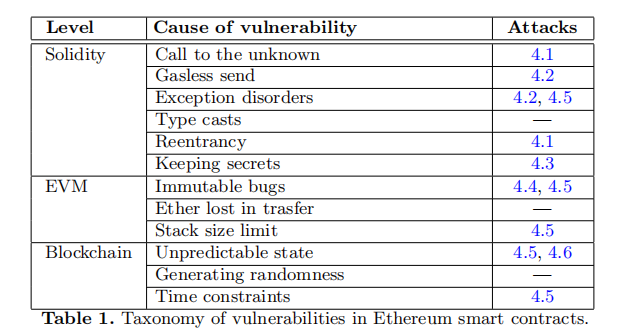
**采矿过程**。矿工将用户发送的交易分组到区块中，并尝试将它们附加到区块链中以收取相关费用。只有那些满足一组给定条件的区块（统称为有效性）才能附加到区块链。特别是，其中一个条件需要解决一个中等难度的“工作证明”难题，这取决于前一个块和新块中的事务。问题的难度是动态更新的，因此平均挖掘率是每12秒1个块。[[5]](https://fanyi.baidu.com/" \l "_ftn5" \o ")

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).

当一个矿工解决了这个难题并向网络广播了一个新的有效区块时，其他矿工放弃了他们的尝试，通过添加新区块来更新区块链的本地副本，并开始在其上“挖掘”。解决这个难题的矿工会得到新区块交易费用的奖励（还有一些区块奖励以太币）。

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. 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.[[6]](https://fanyi.baidu.com/" \l "_ftn6" \o ")

可能会发生两个（或更多）矿工几乎同时解决这个难题。在这种情况下，区块链分叉成两个（或更多）分支，新块指向同一父块。共识协议规定矿工延长最长的分支。因此，即使两个分支都可以暂时地继续存在，但最终分叉将被解决为最长的分支。只有其中的交易将成为区块链的一部分，而最短分支中的交易将被丢弃。奖励机制受[52]中GHOST 协议的启发，将全部费用分配给最长分支中区块的矿工，将部分费用分配给挖掘废弃分支块的矿工。例如，假设块A和块B具有相同的父块，并且矿工在块A的上方附加一个新块。矿工可以将部分报酬捐赠给“Uncle块”B的矿工，以增加其分支在分叉解析过程中的权重。[[6]](https://fanyi.baidu.com/" \l "_ftn6" \o ")

**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.

**将Solidity编译成EVM字节码**。尽管合约以函数集合的形式呈现，但是EVM字节码不支持函数。因此，Solidity编译器转换合约，以便它们的第一部分实现函数调度机制。更具体地说，每个函数都由基于其名称和类型参数的签名唯一标识。在函数调用时，这个签名作为输入传递给被调用的合约：如果它匹配某个函数，则执行跳转到相应的代码，否则跳转到回退函数。这是一个没有名字和参数的特殊函数，可以任意编程。当合约被传递一个空签名时，也会执行回退功能：例如，当向合约发送以太币时。

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.

Solidity具有三种不同的构造，可以从另一个合约调用一个合约，这也允许发送以太币。所有这些构造都是使用相同的字节码指令编译的。结果是，相同的行为可以通过几种方式实现，第3节详细介绍了一些细微的差异。

# 3 A taxonomy of vulnerabilities in smart contracts

# 3智能合约漏洞分类

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.

在本节中，我们将系统化以太坊智能合约的安全漏洞。我们根据引入漏洞的级别将漏洞分为三类（Solidity、EVM字节码或区块链）。此外，我们还通过一小段代码在Solidity级别上说明了每个漏洞。所有这些漏洞都可以被利用（实际上，大多数漏洞已经被利用）进行攻击，例如从合约中窃取资金。表1总结了我们的漏洞分类法，第4节给出了攻击的链接。

**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]](https://fanyi.baidu.com/" \l "_ftn8" \o ")

– 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]](https://fanyi.baidu.com/" \l "_ftn9" \o ")

– besides the primitives above, one can also use a direct call as follows:

**调用未知**。Solidity中用于调用函数和传输ether的一些原语可能会产生调用被调用方/接收方的回退函数的不利方面。我们在下面举例说明。

–call调用一个函数（另一个合约的函数，或它本身的函数），并将以太网传输给被调用方。E、 例如，可以调用合约c的函数ping，如下所示：

c。调用值（金额）（字节4（sha3（“ping（uint256）”）），n）；

其中被调用函数由其散列签名的前4个字节标识，amount确定需要向c传输多少个wei，n是ping的实际参数。值得注意的是，如果具有给定签名的函数在地址c处不存在，则执行c的回退函数。[[8]](https://fanyi.baidu.com/" \l "_ftn8" \o ")

–send用于将以太币从正在运行的合约转移到某个收件人r，如r.send（amount）。传输以太后，send执行收件人的回退。与send相关的其他漏洞在“异常紊乱”和“无气send”中有详细说明。

–delegatecall与call非常相似，不同之处在于被调用函数的调用是在调用者环境中运行的。例如，执行c.delegatecall（bytes4（sha3（“ping（uint256）”）），n），如果ping包含字段this，它指的是调用者的地址而不是c，如果以太传输到某个接收者d-通过d.send（amount）-以太从调用者余额中获取（例如，参见第4.6节中的攻击）。[[9]](https://fanyi.baidu.com/" \l "_ftn9" \o ")

–除了上述原语之外，还可以使用如下直接调用：

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.

第一行声明Alice的合约接口，最后两行包含Bob的合约：其中，pong通过直接调用调用Alice的ping。现在，如果程序员错误地键入了合约Alice的接口（例如，将参数的类型声明为int，而不是uint），并且Alice没有具有该签名的函数，那么对ping的调用实际上会导致对Alice的fallback函数的调用。

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.

fallback函数并不是唯一一段可以意外执行的代码：在第9页的漏洞“类型转换”和第11页的“不可预测状态”中报告了其他情况。

**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:

**异常障碍**。在Solidity中，有几种情况可能引发异常，例如：（i）执行耗尽gas；（ii）调用堆栈达到其极限；（iii）执行命令throw。然而，Solidity在处理异常的方式上并不统一：有两种不同的行为，这取决于合约如何称呼对方。例如，



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 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.

现在，假设某个用户调用了Bob的pong，而Alice的ping抛出了一个异常。然后，执行停止，整个事务的不利方面恢复。因此，字段x在事务之后包含0。现在，假设Bob通过调用调用ping。在这种情况下，只有调用的不利方面被还原，调用返回false，执行继续。因此，x在事务之后包含2。

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

更一般地说，当抛出异常时，假设存在嵌套调用链。然后，异常处理如下：

– 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;

–如果链中的每个元素都是直接调用，则执行停止，所有不利方面（包括以太币转移）都会恢复。此外，消耗所发起交易分配的所有gas；

– 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. Further, all the gas allocated by the call is consumed.[[10]](https://fanyi.baidu.com/" \l "_ftn10" \o ")

–如果链中至少有一个元素是调用（delegatecall和send的情况类似），则异常会沿着链传播，恢复被调用合约中的所有不利方面，直到到达调用为止。从这一点开始，执行继续，调用返回false。此外，调用分配的所有gas都被消耗掉。[[10]](https://fanyi.baidu.com/" \l "_ftn10" \o ")

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 ∼ 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).

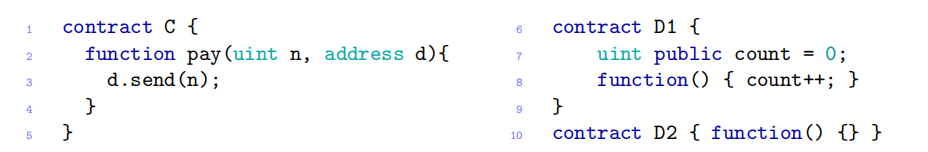
处理异常情况的不规范可能影响合约的安全。例如，认为以太币转移成功仅仅因为没有异常就可能导致攻击（例如，见第4.2节和第4.5节）。[14]中的定量分析表明，∼28%的合约不控制调用/发送调用的返回值（但是请注意，没有这些检查并不一定意味着存在漏洞）。

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. 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.[[11]](https://fanyi.baidu.com/" \l "_ftn11" \o ")

无Gas发送。当使用send功能将以太币传输到合约时，可能会导致gas耗尽异常。这对于程序员来说可能是非常意外的，因为传输以太币通常与执行代码无关。这一异常背后的原因很微妙。首先，注意c.send（amount）的编译方式与带有空签名的调用的编译方式相同，但是被调用方可用的gas单位的实际数量始终是2300。现在，由于调用没有签名，它将调用被调用方的回退函数。然而，2300个gas单元只允许执行一组有限的字节码指令，例如那些不改变合约状态的指令。在任何其他情况下，调用都将以gas耗尽异常结束。[[11]](https://fanyi.baidu.com/" \l "_ftn11" \o ")

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.

我们通过一个小例子来说明send的行为，其中包括一个通过函数pay发送ether的contract C和两个接收者D1、D2。

There are three possible cases to execute pay:

– n6= 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 6= 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∈{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 ≥ 0.4.0, the behaviour is the same as in one of the previous two cases, according whether d = D1 or d = D2.

– n！=0，d=D1。send-in C失败，出现gas不足异常，因为2300个气体单位不足以执行状态更新D1的回退。

– n！=0，d=D2。C发送成功，因为2300单位的gas足以执行D2的空回退。

– n！=0和d∈{D1，D2}。对于<0.4.0的编译器版本，由于gas不足以执行任何回退，甚至不足以执行空回退，因此send-in C会失败，并出现一个out-of-gas异常。对于≥0.4.0的编译器版本，行为与前两种情况中的一种相同，取决于d=D1还是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.

综上所述，通过send发送以太币在两种情况下成功：当接收者是一个具有空回退的合约时，或者当接收者是一个用户时。

**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:

**类型转换**。Solidity编译器可以检测一些类型错误（例如，将整数值赋给string类型的字段）。类型也用于直接调用：调用方必须声明被调用方的接口，并在执行调用时将被调用方的地址强制转换为该接口。例如，再次考虑对ping的直接调用：

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.

pong的签名通知编译器c遵循Alice接口。但是，编译器只检查接口是否声明了函数ping，而不检查：（i）c是 Alice合约的地址；（ii）Bob声明的接口是否与Alice的实际接口匹配。类似的情况也发生在显式类型转换中，例如Alice（c）.ping（），其中c是地址。

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:

合约可以进行类型检查的事实可能会欺骗程序员，使他们相信检查（i）和（ii）中的任何错误都会被检测到。此外，即使存在这样的错误，合约也不会在运行时抛出异常。实际上，直接调用是在用于编译调用的同一EVM字节码指令中编译的（异常管理除外）。因此，在类型不匹配的情况下，在运行时可能会发生三种不同的情况：

– if c is not a contract address, the call returns without executing any code;[[12]](https://fanyi.baidu.com/" \l "_ftn12" \o ")

– 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.

–如果c不是合约地址，则调用返回而不执行任何代码；[[12]](https://fanyi.baidu.com/" \l "_ftn12" \o ")

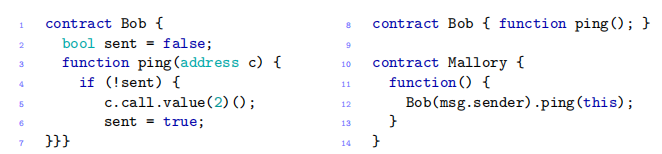
–如果c是具有与Alice的ping相同签名的函数的任何合约的地址，则该函数被执行。

–如果c是一个没有与Alice的ping签名匹配的函数的合约，则执行c的回退。

在所有情况下，都不会抛出异常，并且调用方不知道错误。

**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:

**重入状态**。事务的原子性和顺序性可能会使程序员相信，当调用非递归函数时，它不能在终止前重新输入。但是，情况并非总是如此，因为回退机制可能允许攻击者重新进入调用方函数。这可能导致意外行为，也可能导致调用循环，最终消耗所有gas。例如，假设Bob合约已经在区块链上，当攻击者发布Mallory 合约时：

The function ping in Bob is meant to send exactly 2wei 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 2wei 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.

Bob中的函数ping旨在使用一个带有空签名且没有gas限制的调用将2 wei精确地发送到某个地址c。现在，假设ping是用Mallory的地址调用的。如前所述，调用的副作用是调用Mallory的fallback，后者反过来又调用ping。因为字段sent还没有设置为true，所以Bob再次将2 wei发送给Mallory，并再次调用他的回退，从而开始一个循环。当执行最终耗尽gas时，或者当达到堆栈限制时（请参阅第11页的“堆栈大小限制”漏洞），或者当Bob已经耗尽所有的以太币时，此循环结束。在所有情况下都会引发异常：但是，由于调用不会传播异常，因此只会还原最后一个调用的效果，从而使以前所有的以太传输都有效。

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).

此漏洞存在于函数ping不可重入的事实中，即如果在其终止之前调用，它可能会出现错误行为。值得注意的是，2016年6月造成巨大以太币损失的“DAO攻击”利用了此漏洞（有关攻击的更多详细信息，请参阅第4.1节）。

**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.

**保守秘密**。合约中的字段可以是公共的，即每个人都可以直接读取，也可以是私有的，即其他用户/合约不能直接读取。尽管如此，将字段声明为私有并不能保证其机密性。这是因为，要设置一个字段的值，用户必须向矿工发送一个合适的事务，矿工随后将其发布在区块链上。由于区块链是公开的，每个人都可以检查交易的内容，并推断字段的新值。

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).

许多合约，例如那些实施多人游戏的合约，要求某些字段保密一段时间：例如，如果一个字段存储了一个球员的下一步行动，透露给其他球员可能有利于他们选择下一步行动。在这种情况下，为了确保某个字段在某个事件发生之前保密，合约必须利用适当的加密技术，例如定时承诺[25,29]（见第4.3节）。

**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]](https://fanyi.baidu.com/" \l "_ftn13" \o ")

不变Bug。一旦合约在区块链上发布，就不能再修改了。因此，用户可以相信，如果合约实现了他们预期的功能，那么它的运行时行为也将是预期的，因为这是由共识协议保证的。缺点是，如果合约包含bug，就没有直接的方法来修补它。因此，程序员必须在合约的实现变更或终止合约的方法[45]——尽管这与以太坊的原则是否一致还存在争议。[[13]](https://fanyi.baidu.com/" \l "_ftn13" \o ")

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].

漏洞的不变性已经在各种攻击中被利用，例如窃取以太币，或使任何用户都无法恢复以太币（见第4.4节和第4.5节）。在所有这些袭击中，都没有恢复的可能。唯一的例外是从“The Dao攻击”中恢复过来。对策是区块链的硬叉，基本上消除了攻击中涉及的交易的影响[15]。这个解决方案并没有得到整个以太坊社区的同意，因为它与迄今为止声称的“代码即法律”原则形成了对比。因此，部分矿工拒绝掏腰包，并创建了另一个区块链[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.

以太币在传输过程中丢失。发送以太时，必须指定收件人地址，该地址采用160位序列的形式。但是，其中许多地址是孤立的，即它们与任何用户或合约都没有关联。如果某个以太被发送到孤立地址，它将永远丢失（请注意，无法检测地址是否为孤立地址）。由于丢失的以太无法恢复，程序员必须手动确保收件人地址的正确性。

**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.

**堆栈大小限制**。每次合约调用另一个合约（甚至通过this.f（）调用自身）时，与事务相关联的调用堆栈都会增加一帧。调用堆栈被限制为1024帧：当达到这个限制时，进一步的调用抛出一个异常。

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.

直到2016年10月18日，有可能利用这一事实进行如下攻击。对手首先生成一个几乎满的调用堆栈（通过一系列嵌套调用），然后调用受害者的函数，再调用一次就会失败。如果受害者的合约未能妥善处理例外情况，对手可以设法成功地进行攻击。此漏洞可与其他漏洞一起被利用：例如，在第4.5节中，我们通过利用“异常无序”和“堆栈大小限制”漏洞来实现恶意合约。

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 redefined 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].

以太坊区块链[1]的一个硬叉解决了这一漏洞原因。分叉改变了多条EVM指令的开销，重新定义了call和delegatecall的耗气量计算方法。在分叉之后，调用者最多可以分配63/64的gas：因为目前每个块的gas限制是∼4,7M单位，这意味着调用堆栈的最大可到达深度总是小于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.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.

不可预知的状态。合约的状态由其字段的值和余额决定。一般来说，当用户向网络发送一个事务以调用某个合约时，他不能确定该事务将以发送该事务时合约的相同状态运行。这可能是因为，其他事务已经更改了合约状态。即使用户的速度足够快，能够第一个发送事务，也不能保证这样的事务将是第一个运行的。实际上，当矿工将事务分组到块中时，他们不需要保留任何顺序；他们也可以选择不包括某些事务。

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.

还有另一种情况，用户可能不知道其事务将在其中运行的实际状态。这发生在区块链分叉的情况下（参见第2节）。回想一下，当两个矿工同时发现一个新的有效区块时，区块链分成两个分支。一些矿工将尝试在其中一个分支上添加新的块，而另一些矿工将在另一个分支上工作。不过，一段时间后，只有最长的分支会被视为区块链的一部分，而最短的分支会被放弃。最短分支中的交易将被忽略，因为它不再是区块链的一部分。因此，相信合约处于某种状态，可能是用户发布新事务（例如，向其他用户发送以太）的决定性因素。但是，稍后这种状态可能会恢复，因为导致这种状态的事务可能恰好位于fork的最短分支中。

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).

在某些情况下，不知道事务的运行状态可能会导致漏洞。 例如，调用可以动态更新的合约时就是这种情况。确实需要注意的是，尽管合约的代码一旦在区块链上发布就不能更改，但是通过一些预先考虑，可以制定一个合约，其组件可以根据其所有者的请求进行更新。稍后，所有者可以将此类合约链接到恶意组件，例如窃取调用者的以太（参见第4.6节中的攻击）。

**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.

**产生随机性**。EVM字节码的执行是确定的：在没有错误行为的情况下，执行事务的所有矿工将得到相同的结果。因此，为了模拟非确定性选择，许多合约（例如彩票、游戏等）生成伪随机数，其中初始化种子是为所有矿工唯一选择的。

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.

一个常见的选择是为这个种子（或为随机数本身）获取哈希或某个块的时间戳，该块将在未来的给定时间出现在区块链中。由于所有矿工对区块链都有相同的看法，因此在运行时，每个人的价值都是相同的。显然，这是一种生成随机数的安全方法，因为未来块的内容是不可预测的。然而，由于矿工控制将哪些事务放入一个块中以及按什么顺序进行，恶意矿工可能会试图精心设计他的块，从而使伪随机生成器的结果产生偏差。[30]中对比特币区块链随机性的分析表明，攻击者控制了网络的少数采矿能力，可以投资50比特币，显著偏离结果的概率分布；最近的研究[49]证明，在资源更有限的情况下，这也是可能的。

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.

该问题的替代解决方案基于定时承诺协议[25,29]。在这些协议中，每个参与者选择一个秘密，然后将其摘要传递给其他参与者，并支付押金作为担保。后来，参与者必须要么透露他们的秘密，要么失去他们的存款。然后结合所有参与者的秘密来计算伪随机数[18,19]。同样，在这种情况下，对手可能会通过不透露他的秘密来影响结果：然而，这样做会导致失去他的存款。然后协议可以设置存款的数量，这样不泄露秘密是一种不合理的策略。

**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.

**时间限制**。许多应用程序使用时间约束来确定在当前状态下允许（或强制）哪些操作。通常，时间约束通过使用块时间戳来实现，块时间戳由所有矿工商定。

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. 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.[[14]](https://fanyi.baidu.com/" \l "_ftn14" \o ")

合约可以检索挖掘块的时间戳；块中的所有事务共享相同的时间戳。这保证了执行后与合约状态的一致性，但也可能使合约受到攻击，因为创建新块的矿工可以任意选择时间戳。如果一个矿工持有合约上的股份，他可以通过为他正在开采的区块选择一个合适的时间戳来获得优势。在第4.5节中，我们展示了利用此漏洞的攻击。[[14]](https://fanyi.baidu.com/" \l "_ftn14" \o ")

# 4 Attacks

# 4攻击

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

我们现在举例说明一些攻击—其中许多攻击是受真实用例启发的—这些攻击利用了第3节中介绍的漏洞。

## 4.1 The DAO attack

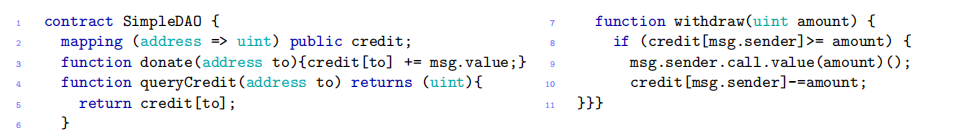
## 4.1 The DAO攻击

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

DAO[23]是一个实施众筹平台的合约，在2016年6月18日遭到攻击之前筹集了1.5亿美元[4]。一名攻击者成功地将6000万美元置于他的控制之下，直到区块链的硬叉消除了参与攻击的交易的影响。

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]](https://fanyi.baidu.com/" \l "_ftn15" \o ")

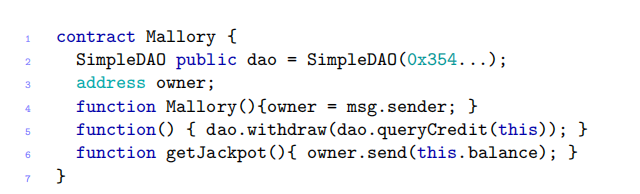
我们现在提供一个DAO的简化版本，它与原始版本有一些相同的漏洞。然后我们展示了利用它们的两个攻击。[[15]](https://fanyi.baidu.com/" \l "_ftn15" \o ")

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

SimpleDAO允许参与者根据自己的选择向基金合约捐赠以太币。合约方可以提取资金。

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.

攻击#1。这种攻击与实际DAO上使用的攻击类似，允许对手从SimpleDAO中窃取所有以太币。攻击的第一步是发布合约。

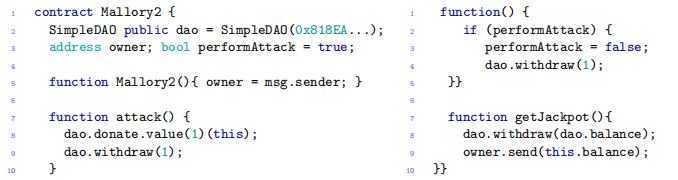


Then, the adversary donates some ether for Mallory, and invokes Mallory's fallback. The fallback function invokes withdraw, which transfers the ether to Mallory. Now, the function call used to this purpose has the side effect of invoking Mallory's fallback again (line 5), which maliciously calls back 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.

然后，对手为Mallory捐赠了一些以太币，并调用了Mallory的回退。fallback函数调用draw，它将以太转移到Mallory。现在，用于此目的的函数调用的副作用是再次调用Mallory的fallback（第5行），后者恶意地回调draw。注意，draw在更新credit字段之前被中断：因此，第8行的检查再次成功。因此，DAO第二次向Mallory发送credit，然后再次调用他的fallback，以此类推，直到发生以下事件之一：（i）gas耗尽，或（ii）调用堆栈已满，或（iii）DAO的余额变为零。攻击的总体效果是，通过一系列的攻击，对手可以从DAO中窃取所有的以太币。注意，对手可以通过在发起事务中提供更多gas来延迟gas耗尽异常，因为第9行的调用没有指定气体限制。

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., 1wei). Then, the adversary invokes attack to donate 1wei to herself, and subsequently withdraws it. The function withdraw checks that the user credit is enough, and if so it transfers the ether to Mallory2.

攻击#2。另外，我们的第二次攻击允许对手从SimpleDAO窃取所有以太，但它只需要调用两次fallback函数。第一步是发布Mallory2，为其提供少量以太币（例如1wei）。然后，对方发动攻击，将1wei捐给自己，随后撤回。函数draw检查用户credit 是否足够，如果足够，它将以太币转移到Mallory2。

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 1wei 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 (2256−1)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.

与上一次攻击一样，call调用Mallory2的fallback，后者反过来调用withdraw。在这种情况下，取款在更新credit之前被中断：因此，第8行的检查再次成功。因此，the DAO第二次将1wei发送到Mallory2，并再次调用他的回退。但是这次回退什么也不做，嵌套调用开始关闭。结果是Mallory2的credit被更新了两次：第一次为零，第二次为（2256-1）wei，因为下溢。为了完成攻击，Mallory2调用getJackpot，它从SimpleDAO窃取所有的以太币，并将其转移给Mallory2的所有者。

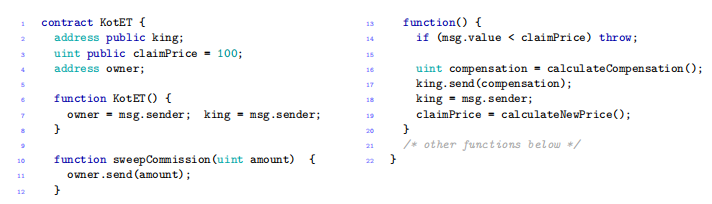
Both attacks were possible because SimpleDAO sends the specified amount of ether before decreasing the credit. Overall, the attacks exploit the “call to the unknown”, and “reentrancy” vulnerabilities. The first attack is more effective with a larger investment, while the second one is already rewarding with an investment of just 1wei (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.

这两种攻击都是可能的，因为SimpleDAO在降低信用之前发送了指定数量的以太币。总的来说，这些攻击利用了“对未知的调用”和“可重入性”漏洞。第一次攻击更有效，投资较大，而第二次攻击的回报仅为1wei（以太币的最小部分）。请注意，第二种攻击也适用于SimpleDAO的一个变体，它检查第9行的call的返回代码，并在失败时抛出异常。

## 4.2 King of the Ether Throne

## 4.2以太之王

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. We discuss a simplified version of the game (with the same vulnerabilities), implemented as the contract **KotET**:[[16]](https://fanyi.baidu.com/" \l "_ftn16" \o ")

“以太之王”[16,17]是一款玩家争夺“以太王”头衔的游戏。如果有人想成为国王，他必须付给现任国王一些以太币，再加上一小笔合约费。为王的奖品单调地增加。我们讨论了游戏的简化版本（具有相同的漏洞），实现为KotET合约：

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.

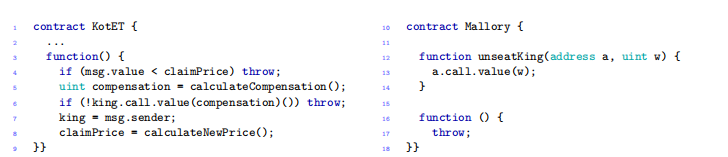
每当玩家发送消息值（msg.value ）到合约，他还触发了执行KotET's 的回退函数。回退首先检查发送的以太是否足以购买头衔：如果没有，则抛出异常（恢复以太传输）；否则，玩家将被接受为新国王。在这一点上，一个补偿是发送给解雇国王，玩家加冕。msg.value 和赔偿金由合约保管。KotET的所有者可以通过佣金收回合约中累积的以太币。

Apparently, the contract may seem honest: in fact, it is not, because not checking the return code of send may result in stealing ether. 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.[[17]](https://fanyi.baidu.com/" \l "_ftn17" \o ")

显然，合约看起来可能是诚实的：事实上并非如此，因为不检查send的返回码可能导致盗窃。事实上，由于send配备了少量的gas（请参阅“gasless send”漏洞），因此如果国王的地址是一个具有昂贵后备功能的合约，那么第17行的send将失败。在这种情况下，由于send不会传播异常（参见“异常无序”），因此补偿由合约保留。[[17]](https://fanyi.baidu.com/" \l "_ftn17" \o ")

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:

现在，假设一个诚实的程序员想要实现KotET的公平变体，在第6行用call替换send，并检查其返回代码：

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 unseat King 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.

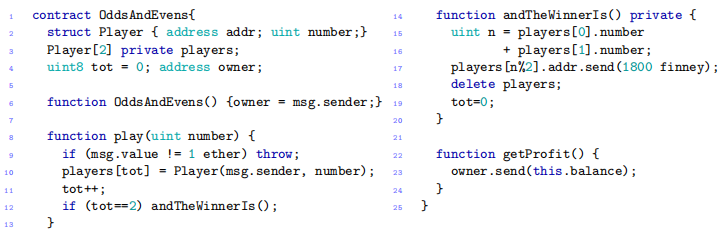
此变体比以前的版本更可信，但易受拒绝服务攻击的攻击。要了解原因，请考虑一个攻击者Mallory，它的回退只是抛出一个异常。敌人用适量的以太币解封，Mallory就作了新王。在这一点上，没有其他人可以得到他的王冠，因为每次KotET 试图发送赔偿Mallory，他就会抛出一个异常，阻止玩家加冕成功。

## 4.3 Multi-player games

## 4.3多人游戏

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]](https://fanyi.baidu.com/" \l "_ftn18" \o ")

考虑一个合约，它在两个参与者之间实现了一个简单的“奇数偶数”游戏。每个玩家选择一个数字：如果和是偶数，第一个玩家赢，否则第二个玩家赢。[[18]](https://fanyi.baidu.com/" \l "_ftn18" \o ")

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 1ether 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 andTheWinner is to send 1.8ether to the winner. The remaining 0.2ether are kept by the contract, and they can be collected by the owner via getProfit.

合约记录了两名运动员在赛场上的赌注。因为这个字段是私有的，所以其他合约不能直接读取它。要加入游戏，每个玩家必须在调用函数play时转移1以太币。如果转移的金额不同，则通过抛出异常（第9行）将其发送回玩家。一旦第二个玩家加入了游戏，合约就生效了，玩家将向获胜者发送1.8以太。剩余的0.2元由合约保管，所有者可通过getProfit收取自己的所获。

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. This attack exploits the “keeping secrets” vulnerability.[[19]](https://fanyi.baidu.com/" \l "_ftn19" \o ")

一个对手可以进行攻击，这总是让他赢得比赛。为此，对手模仿第二个玩家，等待第一个玩家下注。现在，尽管字段players 是私有的，但是对手可以通过检查他加入游戏的区块链交易来推断第一个玩家的赌注。然后，对手可以通过使用适当的赌注调用play来赢得游戏。此攻击利用“保密”漏洞。[[19]](https://fanyi.baidu.com/" \l "_ftn19" \o ")

## 4.4 Rubixi

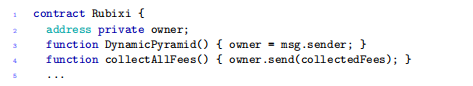
## 4.4 Rubixi合约

Rubixi [2,9] is a contract which implements a Ponzi scheme, a fraudulent highyield 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.

Rubixi[2,9]是一个实施庞氏骗局的合约，庞氏骗局是一种欺诈性的高收益投资计划，参与者从新来者的投资中获利。此外，合约所有人可以收取一些费用，在投资时支付给合约。以下攻击允许对手利用“不可变的bug”漏洞从合约中窃取一些以太币。

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.

在合约开发过程中的某个时候，它的名字从DynamicPyramid变成了Rubixi。然而，程序员忘记了相应地更改构造函数的名称，然后它就变成了一个任何人都可以调用的函数（相反，在创建合约时，构造函数只运行一次）。DynamicPyramid（）函数设置所有者地址；所有者可以通过collectallifes提取利润。

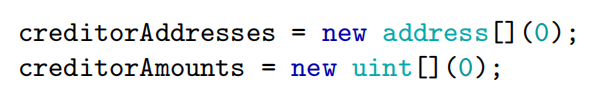
After this bug became public, users started to invoke DynamicPyramid in order to become the owner, and so to withdraw the fees.

在这个bug公开之后，用户开始调用DynamicPyramid以成为所有者，从而收回费用。

## 4.5 GovernMental

## 4.5 GovernMental合约

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:

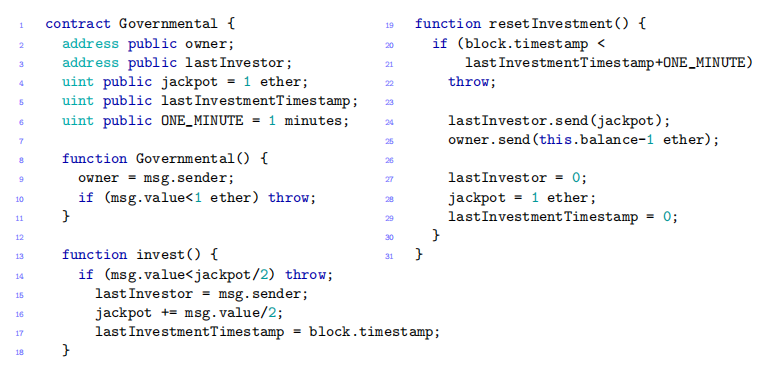
GovernMental [12，13]是另一个有缺陷的庞氏骗局。要加入该计划，参与者必须向合约中加入一定数量的以太币。如果12小时内没有人加入该计划，最后一名参与者将获得合约中的所有费用（玩家保留的费用除外）。参与者列表及其credit存储在两个数组中。当12小时到期时，最后一个参与者可以认领这笔钱，数组被清除如下：

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]](https://fanyi.baidu.com/" \l "_ftn20" \o ")

从这个Solidity代码片段获得的EVM代码逐个清除数组的每个位置。在某一点上，GovernMental 的参与者名单增长得如此之长，以至于清除数组所需的gas量将超过单个交易所允许的最大gas量[11]。从那时起，任何清除数组的尝试都失败了。[[20]](https://fanyi.baidu.com/" \l "_ftn20" \o ")

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

我们现在提供了一个简化版的GovernMental，它与原始合约有一些相同的漏洞。

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.

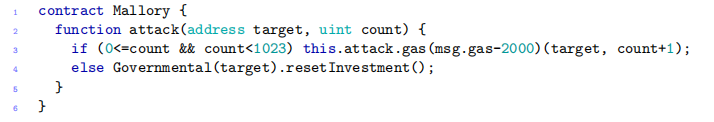
Governmental 合约将玩家的投资集中在几轮中，每轮只偿还一个赢家，即至少一分钟内最后一个赢家。要加入该计划，玩家必须至少投资一半的头奖（第14行），其金额随每次新投资而增长。任何人都可以调用resetInvestment，它将头彩（投资总额的一半）支付给赢家（第24行），并将剩余的以太发送给合约所有者。合约假设玩家要么是用户，要么是具有空回退的合约，因此在发送过程中不会发生gas不足的异常。

We now show three different attacks to our simplified GovernMental.[[21]](https://fanyi.baidu.com/" \l "_ftn21" \o ")

我们现在展示三种不同的攻击。[[21]](https://fanyi.baidu.com/" \l "_ftn21" \o ")

Attack #1.This attack exploits the vulnerabilities “exception disorder” and “stack size limit”, and is performed by the contract owner. 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:[[22]](https://fanyi.baidu.com/" \l "_ftn22" \o ")

攻击#1。此攻击利用“异常无序”和“堆栈大小限制”漏洞，由合约所有者执行。他的目标是不支付赢家，所以以太是保留合约，并由业主在以后的时间赎回。为了实现这个目标，所有者必须使第24行的发送失败。他的第一步是公布以下合约：[[22]](https://fanyi.baidu.com/" \l "_ftn22" \o ")

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.

然后，所有者调用Mallory的攻击，它开始递归地调用自己，使堆栈增长。当调用堆栈达到1022的深度时，Mallory调用Governmental的resetInvestment，然后以1023的堆栈大小执行。此时，由于调用堆栈限制，第24行的发送失败（第二次发送也失败）。由于政府没有检查send的返回代码，执行继续，重置合约状态（第27-29行），并开始另一轮。每次攻击都会增加合约的余额，因为合法的胜利者是没有报酬的。为了收集乙以太币，所有者只需要等待另一轮正确终止。

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.

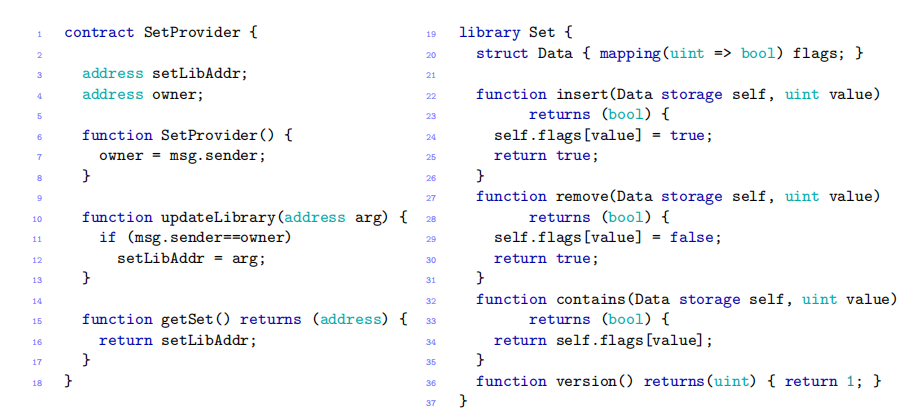
攻击#2。在这种情况下，攻击者是一名矿工，他还模仿一名玩家。作为一名矿工，他可以选择不将GovernMental的交易包括在区块中，但他自己的交易除外，以便成为本轮的最后一名玩家。此外，攻击者可以对交易进行重新排序，这样他的交易将首先出现：实际上，通过先玩并选择适当数量的以太币进行投资，他可以阻止其他玩家加入该计划（第14行），从而导致最后一个玩家在回合中。此攻击利用了“不可预测状态”漏洞，因为玩家无法确定，当他们发布一个事务以加入方案时，所投入的资金是否足以使此操作成功。

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.

攻击#3。在这种情况下，攻击者也是一个冒充玩家的矿工。假设攻击者成功加入该方案。要成为一轮中最后一名选手一分钟，他可以使用区块时间戳来玩。更具体地说，攻击者设置新块的时间戳，使其至少比当前块的时间戳晚一分钟。正如与“时间限制”漏洞一起讨论的，对时间戳的选择有一个容忍度。如果攻击者成功发布带有延迟时间戳的新区块，他将是该回合的最后一名玩家，并将赢得头奖。

## 4.6 Dynamic libraries

## 4.6动态库

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.

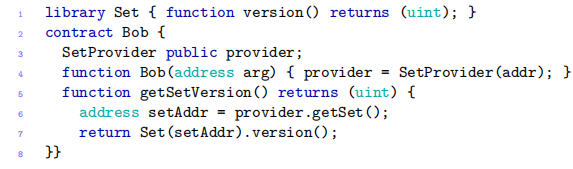
我们现在考虑一个合约，它可以动态更新它的一个组件，这是一个集合操作库。因此，如果开发了这些操作的更有效的实现，或者修复了一个bug，那么合约可以使用新版本的库。

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.

SetProvider合约的所有者可以使用函数updateLibrary用新的库地址替换库地址。任何用户都可以通过getSet获取库的地址。库集合实现一些基本的集合操作。库是特殊合约，例如不能有可变字段。当用户声明接口是库时，通过delegatecall直接调用其任何函数。标记为storage 的参数通过引用传递。

Assume that Bob is the contract of an honest user of SetProvider. In particular, Bob queries for the library version via getSetVersion:[[23]](https://fanyi.baidu.com/" \l "_ftn23" \o ")

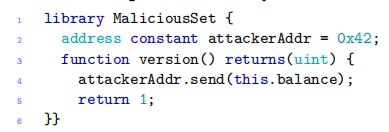
假设Bob是SetProvider合约的一个诚实用户。特别是，Bob通过getSetVersion查询库版本：[[23]](https://fanyi.baidu.com/" \l "_ftn23" \o ")

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.

现在，假设setProvider的所有者也是敌人。他可以攻击鲍勃如下，目的是偷走他所有的以太币。在攻击的第一步，对手发布一个新的库MaliciousSet，然后调用SetProvider的updateLibrary函数使其指向MaliciousSet。

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.

注意，MaliciousSet在第4行执行一个send，将以太传输给攻击者。由于Bob已将接口集声明为库，因此对version的任何直接调用都将作为delegatecall实现，并因此在Bob的环境中执行。因此，在send中，第4行实际上是指Bob的余额，导致send将他的所有以太币转移给攻击者。之后，函数将正确返回版本号。

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:



创建恶意库的另一种方法是使用函数selfdestruct。这是一个特殊函数，它禁用执行它的合约，并将其所有余额发送到目标地址。更具体地说，对手可以用以下内容替换MaliciousSet的第4行：

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

这将使鲍勃的合约永远失效，并把他的余款送给对手。

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.

上面概述的攻击利用了“不可预测状态”漏洞，因为Bob无法知道在使用SetProvider时将执行哪个版本的库。一般来说，库的主要问题是合约发布后更新的零件的存在。这使得攻击者可以用恶意的部分来更改这些部分。

# 5 Discussion

# 5讨论

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 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.

我们对以太坊智能合约的安全性进行了分析。我们的分析是基于不断增长的关于这个主题的学术文献、参加有关以太坊的互联网博客和论坛的情况，以及我们在编写智能合约方面的实践经验。据我们所知，我们的分析涵盖了迄今为止报告的所有主要漏洞和攻击。我们的分类法扩展到智能合约领域，包括软件安全漏洞的其他分类[27、28、42、50]。我们预计，随着新的漏洞和攻击的发现，我们的分类法将不断发展。

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.

可以预见，对安全敏感的区块链应用的巨大投资与其当前实现的低安全性之间的相互作用将促进对这些主题的研究。本文讨论的攻击强调，智能合约不安全的一个常见原因是难以检测到其预期行为与实际行为之间的不匹配。尽管分析和验证工具（如下面讨论的工具）可能有助于这一方向，但选择使用图灵完备语言限制了验证的可能性。我们希望非图灵完整的、人类可读的语言能够克服这个问题，至少在某些特定的应用领域是这样。最近实验语言的激增[31,33,36,38,51]表明这是一个新兴的研究方向。

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

**智能合约的验证**。最近的一些工作提出了通过合约代码的静态分析来检测漏洞的工具。

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”.

工具Oyente[43]从合约的EVM字节码中提取控制流图，并象征性地执行它以检测一些漏洞模式。特别是，该工具考虑了导致“异常无序”（例如，不检查call、send和delegatecall的返回代码）、“时间限制”（例如，在条件表达式中使用块时间戳）、“不可预测状态”和“重入性”等漏洞的模式。

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 Fcode. 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.

1. 中介绍的工具将智能合约（Solidity或EVM字节码）翻译成函数语言F[53]。然后在函数性语言F上验证各种属性。特别是，通过寻找特定的模式，可以针对“异常混淆”和“可重入性”漏洞检查从Solidity合约获得的代码。从EVM获得的代码支持低级分析，例如计算合约函数的gas量界限。此外，给定一个Solidity程序并将其编译成EVM字节码，该工具验证这两段代码是否具有相同的行为。

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 ∼ 28% of the analyzed contracts potentially contain “exception disorder” vulnerabilities.

这两种工具都在以太坊区块链上发布的合约上进行了试验。这一大规模分析的结果表明，安全漏洞普遍存在。例如，[43]报告说∼28%的合约可能包含“异常无序”漏洞。

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.

这项工作[41]使用Isabelle/HOL证明[47]来验证特定的合约。更准确地说，分析的目标是通过编译作为以太坊名称服务一部分的合约，“合约”的实体代码变为EVM字节码。通过Isabelle/HOL证明的定理指出，在调用合约时，只有其所有者才能减少余额。

**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.

**低级攻击**。除了涉及合约的攻击外，以太坊网络也成为攻击者的攻击目标。他们的攻击利用EVM级别的漏洞，结合以太坊客户端的安全缺陷。

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].

例如，最近的一次拒绝服务攻击利用了一条EVM指令，与执行该指令所需的计算工作量相比，该指令的gas单位成本太低[6]。攻击者用该指令淹没网络，导致其计算能力大幅下降，并减慢区块链同步过程。与从DAO攻击中恢复类似，这个问题也通过分叉区块链得到了解决[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.

客户端实现中的漏洞也可能是攻击的原因。最近的一份技术报告[57]分析了以太坊官方客户端。通过利用块传播算法，他们发现以太坊网络可以被分成几个节点组：这样，节点就可以被迫接受攻击者临时创建的块序列。

**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.

**致谢**。作者热烈感谢以太坊基金会的克里斯蒂安·雷特维纳和苏黎世以太坊基金会的亚瑟·热尔维斯对本文初稿的评论。所有在这部作品中表达的观点都是作者的观点。

# References

1.       Announcement of imminent hard fork for EIP150 gas cost changes, https://blog.ethereum.org/2016/10/13/announcement-imminent-hardfork-eip150-gas-cost-changes/

2.       Bitcointalk: Hi!My name is Rubixi, https://bitcointalk.org/index.php?topic=

[1400536.60](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-thanmillion-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-undergoingdos-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-ethereumcontracts-for-bugs/

15.    Hard fork completed, https://blog.ethereum.org/2016/07/20/hard-forkcompleted/

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-daohack-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., Wu¨st, 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 higherorder 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., B´eguelin, 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.    Wu¨st, K., Gervais, A.: Ethereum Eclipse Attacks. Tech. rep., ETH-Zu¨rich (2016)