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

Kevin Delmolino[1(✉)], Mitchell Arnett[1], Ahmed Kosba[1], Andrew Miller[1,2], and Elaine Shi[2]

[1] Department of Computer Science, University of Maryland, College Park, College Park, USA
del@terpmail.umd.edu, marnett@umd.edu, {akosba,amiller}@cs.umd.edu
[2] Initiative for Cryptocurrencies and Contracts (IC3), Department of Computer Science, Cornell University, Ithaca, USA
elaine@cs.cornell.edu

Abstract. We document our experiences in teaching smart contract programming to undergraduate students at the University of Maryland, the first pedagogical attempt of its kind. Since smart contracts deal directly with the movement of valuable currency units between contractual parties, security of a contract program is of paramount importance.

Our lab exposed numerous common pitfalls in designing safe and secure smart contracts. We document several typical classes of mistakes students made, suggest ways to fix/avoid them, and advocate best practices for programming smart contracts. Finally, our pedagogical efforts have also resulted in online open course materials for programming smart contracts, which may be of independent interest to the community.

1 Introduction

Completely decentralized cryptocurrencies like Bitcoin [18] and other altcoins [5] 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" [3]. Emerging altcoins such as Ethereum [23] and Counterparty [14] extend Bitcoin's design by offering a rich programming language for writing "smart contracts." Smart contracts are user-defined programs that specify rules governing transactions, and that are enforced by a 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 security class – the first of its kind that has ever been attempted.

© International Financial Cryptography Association 2016
J. Clark et al. (Eds.): FC 2016 Workshops, LNCS 9604, pp. 79–94, 2016.
DOI: 10.1007/978-3-662-53357-4_6

Smart Contract Programming: Unique Challenges and Opportunities. 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 attempt to cheat in arbitrary ways that 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 tasks where bugs such as buffer overflows are often benign (except in rare or contrived scenarios), in our lab, we observed several bugs and pitfalls that arise due to the unique nature of smart contract programs and lead to clear and immediate exploits (e.g., theft or loss of money).

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.

In this paper, although we adopt Ethereum's Serpent language, most of the the insights we gain are not language-specific, but can be generalized to smart contract programming under a broad model.

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 (Sect. 2). We will then detail experiences with our lab (Sect. 3), the typical pitfalls we observed in smart contract programming (Sect. 4), and the insights and lessons learned.

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 data 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 Fig. 1.

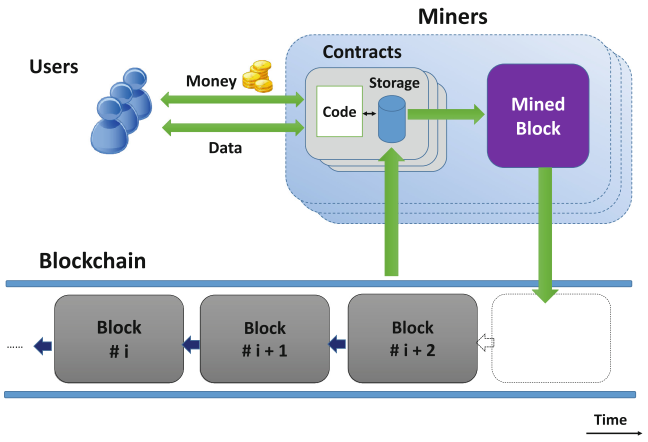

Fig. 1. Schematic of a decentralized cryptocurrency system with smart contracts. A smart contract's state is stored on the public blockchain. A smart contract program is executed by a network of miners who reach consensus on the outcome of the execution, and update the contract's state on the blockchain accordingly. Users can send money or data to a contract; or receive money or data from a contract.

Although the ideas behind cryptocurrencies date back at least twenty-five years (e.g., cryptographic e-cash [13]), a recent surge of interest in this technology has been incited by the success of Bitcoin [18]. For a comprehensive survey on Bitcoin and other cryptocurrencies, see [7,9].

The main interface provided by the underlying cryptocurrency is an append-only log called a *blockchain*, which imposes a partial or 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.

For simplicity, we also assume that the built-in currency has a stable monetary value. Users therefore 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 (the currency unit of Ethereum) 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

Need for General Smart Contracts. Bitcoin offers a rudimentary scripting system that is neither expressive nor user-friendly. A line of work in both academia and industry has attempted to design various smart contract applications in a way that retrofits Bitcoin's scripting language [6,8,19,20]. Due to fundamental limits of the expressiveness of Bitcoin's scripting language, retrofitting the language is not only time consuming, but can also result in asymptotically more costly implementations in terms of number of rounds or on-chain cost. In comparison, many of the same tasks would be easier to program and more efficient, if they were built atop a general purpose smart contract language (of which Ethereum [23] is the first incarnation).

Smart Contract Model. A contract is an instance of a computer program that runs on the blockchain, i.e., executed by all consensus nodes. A smart contract 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 Fig. 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. (A user can sends a message to a contract by including the message data and the address of the contract in her transaction; one contract can send a message to another using a special instruction in its program code.) 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 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 and availability 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 message. A contract can define multiple entry points of execution – in Ethereum's

Serpent language, each entry point is defined as a function. A message contents will specify the entry point at which the contract's code will be invoked. Therefore, messages 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 apply more generally to other cryptocurrencies and smart contract systems as well. We only define as much Ethereum-specific terminology as needed to understand our examples. In particular, the built-in currency of Ethereum is called Ether, and an Ether can be divided into smaller currency units such as "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.

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 Ethereum [23], using its expressive Serpent [22] programming language for composing smart contracts (Fig. 2).

```
data Alice, Bob
data deadline, threshold

# Not shown: collect equal deposits from Alice and Bob
# We assume StockPriceAuthority is a trusted third party contract that can give us the price of the stock

def determine_outcome():
  if block.timestamp > deadline:
    price = StockPriceAuthority.price()
    if price > threshold:
      send(Alice, self.balance)
    else:
      send(Bob, self.balance)
```

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

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 auctions, 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 Sect. 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 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 exhibits 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 (Fig. 4, Lines 17 and 20).

```
 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 += 1000
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)
```

Fig. 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.

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.

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 standard 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 our application, the commitment must also be *non-malleable*, i.e., a player should not be able to maul a previous player's commitment into a related value (e.g., one that will allow her to win). In general, for secure composition of computationally sound primitives, we would recommend the usage of universally composable commitments [10–12]. In this paper, we will use a simple, hash-based commitment that is secure under the random oracle model. To commit a message m, first pick a random `nonce` that is sufficiently long, and then compute the commitment $H(m, \texttt{nonce})$. The opening and verification algorithms are obvious.

In Fig. 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. As students realized and sought to fix bugs in their own programs, the opportunity arose to teach them cryptography as a *well-motivated solution to their immediate practical problems*. The instructor seized this opportunity, and the amendment phase of the project, students were indeed able to implement cryptographic commitments to secure their smart contracts!

4.3 Misaligned Incentives

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

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

Fig. 4. An improved but nonetheless buggy contract. This contract fixes a subset of the problems identified in the original (Fig. 3). When an edge case occurs, the contract refunds the players rather than leaking money (Lines 17 and 20). A cryptographic commitment scheme is used to offer privacy of users' inputs before they are revealed for the winner determination (Line 14 and 33–36). As mentioned in Sect. 4.3, this improved contract is still not safe due to misaligned incentives.

```
def finalize():
  #check to see if both players have revealed answer
  if player[0].has_revealed and player[1].has_revealed:
    p0 = player[0].choice
    p1 = player[1].choice
    #If player 0 wins
    if check_winner[p0][p1] == 0:
      send(player[0].address, reward)
      return(0)
    #If player 1 wins
    elif check_winner[p0][p1] == 1:
      send(player[1].address, reward)
      return(1)
    #If no one wins
    else:
      send(player[0].address, reward/2)
      send(player[1].address, reward/2)
      return(2)
  else:
    return(-1)
```

Fig. 4. (*continued*)

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 (i.e., not to send any further messages) – 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 a 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 setting a deadline before which the second player has to reveal, otherwise the player who revealed first will be able to get the reward. This will protect the first player when the second player aborts. The modifications needed to protect against that case are shown in Fig. 5. Furthermore, we can have 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 stronger motivation to open his bid, but this change is not included here for simplicity.

```
# Declare a timer variable in the beginning
data timer_start
#### < Code omitted. Same as Figure 4 lines (1-22) >

def open(choice, nonce):
  #### < Code omitted. Same as Figure 4 lines (24-32)>
  if sha3([msg.sender, choice, nonce], items=3) ==
  ↪  player[player_num].commit and not
  ↪  player[player_num].has_revealed:
    player[player_num].choice = choice
    player[player_num].has_revealed = 1
    # Keep track of the first reveal time. The other player should
    ↪  reveal before 10 blocks are mined.
    if not timer_start:
      timer_start = block.number
    return(0)
  else:
    return(-1)

def finalize():
  # Check timer: Wait 10 blocks for both players to open
  if block.number - timer_start < 10:
    return(-2)

  if player[0].has_revealed and player[1].has_revealed:
    #### < Code omitted. Same as Figure 4 lines (47-61)>
  # Check for abort: If p1 opens but not p2, send money to p1
  elif player[0].has_revealed and not player[1].has_revealed:
    send(player[0].address, reward)
    return(0)
  # If p2 opens but not p1, send money to p2
  elif not player[0].has_revealed and player[1].has_revealed:
    send(player[1].address, reward)
    return(1)
  else:
    return(-1)
```

Fig. 5. Modifications required to the contract of Fig. 4 to protect against an aborting player.

4.4 Ethereum-Specific Mistakes

Several subtle details about Ethereum's implementation make smart contract programming prone to error.

Call-Stack Bug. 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 callstack depth is already at this limit when the `send` instruction on Line 59 of Fig. 4 is reached, then that instruction will be skipped and the player 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 59, for example, is a contract with buggy code that raises an exception, then Line 60 is never executed and the other player loses out. We stress that the same bug was later manifested in Etherpot [2], a lottery application built atop Ethereum and released to the public. In our online course materials [4] we offer guidance on how to avoid this call-stack bug in Ethereum.

Blockhash Bug. Another Ethereum-specific quirk is that the `block.prevhash` instruction supports only the 256 most recent blocks, presumably for efficiency reasons. This limitation also affected Etherpot [2] and potentially other contracts that went into production. Miller proposed one potential fix to this problem by implementing a global "blockhash service" contract that allows other contracts to retrieve block hashes beyond 256 blocks [1].

Incentive Bugs. Ethereum's underlying mining protocol can introduce subtle, incentive-related bugs. We again use Etherpot [2] as an example. Etherpot uses the hash of a block in the blockchain (e.g., at height T) as a random beacon value to pick the lottery winner. However, by selectively withholding blocks, miners can bias this value, gaining an unfair advantage in the lottery - the miner who first finds a block at height T can check whether this results in them winning the lottery – if not, they can withhold the block until another block is found, gaining a "second chance" to win. To combat this, Etherpot makes sure the prize value of each lottery is less than the base block reward. Thus a miner who withholds a block must sacrifice the block reward they would have earned. However, Ethereum implements a protocol variation called GHOST [17,21], which allows miners who temporarily withhold blocks to still get a (discounted) reward for their block, even if the block is revealed later. Thus Etherpot's reward limit is set too loose.

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, we have released open course materials on smart contract programming [4]. The course materials comprise the following:

– A detailed language reference guide for Ethereum's Serpent programming language.
– A virtual machine image with a snapshot of `pyethereum` and `serpent` compiler installed, providing a simulator environment for experimentation. Since the Ethereum's Serpent language is constantly under development, our Serpent language reference matches 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.

These materials are available at https://mc2-umd.github.io/ethereumlab/.

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 interesting 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 game theory. 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 Sect. 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 Subsequent Pedagogical Efforts and Research

Based on insights gained through our experiences, one of the co-authors of this paper, Miller, gave a smart contract programming tutorial at 1st Cyberport FinTech Programming Workshop. This lab has also inspired later research on cryptocurrencies and smart contracts. Juels et al. [15] recently demonstrated 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. Finally, Kosba et al. propose a general formal model for the "blockchain model of computation" which captures the formal abstraction of smart contract programming [16].

Acknowledgements. We thank the anonymous reviewers for their insightful feedback. This work is funded in part by NSF grants CNS-1314857, CNS-1453634, CNS-1518765, CNS-1514261, a Packard Fellowship, a Sloan Fellowship, two Google Faculty Research Awards, a VMWare Research Award, and by Maryland Procurement Office contract H98230-14-C-0137, ARO grants W911NF11103, W911NF1410358, and W911NF09102.

References

1. Blockhash Contract. https://github.com/amiller/ethereum-blockhashes
2. Etherpot. https://etherpot.github.io/
3. The rise and rise of bitcoin. Documentary. http://bitcoindoc.com/
4. Smart Contract Programming Open Course Materials. http://mc2-umd.github.io/ethereumlab/
5. Ahamad, S., Nair, M., Varghese, B.: A survey on crypto currencies. In: International Conference on Advances in Civil Engineering (2013)
6. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on bitcoin. In: IEEE Symposium on Security and Privacy (2013)
7. Barber, S., Boyen, X., Shi, E., Uzun, E.: Bitter to better — how to make bitcoin a better currency. In: Keromytis, A.D. (ed.) FC 2012. LNCS, vol. 7397, pp. 399–414. Springer, Heidelberg (2012)
8. Bentov, I., Kumaresan, R.: How to use bitcoin to design fair protocols. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part II. LNCS, vol. 8617, pp. 421–439. Springer, Heidelberg (2014)
9. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: SoK: research perspectives and challenges for bitcoin and cryptocurrencies. In: IEEE Symposium on Security and Privacy, SP, San Jose, CA, USA, pp. 104–121, 17–21 May 2015
10. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: IEEE Symposium on Foundations of Computer Science (FOCS) (2001)
11. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007)

12. Canetti, R., Rabin, T.: Universal composition with joint state. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 265–281. Springer, Heidelberg (2003)
13. Chaum, D., Fiat, A., Naor, M.: Untraceable electronic cash. In: Goldwasser, S. (ed.) CRYPTO 1988. LNCS, vol. 403, pp. 319–327. Springer, New York (1990)
14. Dermody, A.K.R., Slama, O.: Counterparty Announcement, January 2014. https://bitcointalk.org/index.php?topic=395761.0
15. Juels, A., Kosba, A., Shi, E.: Rings of gyges: using smart contractsfor crime. Manuscript (2015)
16. Kosba, A., Miller, A., Papamanthou, C., Shi, E., Wen, Z.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. https://eprint.iacr.org/2015/675.pdf
17. Lewenberg, Y., Sompolinsky, Y., Zohar, A.: Inclusive block chain protocols. In: Financial Cryptography and Data Security (FC) (2015)
18. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
19. Pass, R., Shelat, A.: Micropayments for decentralized currencies. In: Proceedings of 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015, pp. 207–218 (2015)
20. Ruffing, T., Kate, A., Schröder, D.: Liar, liar, coins on fire! Penalizing equivocation by loss of bitcoins. In: Proceedings of 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015 (2015)
21. Sompolinsky, Y., Zohar, A.: Accelerating bitcoin's transaction processing. Fast money grows on trees, not chains. IACR Cryptology ePrint Archive 2013:881 (2013)
22. Etheruem Wiki: Serpent (2015). https://github.com/ethereum/wiki/wiki/Serpent
23. Wood, G.: Ethereum: a secure decentralized transaction ledger (2014). http://gavwood.com/paper.pdf