Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab

Kevin Delmolino
del@terpmail.umd.edu

Mitchell Arnett
marnett@umd.edu

Ahmed Kosba
akosba@cs.umd.edu

Andrew Miller
amiller@cs.umd.edu

Elaine Shi
runting@gmail.com

1 Introduction

Completely decentralized cryptocurrencies like Bitcoin [10] and other altcoins [2] have captured the public's attention and interest, and have been much more successful than any prior incarnations of electronic cash. Many would call the rise of these electronic currencies a technological revolution, and the "wave of the future" [1]. Emerging altcoins such as Ethereum [13] and Counterparty [8] extend Bitcoin's design by offering a rich programming language for writing "smart contracts." [11] Smart contracts are user-defined programs that specify rules governing transactions, and that are enforced by the network of peers (assuming the underlying cryptocurrency is secure). In comparison with traditional financial contracts, smart contracts carry the promise of low legal and transaction costs, and can lower the bar of entry for users.

In Fall 2014, at the University of Maryland, we organized a new, hands-on smart contract programming lab in our undergraduate-level "CMSC 414: Computer and Network Security" class – the first of its kind that has ever been attempted.

Smart contract programming: unique challenges. Although smart contract programming in many ways resembles traditional programming, it raises important new security challenges. Contracts are "play-for-keeps", since virtual currencies have real value. If you load money into a buggy smart contract, you will likely lose it. Further, smart contract programming requires an "economic thinking" perspective that traditional programmers may not have acquired. Contracts must be written to ensure fairness even when counterparties may be incentivized to cheat in arbitrary ways to maximize their economic gains.

As an outcome of our lab, we observed several classes of typical mistakes students made. In contrast to traditional software development where bugs such as buffer overflows are typical, in our lab, we observed bugs and pitfalls that arise due to the unique nature of smart contract programs.

Our lab experiences show that even for very simple smart contracts (e.g., a "Rock, Paper, Scissors" game), designing and implementing them correctly was highly non-trivial. This suggests that extra precautions and scrutiny are necessary when programming smart contracts.

Open-source course and lab materials. Based on lessons and insights drawn through this experimental lab, we have designed new, open course materials and lab designs for smart contract programming [4]. We hope that these open-source course materials and labs will aid both instructors who wish to teach smart contract programming and students/developers who wish to teach themselves smart contract programming.

Broader insights gained. Inspired by our experimental smart contract lab, we argue why cryptocurrency and smart contracts will serve as a great pedagogical platform for Cybersecurity education. We also draw from our experiences why the "build, break, and amend your own program" approach is beneficial to instructing adversarial thinking and incentivizing a student-driven learning atmosphere.

Roadmap. In the remainder of this paper, we will first give more background on cryptocurrency and smart contracts (Section 2). We will then detail experiences with our lab (Section 3), the typical pitfalls we observed in smart contract programming (Section 4), and the insights and lessons learned.
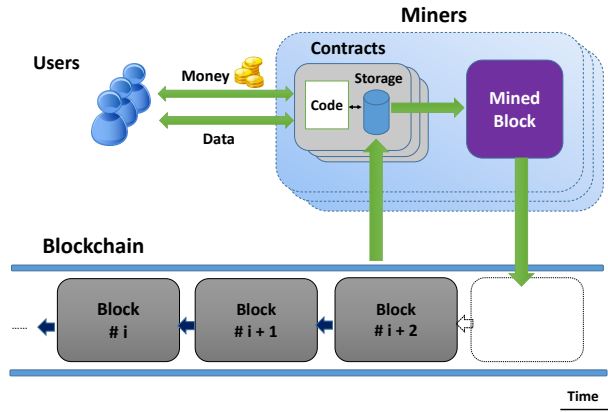
Figure 1: **Schematic of a decentralized cryptocurrency system with smart contracts.** A smart contract's state is stored on the public blockchain. A smart contract program is executed by a network of miners who reach consensus on the outcome of the execution, and update the contract's state on the blockchain accordingly. Users can send money or data to a contract; or receive money or data from a contract.

2 Background

In this section, we provide some background on cryptocurrencies and the programming model of smart contracts.

2.1 Background on Decentralized Cryptocurrencies

Smart contracts are built on top of an underlying cryptocurrency platform. A cryptocurrency is a decentralized system for interacting with virtual money in a shared global ledger. Users transfer money and interact with contracts by publishing signed messages called *transactions* to the cryptocurrency network. The network consists of nodes (called miners) who propagate information, store data, and update the data by applying transactions. A high-level schematic is shown in Figure 1.

Although many of the ideas behind cryptocurrencies date back around twenty years (e.g., cryptographic e-cash [7] and smart contracts [11]), a recent surge of interest in this technology has been incited by the success of Bitcoin [9]. For a comprehensive survey on Bitcoin and other cryptocurrencies, see [6, 5].

The main interface provided by the underlying cryptocurrency is an append-only log called a *blockchain*, which imposes a total ordering on transactions submitted by users. The data in the blockchain is guaranteed to be *valid* according to certain predefined rules of the system (e.g., there are no double-spends or invalid signatures). All of the data in the blockchain is public, and every user can access a copy of it. No one can be *prevented* from submitting transactions and getting them included in the blockchain (with at most some small delay). There is global agreement among all nodes and users about the contents of the blockchain, except for the most recent handful of blocks which have not yet settled.

We also assume that the built-in currency has a stable monetary value. Users have an incentive to gain more of (and avoid losing) units of this currency. Anyone can acquire the virtual currency by purchasing or trading for it using other fiat currencies (e.g., US dollars) or virtual currencies. The currency is assumed to be fungible; one unit of ether is exactly as valuable as any other, regardless of the currency's "history."

The system keeps track of "ownership" of the currency by associating each unit of currency to an "address". A user address is a hash of a public key; whoever knows the corresponding private key can spend the money associated to that address. Users can create as many accounts as they want, and the accounts need not be linked to their real identity.

2.2 Background on Smart Contracts

A contract is an instance of a computer program that runs on the blockchain. It consists of program code, a storage file, and an account balance. Any user can create a contract by posting a *transaction* to the blockchain. The program code of a contract is fixed when the contract is created, and cannot be changed.

As shown in Figure 1, a contract's storage file is stored on the public blockchain. A contract's program logic is executed by the network of miners who reach consensus on the outcome of the execution and update the blockchain accordingly. The contract's code is executed whenever it receives a message, either from a user or from another contract. While executing its code, the contract may read from or write to its storage file. A contract can also receive money into its account balance, and send money from its account balance to other contracts or users.

Conceptually, one can think of a contract as a special "trusted third party" – however, this party is *trusted only for correctness but not for privacy*. In particular, a contract's entire state is visible to the public.

Contract invocation. A contract's code will be invoked whenever it receives a transaction from a user. A contract can define multiple entry points of exe-

2

cution – in Ethereum's Serpent language, each entry point is defined as a function. A transaction's contents will specify the entry point at which the contract's code will be invoked. Therefore, transactions act like function calls in ordinary programming languages. After a contract finishes processing a message it receives, it can pass a return value back to the sender.

Gas. Ethereum uses the concept of "gas" to discourage over-consumption of resources (e.g., a contract program that causes miners to loop forever). The user who creates a transaction must spend currency to purchase gas. During the execution of a transaction, every program instruction consumes some amount of gas. If the gas runs out before the transaction reaches an ordinary stopping point, it is treated as an exception: the state is reverted as though the transaction had no effect, but the ether used to purchase the gas is not refunded! When one contract sends a message to another, the sender can offer only a *portion* of its available gas to the recipient. If the recipient runs out of gas, control returns to the sender, who can use its remaining gas to handle the exception and tidy up.

Ethereum specifics. Our lab employs Ethereum's Serpent language to illustrate smart contract programming, although the lessons are intended to apply more generally to other cryptocurrencies and smart contract systems as well. Therefore we only define as much Ethereum-specific terminology as needed to understand our examples. In particular, the built-in virtual currency of Ethereum is called Ether, and units of Ether are called "wei."

2.3 A Taste of Smart Contract Design

In this section, we will give the reader a brief overview of smart contract design by describing the Ethereum implementation of a simple, yet useful, motivating example – the financial swap instrument. This contract allows two parties, Alice and Bob, to take opposing bets about the price of a stock at some future time. Both parties initially deposit equal amounts of money (as units of Ether currency). After a deadline has passed, the current price of the stock is queried by interacting with a designated stock price authority (which would itself be implemented as a smart contract - we refer to this contract as `StockPriceAuthority`). Depending on the price at that time, the entire combined deposit is awarded to either Alice or Bob.

```
1   data Alice, Bob
2   data deadline, threshold
3
4   # Not shown: collect equal deposits from
     ↪   Alice and Bob
5   # We assume StockPriceAuthority is a
     ↪   trusted third party contract that
     ↪   can give us the price of the stock
6
7   def determine_outcome():
8     if block.timestamp > deadline:
9       price = StockPriceAuthority.price()
10      if price > threshold:
11        send(Alice, self.balance)
12      else:
13        send(Bob, self.balance)
```

Figure 2: This Serpent program implements a simple financial "swap" instrument, illustrating that smart contracts are a powerful and useful tool for programming with money.

The contract's storage allocates space for the following data on lines 1 and 2: 1) the public keys of Alice and Bob; and 2) the deadline and threshold of the swap contract. The contract also defines a function determine_outcome, which any party may invoke. This example serves as motivation of the "useful" aspects of smart contracts as financial instruments. In our other examples, we will tend to focus on gambling games. It also serves to illustrate several low level aspects of Serpent programming.

3 A Recount of Our Smart Contract Programming Lab

In our undergraduate security class at the University of Maryland, students were asked to develop smart contract applications of their choice atop a new cryptocurrency called Ethereum [13]. Ethereum offers a Turing complete programming language called Serpent [12] for composing smart contracts.

Students were divided into groups of four. Due to the experimental nature of the lab, the instructor assigned one of her Ph.D. students to closely supervise each group, to ensure that students could obtain hands-on help. The lab proceeded in two phases.

Creation phase. The first phase is a creation phase where each group created a smart contract application of their own choice. The students created a variety of applications, including games (e.g., Rock-Paper-Scissors, Russian Roulette, custom-designed games), escrow services, auctions (e.g., sealed auc-

tions, silent auctions), a parking meter service, and stock market applications.

At the end of the first phase, each group made a short presentation of their contract application in class. The instructor, TAs, and students jointly observed numerous issues with the programs that students created (see Section 4 for a detailed discussion).

Amendment phase. Therefore, we extended the project to a second phase, called an amendment phase. The goal of this phase was for students to critique their programs, find bugs, and amend their designs. The instructor and TAs had in-person meetings with each project group to help them amend their smart contract programs. Students also formed pair groups to critique and help the other group.

4 Pitfalls of Smart Contract Programming

In this section, we will demonstrate some of the typical pitfalls we observed for smart contract programming. For ease of exposition, we will use a simple "Rock, Paper, Scissors" example to illustrate three classes of typical mistakes. Similar mistakes were commonly observed in various other applications developed by the students.

Quick overview of our running example. We will first give a quick overview of the structure of our buggy "Rock, Paper, Scissors" program before we go on to diagnose the bugs. In this contract, two players will play a simple "Rock, Paper, Scissors" for money. The contract program consists of two main functions:

- `player_input`: The players register with the contract and deposit money to play. Each player also provides input to the contract in the form of their choice of rock, paper, or scissors.

- `finalize`: The contract decides a winner and sends the proceeds to the winner.

As we show below, surprisingly, *even for a very simple smart contract like this, it is difficult to create it correctly!*

4.1 Errors in Encoding State Machines

Programming smart contracts typically involves encoding complex state machines. Logical errors in encoding state machines were commonly observed.

The simplest type of logical error is a contract that leaks money in corner cases.

To illustrate this, let us look at our buggy "Rock, Paper, Scissors" example. Figure 3 shows the `player_input` function where players register with the contract and deposit money to play. The contract would then store the players' public keys, inputs, and coins deposited (Lines 14-17).

This contract makes several mistakes:

- If a *third* player attempts to join and sends money to the contract, that money becomes inaccessible (Line 20). Neither the player nor anyone else can ever recover it.

- Similarly, if a player sends an amount of money that is not exactly 1000 wei, the contract also leaks the money.

Note that while a careful player can protect herself from the second problem by never sending the incorrect amount, *she cannot always protect herself from the first problem!* In a decentralized cryptocurrency like Bitcoin or Ethereum, multiple parties may be sending inputs to the contract simultaneously. In this case, it is up to the miner who mines this block to decide how to order these transactions.

To fix these bugs, the contract should *refund* the money back to a player unless the player is successfully registered in the game. This approach is taken in our improved contract (Figure 4, Lines 18 and 21).

What is shown here is merely the simplest example of a logical error when encoding the state machine. In our lab, students created contracts that are far more sophisticated (e.g., stock market applications, various flavors of auctions) that required the design of much more complex state machines. Failure to encode the correct state machine (e.g., omitting certain transitions, neglecting to check the current state) was among the most commonly observed pitfalls.

4.2 Failing to use Cryptography

Another mistake is more subtle: Players send their inputs in cleartext. Since transactions are broadcast across the entire cryptocurrency network, a cheating player may wait to see what his opponent chooses before providing his own input.

Players in a smart contract are typically anonymous, and can be reasonably expected to act selfishly to maximize their financial gains, even if it means deviating from the default or "honest" behavior.

```
1   # A two-player game with a 1000 wei prize
2
3   data player[2](address, choice)
4   data num_players
5   data reward
6   data check_winner[3][3] # a ternary
    ↪   matrix that captures the rules of
    ↪   rock-paper-scissors game
7
8   def init():
9     num_players = 0
10    # code omitted: initialize check_winner
      ↪   according to the game rules
11
12  def player_input(choice):
13    if num_players < 2 and msg.value ==
      ↪   1000:
14      reward += msg.value
15      player[num_players].address =
        ↪   msg.sender
16      player[num_players].choice = choice
17      num_players = num_players + 1
18      return(0)
19    else:
20      return(-1)
21  def finalize():
22    p0 = player[0].choice
23    p1 = player[1].choice
24    # If player 0 wins
25    if check_winner[p0][p1] == 0:
26      send(0,player[0].address, reward)
27      return(0)
28    # If player 1 wins
29    elif check_winner[p0][p1] == 1:
30      send(0,player[1].address, reward)
31      return(1)
32    # If no one wins
33    else:
34      send(0,player[0].address, reward/2)
35      send(0,player[1].address, reward/2)
36      return(2)
```

Figure 3: **Pitfalls in smart contract design.** This buggy contract illustrates a few pitfalls:
Pitfall 1 (Lines 19 and 20): If a third player attempts to join the contract, his money effectively vanishes into a blackhole.
Pitfall 2 (Line 16): Players send their inputs in plaintext to the contract. A malicious player can wait to see his opponents choice before deciding on his own input.

Cryptography is often the first line of defense against potentially malicious parties. Here, the obvious remedy is to use cryptographic commitments. Both players can commit to their inputs in one time epoch, and then in a later epoch open the commitments and reveal their inputs. A commitment satisfies two properties, *binding* and *hiding*. Binding ensures that a player cannot change their input after committing to it. Hiding ensures that a party learns nothing about the others input choice even after observing the commitment.

In Figure 4, we show a fixed contract that properly uses commitments. The previous player_input function is broken up into two phases: in the new player_input function, each player provides a commitment; after both commitments are received, the open function is used to reveal their commited inputs.

Opportunity to teach cryptography. When students were given the opportunity to realize and fix bugs in their own programs, an opportunity arose to teach them cryptography. Given the chance, the instructor grasped this opportunity to teach students cryptographic commitments. In the amendment phase of the project, students were able to implement cryptographic commitments to secure their smart contracts!

4.3 Misaligned Incentives

More subtle bugs remain, even for the improved contract in Figure 4.

For example, one party can wait for the other to open its commitment. Upon seeing that he will lose, that party may elect to abort – thus denying payment to the other player as well. It may seem at first glance like the losing party should be indifferent to revealing his committed input or not (regardless, we would prefer to have a clear positive preference for revealing it); however, the reality is slightly worse, since that party must incur a *gas* cost to even submit transaction that opens his commitment.

This generalizes to a broader question of how to ensure the incentive compatibility of a contract. Can any player profit by deviating from the intended behavior? Does the intended behavior have hidden costs?

In this specific example, we can remedy the problem by having both players include an additional security deposit in the first stage, which they forfeit unless they open their commitments in a timely manner. This way, even the losing player has a motivation to open his bid.

```
1   data player[2](address, commit, choice,
    ↪  has_revealed)
2   data num_players
3   data reward
4   data check_winner[3][3]
5   data timer_start
6
7   def init():
8     num_players = 0
9     # code omitted: initialize check_winner
      ↪   according to the game rules
10
11  def player_input(commitment):
12    if num_players < 2 and msg.value >=
      ↪  1000:
13      reward += msg.value
14      player[num_players].address =
        ↪   msg.sender
15      player[num_players].commit =
        ↪   commitment
16      num_players = num_players + 1
17      if msg.value - 1000 > 0:
18        send(msg.sender, msg.value-1000)
19      return(0)
20    else:
21      send(msg.sender, msg.value)
22      return(-1)
23
24  def open(choice, nonce):
25    if not num_players == 2: return(-1)
26    # Determine which player is opening
27    if msg.sender == player[0].address:
28      player_num = 0
29    elif msg.sender == player[1].address:
30      player_num = 1
31    else:
32      return(-1)
33    # Check the commitment is not yet
      ↪   opened
34    if sha3([msg.sender, choice, nonce],
      ↪   items=3) ==
      ↪   player[player_num].commit and not
      ↪   player[player_num].has_revealed:
35      # Store opened value in plaintext
36      player[player_num].choice = choice
37      player[player_num].has_revealed = 1
38      # Wait 10 blocks for second player to
        ↪   open
39      if not timer_start:
40        timer_start = block.number
41      return(0)
42    else:
43      return(-1)
```

```
44  def finalize():
45    # Wait 10 blocks for both players to
      ↪   open
46    if block.number - timer_start < 10:
47      return(-2)
48    #check to see if both players have
      ↪   revealed answer
49    if player[0].has_revealed and
      ↪   player[1].has_revealed:
50      p0 = player[0].choice
51      p1 = player[1].choice
52      #If player 0 wins
53      if check_winner[p0][p1] == 0:
54        send(player[0].address, reward)
55        return(0)
56      #If player 1 wins
57      elif check_winner[p0][p1] == 1:
58        send(player[1].address, reward)
59        return(1)
60      #If no one wins
61      else:
62        send(player[0].address, reward/2)
63        send(player[1].address, reward/2)
64        return(2)
65    # If p1 opens but not p2, send money to
      ↪   p1
66    elif player[0].has_revealed and not
      ↪   player[1].has_revealed:
67      send(player[0].address, reward)
68      return(0)
69    # If p2 opens but not p1, send money to
      ↪   p2
70    elif not player[0].has_revealed and
      ↪   player[1].has_revealed:
71      send(player[1].address, reward)
72      return(1)
73    # If neither opens, forfeit both bets
74    else:
75      return(-1)
```

Figure 4: **An improved but nonetheless buggy contract.** When an edge case occurs, the contract refunds the players rather than leaking money (Lines 18 and 21). A cryptographic commitment scheme is used to offer privacy of users' inputs before they are revealed for the winner decision (Line 15 and 36). As mentioned in Section 4.3, this improved contract is still not safe due to misaligned incentives.

4.4 Ethereum-specific Mistakes

Several subtle details about Ethereum's implementation make smart contract programming prone to error. Without going into too much detail, contracts must be written "defensively" to avoid exceptions that can occur when multiple contracts interact. One Ethereum contract can send a message to another contract, which can in turn send a message to another. However, Ethereum limits the resulting call-stack to a fixed size of 1024. For example, if the call-stack depth is already at this limit when the `send` instruction on Line 62 of Figure 4 is reached, then that instruction will be skipped and the winner will not get paid. Furthermore, a `send` instruction sends by default the maximum available gas to the recipient. If the recipient of the `send` instruction on Line 62, for example, is a contract with buggy code that raises an exception, then Line 63 is never executed and the other player loses out. In our online course materials [4] we offer guidance on avoiding these Ethereum-specific hazards.

4.5 Complete, Fixed Contract

Due to space constraints, we provide a fully working, incentive compatible, and secure contract for the "Rock, Paper, Scissors" game in our online course materials [4].

5 Conclusion

5.1 Open-Source Course and Lab Materials

Our smart contract programming lab was an audacious, original attempt at instructing a technology of in-development nature. Ethereum and its Serpent language have only recently emerged, and are rapidly undergoing changes. The Serpent language is not well documented and development environment support (e.g., debugging tools) is also rudimentary. Therefore, several students struggled in installing the simulation environment and getting up to speed.

To facilitate future pedagogical endeavors on smart contract programming, and avoid issues resulting from the in-development nature of the technology, we will shortly open source well-structured course and lab materials on smart contract programming. For a sneak peek of our course materials, please visit `http://mc2-umd.github.io/ethereumlab/` [4].

The course materials comprise the following:

- A detailed language reference guide for Ethereum's Serpent language − a smart contract language that we adopted in the lab.

- A virtual machine image with a snapshot of `pyethereum` and `serpent` installed, providing a simulator environment for experimentation. Since the Ethereum's Serpent language is constantly under development, our Serpent language reference matches with the snapshot installed in this VM.

- A tutorial that builds on our "Rock, Paper, Scissors" example, intended to walk the student through the typical pitfalls in programming safe smart contracts. The student is presented with the buggy version of the contract and asked to fix the bugs in a step-by-step, guided manner.

5.2 Cryptocurrency and Smart Contracts as a Cybersecurity Pedagogical Platform

Our experiences also led us to conclude that cryptocurrency and smart contracts are a great platform for cybersecurity pedagogy. First, cryptocurrency and smart contracts, like other cool emerging technologies, could easily capture the students' attention and imagination. Second, cybersecurity is a science that is interdisciplinary in nature; and cryptocurrency is a platform that captures multiple core cybersecurity notions, e.g., cryptography, programming languages, and incentives. Third, cryptocurrency and smart contracts easily motivate "adversarial thinking". For example, in our lab, students had to analyze their own smart contracts and reason how other selfish players can harm honest participants and maximize their own financial gains.

5.3 The "Build, Break, and Amend Your Own Programs" Approach to Cybersecurity Education

Inspired by our smart contract programming lab, we also feel that the "Build, break, and amend your own programs" approach is very helpful for cybersecurity education.

In our labs, students learned why security is difficult and learned adversarial thinking by analyzing and breaking their own programs. Students initially failed to make proper use of cryptography in their smart contracts (see Section 4). But then, by realizing why their smart contracts are not safe, they become self-driven in learning cryptographic building blocks.

In future work, we plan to further extend these pedagogical ideas, such that students can learn

through hands-on, creative experiences, and learn adversarial thinking through attacking and amending their own code.

5.4 Inspired Research

This lab has inspired our current research program on cryptocurrencies and smart contracts. Several exciting new results are forthcoming.

Crime version of this "step by step" paper. [Blinded et. al.] [3] recently demonstrate how smart contracts can be leveraged to facilitate criminal activities and create incentive compatible underground eco-systems. They then discuss countermeasures and advocate the responsible deployment of technology. Their paper would be the criminal counterpart of our "step by step" paper.

References

[1] The rise and rise of bitcoin. Documentary, http://bitcoindoc.com/.

[2] ShaikShakeel Ahamad and Madhusoodhnan Nair2and Biju Varghese. A survey on crypto currencies. In *Int. Conf. on Advances in Civil Engineering, AETACE*, 2013.

[3] Anonymous. Rings of gyges: Using smart contracts for crime. Manuscript, 2015.

[4] Mitchell Arnett, Kevin Delmolino, Ahmed Kosba, Andrew Miller, and Elaine Shi. Ethereumlab: Towards safe smart contracts. http://mc2-umd.github.io/ethereumlab/.

[5] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to betterhow to make bitcoin a better currency. In *Financial cryptography and data security*, pages 399–414. Springer, 2012.

[6] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies (Extended Version). Cryptology ePrint Archive, Report 2015/452, 2015.

[7] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In *Proceedings on Advances in cryptology*, pages 319–327. Springer-Verlag New York, Inc., 1990.

[8] Adam Krellenstein Robby Dermody and Ouziel Slama. Counterparty announcement. https://bitcointalk.org/index.php?topic=395761.0, January 2014.

[9] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[10] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf, 2009.

[11] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[12] Etheruem Wiki. Serpent. https://github.com/ethereum/wiki/wiki/Serpent, 2015.

[13] Gavin Wood. Ethereum: A secure decentralized transaction ledger. http://gavwood.com/paper.pdf, 2014.