HACK THIS CONTRACT

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

MSc.

by

Aayushi Suresh Jain

May 2020

© 2020 Aayushi Suresh Jain
ALL RIGHTS RESERVED

ABSTRACT

Smart contracts are computer programs on top of blockchains that can be 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 between two parties, the security of the contract program is of utmost importance. Despite prior work in the form of numerous blog posts, Internet discussion forums, DASP top 10, ConsenSys best practices and research papers has been done to tackle vulnerabilities in Ethereum smart contracts, the problem is that students only hear about them in the lecture or read about them online. However, they don't have a concrete notion of what form they take or how they come about. This thesis describes the development process of an educational tool, 'Hack This Contract' (website) aimed at helping students learn/identify security vulnerabilities in smart contracts and also motivate the need for secure smart contract development. Whilst, in the first half of development, additional contracts replicating the Parity Multisig Wallet Hack have been incorporated, analysis of students' feedback has shifted the focus of the second half of development towards realizing the need for secure authentication mechanism and implementation of the same. Ultimately, I have shared my findings, experiences as well as challenges encountered during the design of such a system and discussed to what extent was 'Hack This Contract' effective in addressing its goals.

TABLE OF CONTENTS

List of Figures **5**

1 Introduction **1**

2 Related Work **3**

3 [Approach — Method — Implementation] **5**

4 Results **14**

5 Discussion **19**

5.1 Limitations 24

5.2 Future work 25

6 Conclusion **27**

Bibliography **28**

LIST OF FIGURES

3.1 User Authentication Workflow 9
3.2 Login using Metamask - Step 1. 9
3.3 Allow Hack This Contract to connect to your Metamask account - Step 2. 10
3.4 Metamask popup asking to sign the message - Step 2. 10
3.5 User Authentication successful - Step 3. 11
3.6 Dashboard view for the users not taking the course 13
3.7 Dashboard view for the users taking the course 13

4.1 % of agreement level amongst students on, if they are confident in auditing smart contracts. 15
4.2 % of agreement level amongst students on, if they are comfortable in writing simple smart contracts for hire. 15
4.3 Visualization of how many % of students guessed the correct smart contract, for a particular vulnerability. In this case, ERC20 is the right answer. 16
4.4 Visualization of how many % of students guessed the correct smart contract, for a particular vulnerability. In this case, Coin Flip is the right answer. 16
4.5 % of the agreement level amongst students on recommending 'Hack This Contract' as a primer to a friend who wants to learn how smart contracts work. 17
4.6 Difficulty level of smart contracts across varying number of students. 17

5.1 System Architecture of Hack This Contract 22

CHAPTER 1
INTRODUCTION

"This is the final report of my Specialization Project, a two-semester project required for the Connective Media Master program. This project was a one-person research-oriented project done under the guidance of a faculty member at Cornell Tech"

Coding for blockchain is a relatively new field. There aren't many security standards, documentations or best practices to draw on. Smart Contracts are the crux of all Ethereum Decentralized Applications (DApps). They hold digital assets worth millions of dollars, thus making them a soft target for attackers. Several security vulnerabilities in Ethereum smart contracts have been discovered both by hands-on development experience [15], and by static analysis of all the contracts on the Ethereum blockchain [18]. The vulnerabilities stated in these papers have been exploited by some real attacks on Ethereum contracts, causing losses of money. The most successful attack being the infamous DAO Hack [10] where the thief managed to steal $60M from a contract, before Ethereum was hardforked. Besides this, a significant part of errors in the implementation of smart contracts is caused by a misalignment between the semantics of Solidity, the high-level programming language supported by Ethereum, and the intuition of programmers. 1 in 20 smart contracts are vulnerable to hacking due to poor coding that contains bugs [20].

However, patching security vulnerabilities of decentralized applications on the Ethereum Blockchain is not so straightforward. Due to the immutable nature of smart contracts, it's difficult to upgrade already deployed contracts – a quality that makes them reliable and trustless, but also a precarious minefield. Thus

smart contract developers need to be vigilant and apply defensive programming techniques when designing smart contracts to prevent vulnerabilities in initial design and history from repeating.

Although with constant changes in the security landscape, as new bugs and security risks are discovered, new practices and fixes are also being developed. But not a lot of work has been done when it comes to spreading awareness amongst students and future developers about common smart contract vulnerabilities and ways to mitigate them. Available reference courses [15, 11] on smart contracts that may serve as a blueprint are scarce. This in turn is another major cause of the proliferation of vulnerable smart contracts as documentation of known vulnerabilities has to be distilled through several sources, such as official documentation [3, 8], research papers [15, 18], Internet discussion forums [2] and blog posts [16, 22]. And this is where 'Hack This Contract' comes.

It's an educational teaching tool for blockchain smart contract security. The website consists of 8 vulnerable smart contracts for students to practice attacking. Thus students will learn how to interact with the existing Ethereum smart contract network, understand the nuances of Solidity - the programming language, identify/learn historical vulnerabilities associated with smart contracts, along with ways to fix them and the need for strong security audits of smart contracts.

The development of 'Hack This Contract' is divided into two phases (work split over two semesters), each with its own set of learning outcomes. In Phase 1, 'Hack This Contract' was released as the last assignment for the course CS 5433 - Blockchain and Cryptocurrencies, with an intention to get the students to understand known vulnerabilities and to motivate the need for secure smart contract development. In the later sections, I discuss to what extent was this re-

search hypothesis addressed. In Phase 2, based on the feedback from Phase 1, I focused on enhancing the security aspects of the system and improving the process for the TAs around private key generation and netID linkage, besides the logistics of grading the assignment ('Hack This Contract'). The work done is to highlight the importance of incorporating secure authentication mechanisms when designing such a system.

The purpose of this thesis is to share my findings and experiences in order to aid future developers, course TAs, designing similar systems as well as to identify the limitations and challenges encountered during development of such systems.

CHAPTER 2
RELATED WORK

Much of the prior work done in this field can be categorized as follows:

Smart Contracts: To get up to speed with the Ethereum world, I refer to the Ethereum basics [3, 23]. [17] offers additional perspectives on smart contracts, with a programming point of view. For platforms and use cases, [13] provides an interesting empirical analysis of smart contracts regarding platforms, applications, and design patterns. For smart contracts, information on available university based courses is scarce. However, the authors of [15] were the first to document the teaching of smart contracts as a university course. They report "several typical classes of mistakes [undergraduate] students made such as failure to encode the state machine properly, failure to use cryptography and misaligned incentives, suggest ways to fix/avoid them, and advocate best practices for programming smart contracts." This can be regarded as a reference course. For their lab they used Serpent, a high level programming language in the

Ethereum world. Their pedagogical approach of "build, break, and amend your own program" seemed to be beneficial to teach adversarial thinking. Drawing inspiration from the previous paper, the authors of [11], also designed a graduate course around development of secure smart contracts and documented the lessons learnt aiming to inspire teachers designing similar courses. Based on an analysis of the students' development efforts, answers and feedback, they present insights into problems faced by researchers and developers when dealing with smart contracts programming. The differentiating factor was the focus and the didactic design of the course. The main focus was to teach students how to develop secure smart contracts. And the end assessment was based on the final assignment, where students were asked to implement a project of their choice, after making them go through lectures, workshops with adhoc tasks and live feedback, security challenges on known vulnerabilities and a guided project with tokens. From an implementation perspective, this well-designed course can be regarded as an immediate predecessor to my thesis.

Security Issues: [12] presents a useful survey of attacks on Ethereum smart contracts, whereas [18] not only investigates the security of smart contracts deployed on the Ethereum main chain, but also proposes to use symbolic execution (as implemented in the tool Oyente) to make contracts less vulnerable. The blog posts [16, 22] provide a guide to auditing smart contracts and review relevant attacks respectively.

Best practices and Mitigation: [24] includes the collection of coding patterns with proposals how to mitigate typical attacks. The challenges and new directions for blockchain-oriented software engineering in [21] provided useful insights, as did [19] with their elaboration on validation and verification of smart contracts.

Non Academia: Outside academia and free to use on the internet, there are two projects that I consider as sources of inspiration for 'Hack This Contract'. OpenZeppelin's Ethernaut [7] is a war game comprised of eight security challenges, where vulnerabilities of smart contracts need to be successfully exploited to advance to the next level. This game acts both as a tool for those interested in learning Ethereum, and as a way to catalogue historical hacks in levels. The challenges address known vulnerabilities concerning the fallback function, a misnamed constructor, math issues like overflow, forced transfer of Ether, reentrancy, hidden variables, delegate call, insecure contract interaction, failing transactions, and randomness. The online tutorial CryptoZombies [6] provides a nice gamification of how to develop smart contracts. It introduces the programming language Solidity in several steps. By forming an army of zombies, one learns to program a suite of contracts similar to the popular CryptoKitties [1]. Finally, one is instructed on how to build a app around the zombie contracts. Additionally, there is Underhanded Solidity Coding Contest that tests people's ability to write harmless looking Solidity code. It's a good way to help them understand the shortcomings of the programming language.

CHAPTER 3
[APPROACH — METHOD — IMPLEMENTATION]

The development of 'Hack This Contract' is split over two semesters and hence can be divided into two phases:

Phase 1 (Spring 2019):

The approach followed in this phase can be structured around three tasks in the given order: **Implementation** -> **User Study** -> **Inference and Documentation.**

1. **Implementation -** The website of 'Hack This Contract' was already built before I started working on it. The implementation phase involved adding two additional contracts with autograders - replicating the Parity Multi-sig Wallet Hack [14] as close as possible. The source code of the Parity Contracts was taken from GitHub. Prerequisites needed for developing smart contracts and for interacting with them involves basic proficiency in understanding of languages, tools, and technologies like Solidity, Remix, Geth and Web3.py/Web3.js.

   Following this, the website was ready with 8 challenges in total that students were asked to solve and was released as an assignment for the course CS 5433 - Blockchain and Cryptocurrencies. Every challenge consists of a smart contract that has to be hacked. Students can deploy/re-deploy a smart contract by clicking on the 'deploy/re-deploy' tab. Doing so generates a specific smart contract address that is unique for each student. Every contract also has the 'update status' tab, that internally runs the auto-grader to verify if the students hacked the contract or not. The website also has a leaderboard – a scoring system to track each student's progress.

2. **User Study -** 'Hack This Contract' was a released as an assignment and to know how well it served certain goals, a user study was conducted in the form of a survey [4], that was sent out at the end of the assignment. I had prepared the questionnaire for the survey. Questions ranged from asking students to rate their agreement if 'Hack This Contract' helped them become confident in auditing contracts or writing simple smart contracts, rate the difficulty level of various challenges, guess the smart contract for a particular vulnerability etc. Survey results [5] and its analysis are listed

in the later sections.

3. **Inference and Documentation -** The above process, lessons learnt and the feedback from students was documented as a part of the final report.

Based on the feedback from students after Phase 1, a major problem became apparent - Since the leaderboard was public, students were able to access the addresses that scored full points and traced the same on Etherscan, thereby tracking the entire transaction history involved in hacking of the contract. Hence this turned into a problem specification for Phase 2. Work done in Phase 2 wasn't to extend our insights from what 'Hack This Contract' already helped achieve in Phase 1, but focus was shifted towards working on enhancing the security aspects of the system.

Phase 2 (Fall 2019):

The approach followed in this phase can be structured around two tasks in the given order: **Implementation ->** **Documentation.**

1. **Implementation -** This phase was solely focused around implementing a User Authentication mechanism (as seen in Figure 3.1 below) that was missing in Phase 1, creating tools for the TAs that streamlined the entire process of authentication, creating a private leaderboard and lastly creating two separate flows for two sets of users - students taking the course CS 5433 and users not taking the course. By two separate flows, I mean two views of the dashboard with different welcome messages but the same contracts. Since 'Hack This Contract' would also be used outside of class, it would have another set of users that wouldn't be assigned NetIDs and hence would see a different welcome message with their Ethereum address instead of the NetID.

a) User Authentication - MetaMask is a browser extension that allows web applications to interact with the Ethereum blockchain. For users, it works as an Ethereum wallet, allowing them to store and send any standard Ethereum-compatible tokens (so-called ERC-20 tokens). For developers, it allows them to design and run Ethereum DApps right in your browser without running a full Ethereum node. MetaMask talks to the Ethereum blockchain for you. In order to make transactions on the Ethereum blockchain, one requires ether and this browser plugin allows users to make Ethereum transactions through regular websites.

In Phase 1, students could log into the website by inputting just their Ethereum address (MetaMask wallet address) and no other credentials. However, in Phase 2 MetaMask was integrated with 'Hack This Contract', such that students had to use it to log in to the website. This was because authentication was needed to ensure that it is indeed the user that is logging into the system and not an imposter. And MetaMask includes a secure sign-on process, providing a user interface to manage one's identities on different sites and sign blockchain transactions. The workflow for the same can be seen below:

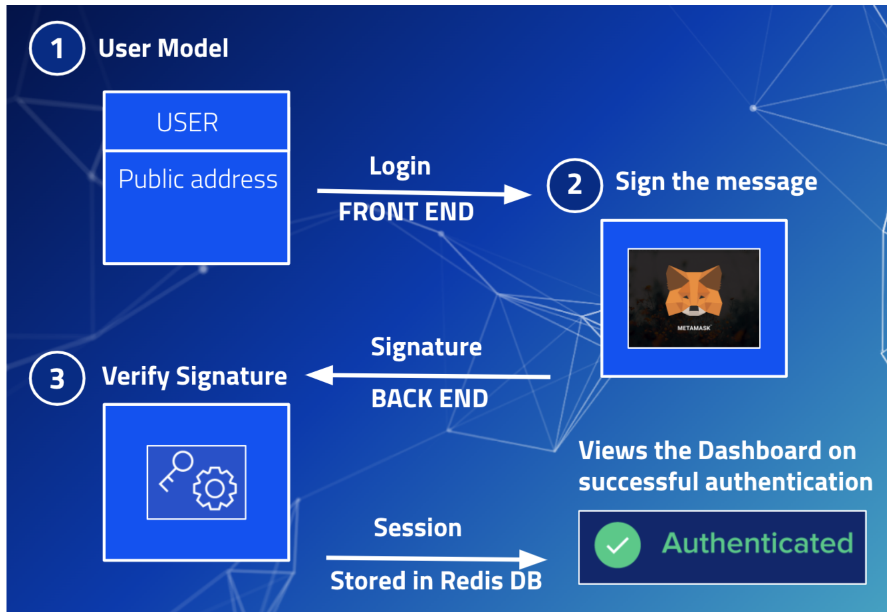

Figure 3.1: User Authentication Workflow

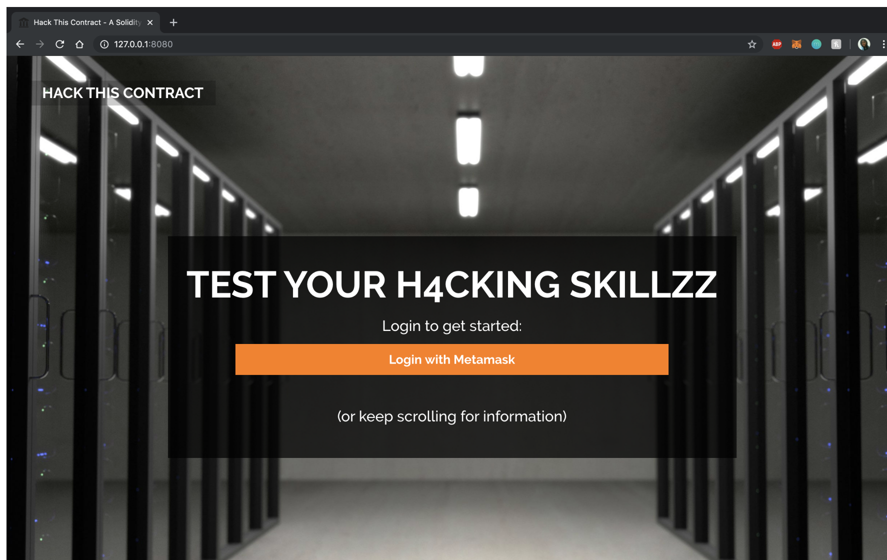

Figure 3.2: Login using Metamask - Step 1.

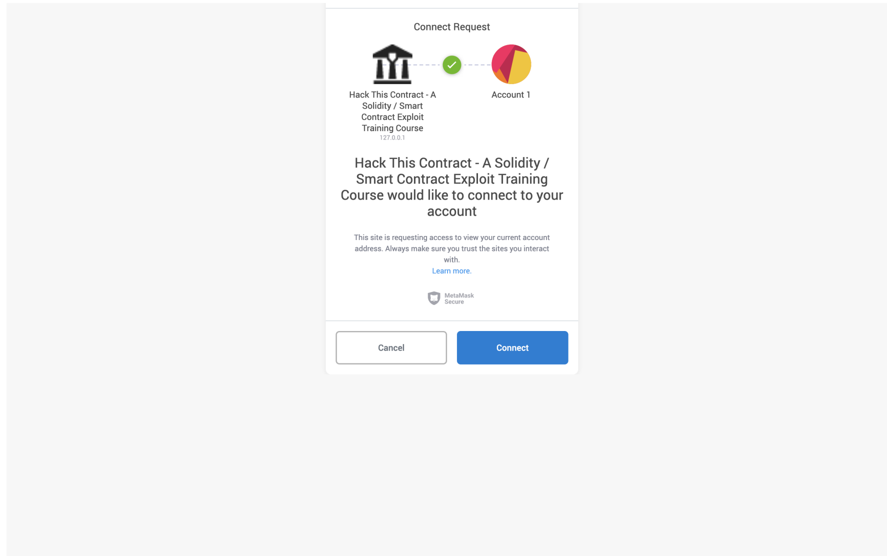

Figure 3.3: Allow Hack This Contract to connect to your Metamask account - Step 2.

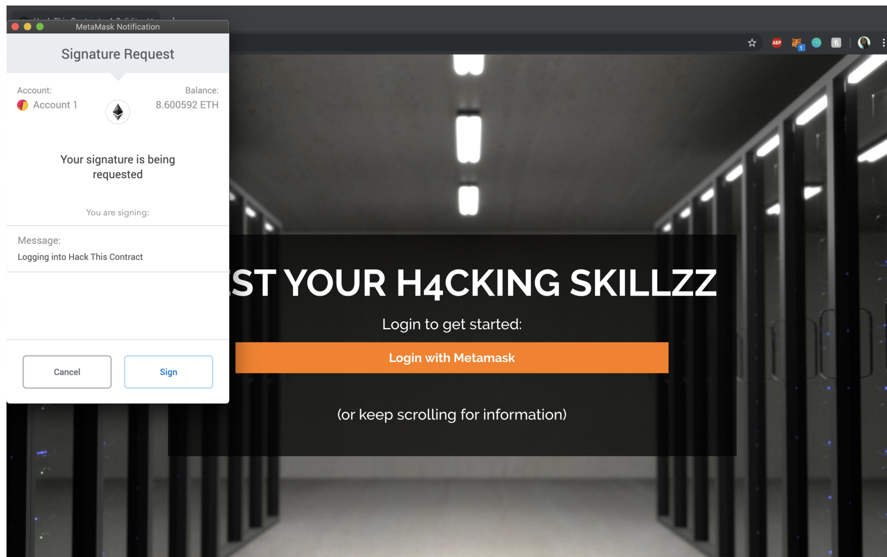

Figure 3.4: Metamask popup asking to sign the message - Step 2.

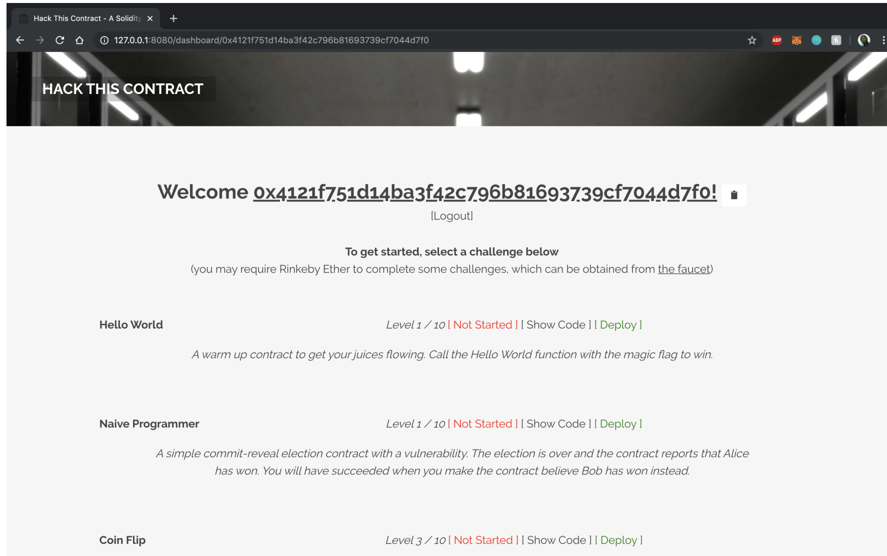

Figure 3.5: User Authentication successful - Step 3.

As seen above, when the user tries to log into the website using Metamask as seen in 3.2, a pop up is shown as seen in 3.4 asking the users to sign a message 'Logging into Hack This Contract'. Metamask internally signs the message using the user's private key, following which the signature is returned to the backend. The verification of the signature takes place at the backend using Web3.py's verify_message(). If the user is authenticated as seen in 3.1, control is routed to the dashboard page that displays the various challenges and a session specific to the user is generated. The session is valid for 8 hours. If the user is not authenticated, the corresponding error message is displayed. Metamask was chosen as the Web3 Provider because it's an easily available and accessible browser extension and commonly used wallet for sending and receiving ETH and ERC20. Several checks have also been placed to ensure that the user first installs Metamask as an extension, creates an account on the same and allows 'Hack This Contract' to connect with the Metamask account as seen in 3.3.

b) Tools for TA - Instead of students creating an account on Metamask and using that address to log into 'Hack This Contract', it was best decided to have unique private keys generated for each student that would be linked to their NetIDs. Hence I built a script to streamline the entire process right from generating private keys and linking them with students' NetIDs, to generating corresponding public addresses for the private keys and preloading those addresses with ether. The end result is a CSV file containing NetIDs, private keys and public addresses of the students. Before the start of the assignment, this file is uploaded by the TAs on the website, and in the backend it is internally fed to an sqlite3 database for keeping track of the scores. The upload endpoint has a secured login mechanism in place. All the students need to do is import the private key given to them and rest remains the same. Lastly, contracts are now spawned (deployment of security challenges) from student addresses as opposed to an address generated from running a Geth node internally.

c) Private leaderboard accessible only to the TAs - As mentioned earlier, at the end of Phase 1, a major problem became apparent - Since the leaderboard was public, students were able to access the addresses of those that scored full points and traced the same on Etherscan, thereby tracking the entire transaction history involved in hacking of the contract. The public leaderboard is what helped students in the first place trace addresses of those who finished all the challenges. Hence in Phase 2, a solution to the aforementioned problem involved removing the public leaderboard, and making it private with secure login, only accessible to the TAs provided they input the right credentials.

d) Separate flows for two sets of users- This is pertaining more to the

front end, where different display messages are shown to the two sets of users. To elaborate further, users taking the course are addressed by their NetID as seen in 3.7and users not taking the course are addressed by their public address as seen in 3.6.

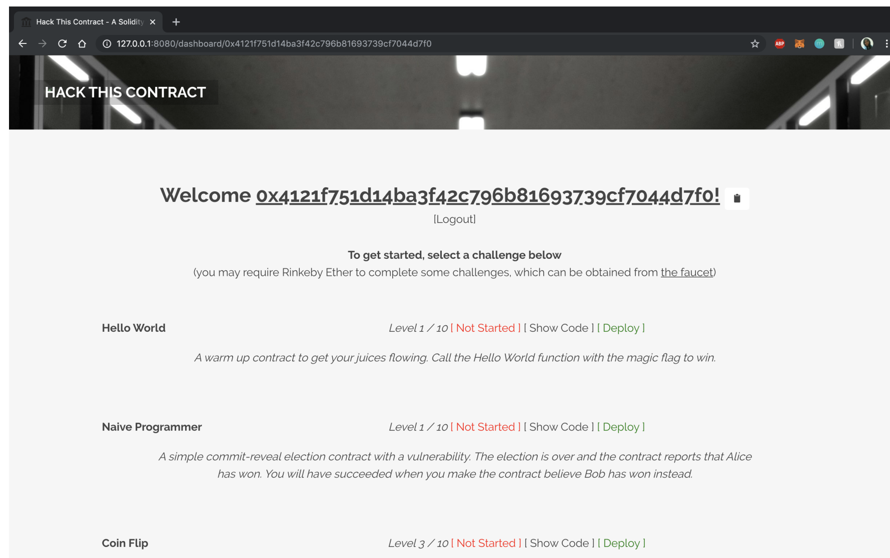

Figure 3.6: Dashboard view for the users not taking the course

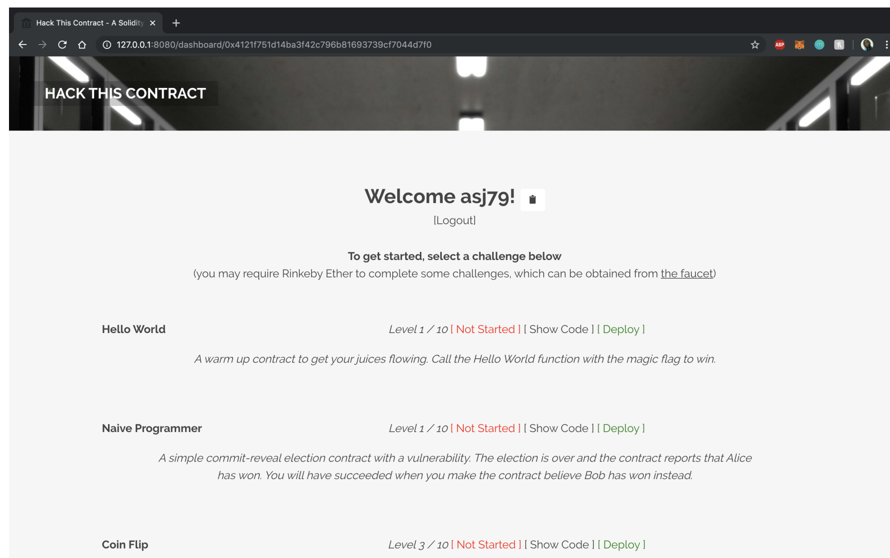

Figure 3.7: Dashboard view for the users taking the course

The source code for this project currently resides in a private repository in GitLab. The link to the website is `https://hackthiscontract.io/` and it will be made live before the upcoming CS 5433 class in Spring 2020.

2. **Documentation -** There was no user study conducted in Phase 2 solely because, no additional challenges were implemented and the work done was just technical changes made to the website. Together they would have acted as a base work to be tested and drawn insights from in the upcoming Spring 2020 semester. The end goal of Phase 2 was to improve the security aspect of 'Hack This Contract' by implementing a user authentication mechanism and creating tools that made the lives of the TAs grading the course CS 5433 easier. Final documentation involved reporting the implementation, limitations, future work, my experiences and challenges involved in designing such a system in detail.

CHAPTER 4
RESULTS

These results pertain to the survey that students took at the end of Phase 1. We had a total of 77 participants (students taking the course CS 5433) who took the survey. In this survey, students were asked to answer 8 questions. Questions concerning rating their agreement from 1 (Strongly Disagree) to 5 (Strongly Agree), choosing the right contract for a particular vulnerability, rating the difficulty level of the smart contracts from 1 (Easy) to 5 (Very Difficult) and general feedback related to 'Hack This Contract'. The results of the same are enlisted below:

- Although 34% of the participants remained neutral, 16% of the participants strongly agreed and 39% of the participants agreed in response to, if solving challenges on Hack This Contract made them confident in auditing smart contracts.

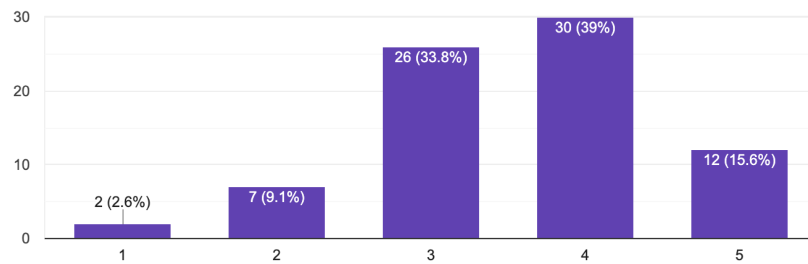

Figure 4.1: % of agreement level amongst students on, if they are confident in auditing smart contracts.

- 13% of the participants strongly agreed and about 37% of the participants agreed in response to, if they were comfortable writing simple smart contracts for hire.

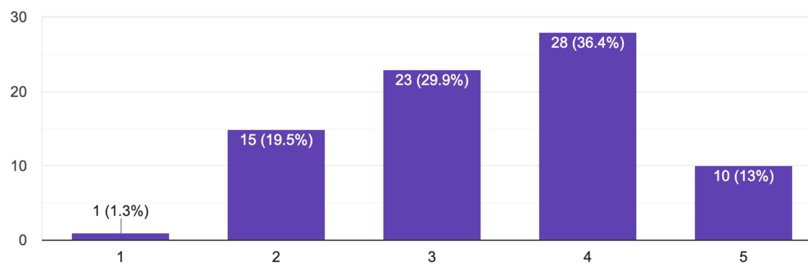

Figure 4.2: % of agreement level amongst students on, if they are comfortable in writing simple smart contracts for hire.

- While 84% of the participants said that ERC20 is the smart contract associated with the known vulnerability called re-entrancy, 91% of the participants said that Coin Flip is the smart contract related to vulnerabilities in public randomness systems.

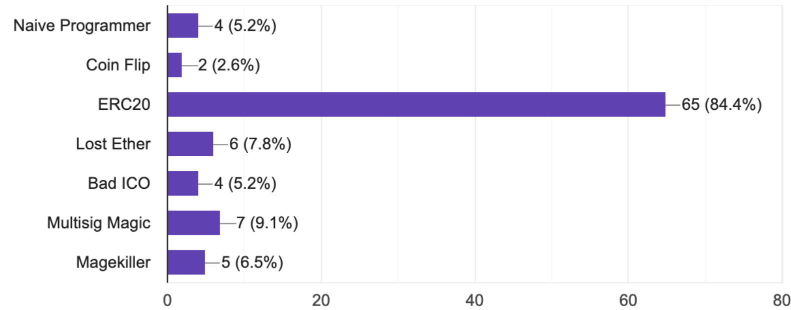

Figure 4.3: Visualization of how many % of students guessed the correct smart contract, for a particular vulnerability. In this case, ERC20 is the right answer.

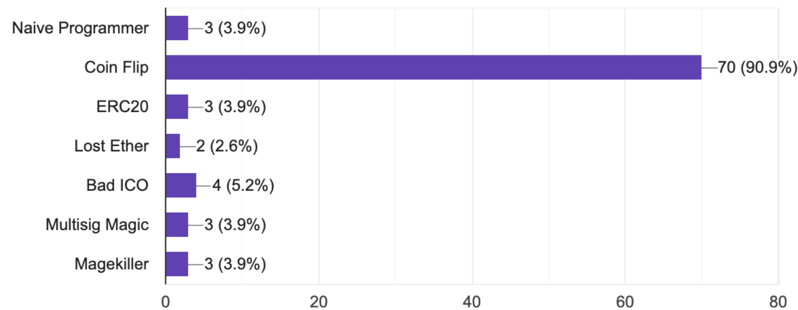

Figure 4.4: Visualization of how many % of students guessed the correct smart contract, for a particular vulnerability. In this case, Coin Flip is the right answer.

- On asking the participants if they would recommend 'Hack This Contract' as a starting point to a friend, who wants to learn how smart contracts work, 34% of the participants strongly agreed, whilst 31% of the participants agreed in response to this question.

Would you recommend 'Hack This Contract' as a starting point to a friend who wants to learn how smart contracts work?

77 responses

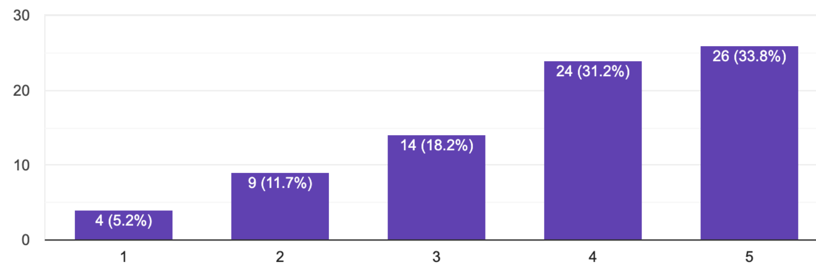

Figure 4.5: % of the agreement level amongst students on recommending 'Hack This Contract' as a primer to a friend who wants to learn how smart contracts work.

- All the participants also gave their perspective on the difficulty level of various smart contracts on 'Hack This Contract', by rating them on the scale mentioned above.

Rate the contracts based on the level of difficulty on a scale of 1-5 with 1=Easy-peasy, 3=Neutral and 5=Very difficult

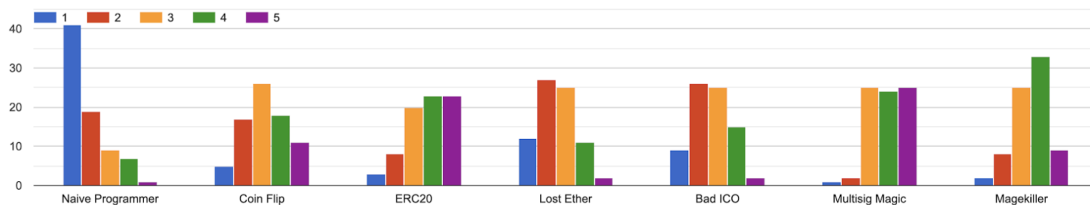

Figure 4.6: Difficulty level of smart contracts across varying number of students.

Some of the general positive feedback from the participants, included state-

ments like: "Hack this contract is a great platform to get students familiar with smart contracts. Moreover, include a variety of smart contracts for not just grading purposes but also to play with them." "I love this homework, and worked hard to become the first full score on the leaderboard. I think this kind of assignment bring a lot of fun, also improve me a lot at the same time." "It was fun working on the contracts. I'd recommend scaling up the infrastructure so that the deployment and grading works more consistently though." "It was a great practice. Would love to see more examples outside the homework just as practice for the final exam." "It would be nice to have another homework where we have to be more creative with smart contracts. E.g. come up with a use case for smart contracts and implement it." "Add more challenges! It was fun to do!" "Pretty good system, fun and insightful project."

General frustration sentiment with smart contracts included statements like: "I felt annoyed by the amount code in Multisig and Magekiller. They were fun hacks, but again, I'm not really learning about vulnerabilities of smart contracts by spending time dissecting and understanding code unrelated to the vulnerability. Trim the fat of these contracts so that I can focus on the important code." "The Coin Flip example also took forever to hack. Even after twenty something random flips, I could never get the randomness to generate a 1."

However, potential for improvement was seen in: "A tutorial on solidity programming will be really helpful." "It would have been really useful to have a short primer, say for 30 minutes, in class on how to develop the intuition for looking for vulnerabilities efficiently." "It would be great if all the contracts use the same compiler version of solidity - it was a bit annoying to have to change it in remix every time" "Some sort of tutorial on how to use remix and what exactly is going on there would be great. There was a learning curve to

get started." "The change of level of difficulty from level one to level 3 problems seemed a lot." "There are duplicate problems such as "Lost Ether" vs "Bad ICO" and "Multisig Magic" vs "Magekiller", which can be solved by using almost the same strategy. Maybe adding more problems with different solutions would be better." "Instructions could be a little clearer. "Naive Programmer" and "Hello World" took way longer than it should just because instructions such as "call hello world with magic flag" were intentionally vague and frustratingly convoluted." "Bug fixes regarding completed challenges being marked as 'Not Started.' Autograder was sometimes slow. Otherwise good!" "Well, a lot of us ended up finishing this assignment on the last day. The leaderboard made it easy to track addresses on Etherscan."

CHAPTER 5
DISCUSSION

Analysis of the Survey Results [5]:

'Hack This Contract' was released as the last assignment for the course CS 5433 with an intention to get the students to understand known vulnerabilities and to motivate the need for secure smart contract development.

The two major takeaways are as follows:

- As expected students felt a bit more comfortable with identifying vulnerabilities than writing contracts. To elaborate further, combining the responses in agreement to the questions, 55% (16% strongly agreed + 39% agreed) of the students agreed that solving challenges on 'Hack This Contract' made them confident in auditing smart contracts. Whilst, 50% (13% strongly agreed + 37% agreed) of the students agreed that they were com-

fortable writing simple smart contracts. However, the gap was smaller than I'd had anticipated. More on this, is further explained in the 'Limitations' section.

- One of the feedbacks was related to most of the students being able to finish the assignment on the last day, as the public leaderboard made it easy to track addresses (of students who finished solving challenges) on Etherscan. Now this was expected behavior from the students, but it made me question the design choices of the system and the need to have some sort of an authentication mechanism in place. As a result, this paved way for most of the work done in phase 2, which involved building a user authentication mechanism for 'Hack This Contract', building tools for the TAs to streamline the authentication process and having a private leaderboard in place.

For questions related to guessing the smart contract for a particular vulnerability:

- 84% of the students correctly guessed 'ERC20' as the smart contract associated with the vulnerability of re-entrancy.
- 91% of the students correctly guessed 'Coin Flip' as the smart contract associated with the vulnerability of public randomness in systems.

Since majority of students correctly matched the smart contract with the known vulnerability, 'Hack This Contract' was successful in getting students to understand known vulnerabilities.

Majority of the students also appropriately rated the difficulty level of smart

contracts. But I feel it's still a subjective measure of the students' perspective.

It was relieving to see that 65% (34% strongly agreed + 31% agreed) of the students as a whole agreed that they would recommend 'Hack This Contract' as a starting point to a friend who wants to learn how smart contracts work.

Although 'Hack This Contract' received positive feedback from the students, but the overwhelming general feedback in terms of room for improvement can be categorized as follows:

a) Need for a technical tutorial

b) Need for additional smart contracts

c) Need for clearer instructions or hints

d) Resolution of bug fixes

All of the above has also been incorporated and further explained later in the 'Future Work' section.

There was some feedback regarding the length of the smart contracts and keeping the compiler version consistent across all smart contracts. These issues have also been addressed later in the 'Limitations' section.

Analysis of the Tech Stack:

Description:

The website uses Flask, a micro web development framework, as a server for the backend. Frontend is built in HTML, CSS and Javascript. Smart contracts are written in Solidity. Student addresses are used to spawn contracts on Rinkeby Testnet. The testing framework including the auto-graders is written in Python. In order to interact with smart contracts, a Geth node is essentially run internally to use various methods of Web3.py/Web3.js. Flask's session instance is used for

session management. Sqlite3 and Redis instances serve as databases to store information related to a particular student's security challenge whereabouts (state of the contract, grade status, score) and session respectively. Python libraries like sha3, secrets and eth_keys are used for generating private and public keys.

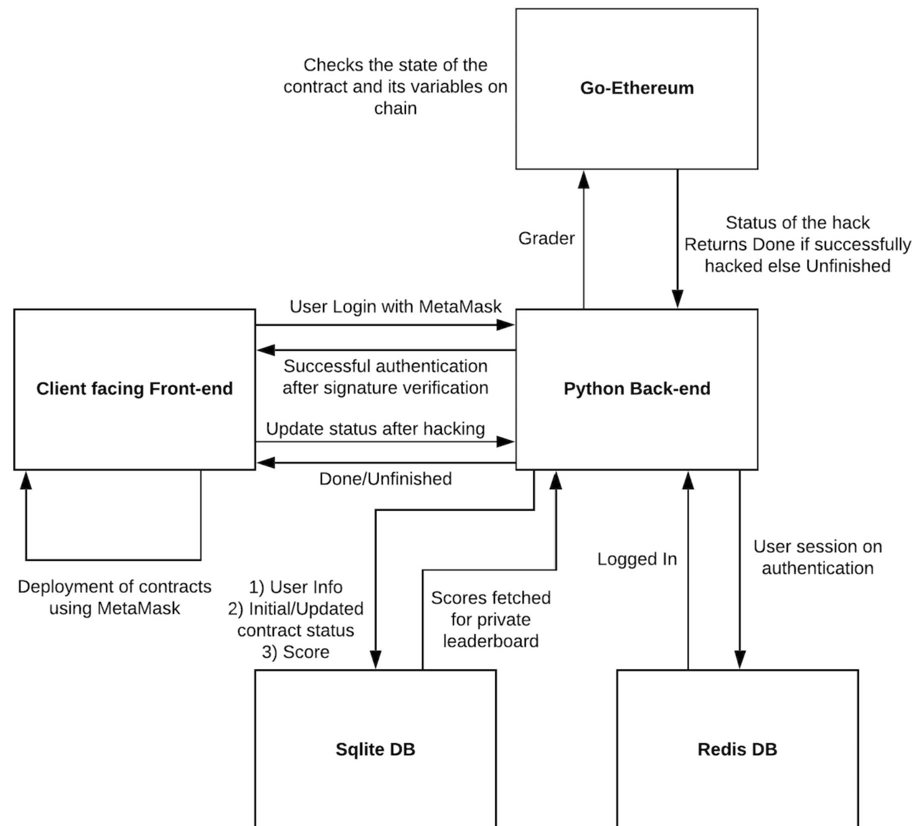

Figure 5.1: System Architecture of Hack This Contract

Performance Enhancers:

Using Flask as the Python Framework definitely boosts performance, because besides being a light weight micro web framework, it has benefits of fast templates, strong WSGI features, thorough unit testability at the web application and library level and extensive documentation. Making the switch from spawn-

ing contracts on the server side to spawning contracts on the client side, using student addresses, helped boost the execution time of deployment of smart contracts.

Challenges related to adoption of a new language and usage of libraries:
The learning curve of Flask is not too high. As with every software engineering product, coding errors stand in the way of progress as was the case with decorators in Flask. However, spending a few hours on the same helped strengthen my understanding of the framework. There were a few challenges during generation of private keys and the corresponding public addresses. These were mainly related to issues with the usage of sha3 python library and picking the right implementation after trying out different ways of randomly generating private keys and the corresponding public addresses.

Distinctive Aspects:
My approach differs from [11] in terms of focus and didactic design. The focus of 'Hack This Contract' is to help students identify vulnerabilities in smart contracts as opposed to develop smart contracts. This is also justified by the survey results to some extent as more % of students felt comfortable auditing smart contracts as opposed to writing them. Although the motivation behind both the approaches remain the same and that is, the need of secure smart contract development.

In terms of didactic design, my approach's emphasis is more on designing the platform itself that consists of vulnerable security challenges and asking students to hack them with no prior training of usage of tools and technologies like Solidity, Remix, Geth and Web3.py/Web3.js. This is in opposed to [11] where students are taught known vulnerabilities through security challenges and workshops, before being asked to implement their own final project. Both

approaches have their own end outcomes outlined and different ways of reaching the same. I also emphasize more on the importance of incorporating secure user authentication mechanisms when designing such systems.

5.1 Limitations

Analysis of Survey Results:

1. Although there is a small gap (5%) between students being more confident in identifying vulnerabilities (55%) than writing smart contracts (50%), I still feel it is inconclusive to deduce this from the study. This is because, I think having 8 security challenges is not a good indicator of this observed behavior. The only way to test if this hypothesis is true, is if adding more number of contracts in the future widens the gap further as opposed to closing it. In other words, I feel that increasing the number of contracts will result in more percentage of students being confident in identifying vulnerabilities than writing smart contracts.
2. With the current smart contracts present on the website, in cases where the users of 'Hack This Contract' are well versed (immediate to expert users) with known historical vulnerabilities, Solidity Programming, Remix IDE etc, the findings of the study won't apply. In order to generalize the findings, more number of smart contracts with varied level of difficulty need to be added. This way we can cater to all levels (beginners/intermediate/expert) of users.

From a technical standpoint:

1. The user authentication mechanism in place is only integrated with Metamask. So if users use a different wallet, the authentication mechanism would keep asking the user to install Metamask. Hence user authentication in 'Hack This Contract' is only compatible with Metamask.
2. I used two databases namely Redis and Sqlite3, to store the contents of the session and the user's progress for a particular smart contract. Ideally I wouldn't want to use two different databases for a system and hence I consider this to be a limitation of my system.
3. Some of Metamask's default methods that help interact with the Ethereum chain, are deprecated and the documentation of the new ones is lacking. This resulted in ad hoc work by using methods of Web3.js instead. I consider this a limitation because one shouldn't import libraries where it's not required.

Lastly, students complained about different compiler versions of Solidity and lengthy codes of the smart contracts. I don't consider these as limitations, because compiler versions are kept different to be consistent, with the version used at the time of the hack. Contracts which are an exact replica of the vulnerable hacks tend to lengthy. Trimming the code can cause it to deviate from its actual behavior or defeat the purpose of building the exact replica.

5.2 Future work

Drawing from the general feedback (survey results) and the aim to test the work done in Phase 2, future work includes incorporating the following:

a) Setting up a technical tutorial: Students felt that there was a learning curve to get started, as setting up the required tools to interact with smart contracts took away more of their time. This instead could have been spent focusing on contract details and learning/identifying vulnerabilities in the contracts. Thus I would address this need by implementing either of the following:

1. Have a link in place on the website to an online tutorial/guide on the usage of languages, tools, and technologies like Solidity, Remix, Geth, and Web3.js for developing smart contracts.
2. Conducting a separate clinic in class on smart contract development, going over the basics of Solidity, Remix, Geth and Web3.js.
3. Add warm up contracts to the website such as calling a function of a smart contract or depleting the balance of a smart contract etc., so that it helps them get upto speed with the smart contract ecosystem.

b) Adding more smart contracts: From a research perspective to gain more insights and also to cater to the students' feedback of 'Hack This Contract' being a fun assignment and their yearning to learn about more vulnerabilities, adding more variety of smart contracts with varied difficulty level, concerning vulnerabilites like Integer underflow/overflow, Unchecked call return values, Race Conditions/Front Running, Block timestamp manipulation, Insufficient gas griefing etc. is the next logical step. The Smart Contract Weakness Classification Registry [9] is also a good reference point to add smart contracts from. It offers a complete and up-to-date catalogue of known smart contract vulnerabilities and anti-patterns along with real-world examples.

c) Having clearer instructions or hints: The current instructions on some of

the contracts such as 'Naive Programmer' and 'Hello World' are confusing and project unclear objectives of what the expected outcome is. Will rephrase those instructions in clearer terms so that students don't spend too much time on the challenge than needed. Having hints is debatable, maybe have hints for the difficult challenges. But this is still subject to future discussion.

d) Continuous resolution of bugs: Bugs are a part and parcel of any software product. However, the issues related to the slowness of the autograder etc. were quickly resolved by scaling the deployment infrastructure. That being said, there will be continuous monitoring and fixing of bugs.

e) Test the work done in Phase 2: Phase 2 involved integrating Metamask with the website for user authentication purposes, creating tools that streamline the same and having a private leaderboard in place for the TAs. All this needs to be tested with students taking the CS 5433 course in the upcoming spring semester for the purpose of a new feedback cycle.

f) Test with Users outside the class: Although 'Hack This Contract' is publicly available, it has only been tested with students taking the CS 5433 course. I would like to test the website with users outside of the class, via some online channels such as an online contest etc.

CHAPTER 6
CONCLUSION

Secure smart contracts are still avant-garde. As expected from the survey results, students felt a bit more comfortable in identifying vulnerabilities (55% of the students responded in agreement) as opposed to writing contracts (50% of the students responded in agreement). Despite the small gap between the

two (we can even owe it to the less number of contracts on the website), we cannot deny the fact that even though there are coding patterns and best practices for most known security bugs, the development of secure smart contracts is not yet a well-established discipline. Besides this, designing such a system comes with its own set of limitations, as the underlying technologies involved, evolve rapidly and documentation lags behind. The available tools are in different stages of development, and even the most mature ones are still difficult to use. Furthermore, secure authentication is of paramount importance when designing such a system. Lastly, there is room for improvement as suggested in the general feedback given by the students, which can be incorporated and tested further to draw more insights.

BIBLIOGRAPHY

[1] Dapper labs inc: Cryptokitties. Available at `https://www.cryptokitties.co` (2018-08-07).

[2] Ethereum reddit page. Available at `https://www.reddit.com/r/ethereum` (2019-12-16).

[3] Ethereum wiki: A next-generation smart contract and decentralized application platform. Available at `https://github.com/ethereum/wiki/wiki/White-Paper` (2018-07-29).

[4] Hack this contract survey. Available at `https://forms.gle/7AjztszCBv1RV3BR8` (2019-12-16).

[5] Hack this contract survey responses. Available at `http://tiny.cc/0k1rhz` (2019-12-16).

[6] Loom network: Cryptozombies. Available at https://cryptozombies.io (2018-08-07).

[7] Openzeppelin: Ethernaut – solidity security challenges. Available at https://github.com/OpenZeppelin/ethernaut (2018-08-07).

[8] Solidity: security considerations. Available at http://solidity.readthedocs.io/en/develop/index.html (2019-12-16).

[9] Swc registry: Smart contract weakness classification and test cases. Available at https://swcregistry.io/ (2019-12-16).

[10] Understanding the dao attack. Available at http://www.coindesk.com/understanding-dao-hack-journalists/ (2019-12-16).

[11] Sack C. Salzer G. Angelo, M. Sok: Development of securesmart contracts–lessons from a graduate course. In *Proceedings of the 3rd Workshop on Trusted Smart Contracts*, WTSC, 2019.

[12] Bartoletti M. Cimoli T. Atzei, N. A survey of attacks on ethereum smart contracts (sok). In *Maffei, M., Ryan, M. (eds.) 6th Int. Conf. on Principles of Security and Trust(POST'17)*, volume 10204 of *LNCS*, page 164–186. Springer, 2017.

[13] Pompianu L. Bartoletti, M. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In *Michael Brenner et al. (ed.) Financial Cryptography and Data Security (FC'17), Int. Workshops*, volume 10323 of *LNCS*, page 494–509. Springer, 2017.

[14] Daian P. Juels A. Gun Sirer E. Breidenbach, L. An in-depth look at the parity multisig bug. (july 2017). Available at http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/ (2019-03-07).

[15] Arnett M. Kosba A.E. Miller A. Shi E. Delmolino, K. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Jeremy Clark et al. (ed.) Financial Cryptography and Data Security (FC'16), Int. Workshops, Revised Selected Papers*, volume 9604 of *LNCS*, pages 79–94. Springer, 2016.

[16] M. Grincalaitis. The ultimate guide to audit a smart contract + most dangerous attacks is solidity (2017). Available at `https://bit.ly/34uTGWz` (2018-08-09).

[17] F. Henglein. Smart contracts are neither smart nor contracts (slides) (2017). Available at `http://hjemmesider.diku.dk/~henglein/smart-contracts-are-neither.pdf` (2018-08-09).

[18] Chu D. Olickel H. Saxena P. Hobor A. Luu, L. Making smart contracts smarter. In *Edgar R. Weippl et al. (ed.) 2016 ACM SIGSAC Conf. on Computer and Communications Security*, page 254–269. ACM, 2016.

[19] McBurney P. Nash W. Magazzeni, D. Validation and verification of smart contracts: A research agenda. pages 50–57. IEEE, 2016.

[20] Kolluri A. Sergey I. Saxena P. Hobor A. Nikolic, I. Finding the greedy, prodigal, and suicidal contracts at scale. In *Cryptography and Security cs.CR*. arXiv:1802.06038, 2018.

[21] Pinna A. Marchesi M. Tonelli R. Porru, S. Blockchain-oriented software engineering: challenges and new directions. In *Sebastian Uchitel et al. (ed.) 39th Int. Conf. on Software Engineering (ICSE'17)*, page 169–171. IEEE Computer Society, 2017.