THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Formal Verification Of Smart Contract

Omar Bassem Jarkas

MSc. Software Engineering

4598179

A thesis submitted for the degree of Masters of Engineering Science at

The University of Queensland in 2020

ENGG7803

Abstract

Formal Verification of Smart Contracts

Omar Jarkas, The University of Queensland, 2020

The innovation of blockchain in the form of Bitcoin has proved to the world the first purely decentralized distributed computing environment capable of handling and maintaining the integrity of data in the form of cryptocurrency. Using an innovative combination of well-known cryptographic tools along with a clever incentive algorithm, Bitcoin was able to introduce the first blockchain environment capable of automating online transaction validation and verification. However, while such technology initially came as a decentralized payment gateway, its potential was not limited to such a domain. Building on such technology, Ethereum came as an environment aimed to overcome all Bitcoin's shortcomings to facilitate more complex decentralized exchange. The addition of the Ethereum Virtual Machine on top of the blockchain facilitated a distributed environment for trustless users to interact and share information and assets. Due to properties such as openness and transparency, there has been a shift in many industries to take advantage of such efficient and trusted technology to model their businesses accordingly. However, while such technology can prove effective to industries, Smart Contracts are merely programmable script that are tied to digital assets deployed on a blockchain. Such immutable coupling of contract's code with real financial assets makes contracts very susceptible to logical vulnerability exploitation. With that being said, contracts deployed on the blockchain must always adhere to ultimate code security, efficiency, and correctness. Rigorous testing, verification, and assessment must be in place predeployment. However, due to the uniqueness of such a system along with its non-deterministic nature, effective testing and verification can be challenging to implement. Often many extra implementations must be constructed. Nevertheless, over the years, the research community has devised many testing and verification tools to adequately secure Ethereum Smart Contracts. One of the most prominent techniques used for Smart Contract security is formal verification. Formal verification is popular in such scenarios since it relies on static testing of code's correctness against devised specifications independent of the execution environment. Such a verification often includes the extra implementation required for accurate system modeling. However, while formal verification is effective, accurate specification models must be constructed for each tool attempting to model Smart Contracts to accommodate such a unique environment and to perform testing of the code's behavior against it. In this thesis, we utilize a formal verification language that implements various verification features based on theorem checking to supply a specification model that relies on the global name-space of the Ethereum environment to supply the extra implementation required for adequate correctness reasoning and verification. Moreover, we introduce a parser that attempts to output a skeleton contract class in Dafny that includes the specification model, recommended design patterns, and standard idioms of the Dafny language to minimize the implementation requirements in favor of direct verification and to reduce the probability of vulnerabilities by maintaining good practice coding.

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, financial support and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my higher degree by research candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis and have sought permission from co-authors for any jointly authored works included in the thesis.

Publications included in this thesis

Example:

1. [1] **Omar Jarkas**, Formal Verification of Smart Contracts, *University of Queensland*(2020)

Acknowledgments

The work couldn't have been possible without the help, guidance, and support of by supervisors.

Dr. Naipeng Dong, Faculty of School of Information Technology and Electrical Engineering, University of Queensland.

Dr. Guangdong Bai, Faculty of School of Information Technology and Electrical Engineering, University of Queensland.

Keywords

Maximum 10 words; Blockchain, Smart Contracts, Ethereum, Formal Verification, Theorem Proving, Dafny, Regex, Specification Model, Design Patterns

Order for the Remainder of the Thesis

Remainder of the thesis should be in the following order

- Dedications
- Table of Contents
- List of Figures and Tables
- List of Abbreviations used in the thesis
- Main text of the thesis
- Bibliography or List of References
- Appendices

Date of thesis submission: 9 November 2020

This work is dedicated to my family, Bassem Jarkas, Rana Kabbara, and Elham Jarkas who supported me throughout the years.

Contents

List of Figures

List of Abbreviations and Symbols

Abbreviations	
PoW	Proof-of-Work Consensus Algorithm
PoS	Proof-of-Stake Consensus Algorithm
EVM	Ethereum Virtual Machine
SC	Smart Contract
FV	Formal Verification
Eth2.0	Ethereum 2.0
Repr	Representation Set
DApps	Distributed Applications

Chapter 1

Introduction

1.1 Formal Verification of Smart Contract

In 2008, a person or a group of people under to pseudonymous Satoshi Nakamoto published a paper detailing the first peer-to-peer network that facilitates online transactions in a purely decentralized manner without the need to rely on third-party operatives such as banks and financial institutions [2]. Nakamoto, using a handful of well-known cryptographic tools was able to provide all the characteristics of a virtual currency in a purely distributed manner. The idea was introduced in the form of a blockchain which uses digital signatures and hash-pointers and a clever incentive mechanism called proof-of-work to automate the functionality of online transactions while preventing double-spending and guarantying integrity and anonymity [3].

The business significance of blockchain came from its ability to provide a secure purely decentralized ecosystem for monetary transactions, however, the technology by itself was groundbreaking and proved to support various business applications not limited to cryptocurrencies [4]. At its core, when the blockchain technology emerged, it offered a fundamentally new way of distributed database systems capable of maintaining integrity, security, and tractability in an efficient and somewhat scalable manner [5]. Such database specification proved to be the perfect environment for the first purely decentralized monetary exchange platform, Bitcoin. However, the blockchain has since evolved to encompass many other suitable applications and industries.

In the second phase of blockchain, after the technical realization and proof of concept, came other application ideas that thrived in the domain of online verifiability and decentralization. These applications can be for the purposes of data protection, voting, distributed anonymous organizations, and identity authentication in the form of Smart Contracts. The complete decentralization of the blockchain environment along with its ability to handle variability, integrity, and immutability specification made it the perfect environment for the inception of real-world binding distributed applications in the form of Smart Contracts [6]. Etheruem, the major Smart Contract blockchain platform, came as a platform that enables the execution and deployment of decentralized applications on the blockchain using a layer of a compilation called the Ethereum Virtual Machine or EVM. This platform came due

to the limitations of Bitcoin to facilitate the construction of complex applications due to its simple stack-based programming language [3]. Smart Contracts are nothing more than computer programs capable of incorporating business logic in computer scripts that are then deployed on the blockchain to guarantee enforceability, integrity, and traceability.

With the need for more effective means of online transaction and interaction, Smart Contract's application, in various fields, are becoming more popular. The idea that it can provide smooth interaction and enforceability between untrusted parties by trusting the platform, can be the basis on which industries thrive. However, the technology is still in its infancy phase [7] and its trustworthiness and security are yet to be fully assessed by researchers [8]. Security issues mainly have to do since the platform is still riddled with vulnerabilities and subtle problems that developers must be aware of and address before attempting the develop and deploy contracts [7].

One of the most important features of Smart Contracts is that it's deployed on an immutable blockchain. While immutability is an essential part of maintaining a tamper-free decentralized ledger that users can trust to uphold integrity and enforceability, it implies that contracts, once deployed, cannot be modified or change. If a vulnerability is discovered in a contract's code, it can be exploited by attackers with no chance of countering it by patching. In many applications having an operational amount of vulnerabilities may be considered manageable, however, due to the immutable direct coupling of code logic with digital financial assets some vulnerabilities or bug in the contract's code can prove catastrophic to all parties involved. This is why making sure that contracts are bug-free predeployment is one the most crucial parts of the design process and is of ultimate importance. Moreover, developers must be aware of previous vulnerabilities, best practices, and security considerations to avoid similar mistakes [4, 6].

For software in general, the process of proving that a code is bug-free is not trivial. However, due to the openness of the Ethereum platform and the unpredictability of the blockchain-based virtual machine environment that is in direct contact with Smart Contracts, finding vulnerabilities in the form of bugs can be extremely challenging. Due to the non-deterministic nature of such an environment, modeling the execution for testing purposes can never accurately depict all possible states of complex contracts. Nevertheless, there are many testing and verification tools recommended by researches for this process, one of the most prominent and suitable technique is formal verification. Formal verification relies on the process of statically verifying code independent of its execution environment by proving the specification correctness of code. Such verification ensures algorithms' behavior by designing modeling techniques to check their correctness.

While formal verification is an effective tool for securing Smart Contracts against vulnerabilities, for it to effectively verify systems, it requires an accurate specification model. Specification models are nothing more than the extra implementation required to be able to model and verify a system using a formal verification technique. However, whether software or hardware systems, finding an adequate specification model to verify systems against is not trivial especially since systems can be very different from each other. Due to the specific purpose of the language used to design Smart Contracts, it cannot be easily modeled using formal methods. The dedication of solidity to only code

Smart Contracts gives it a comfortable space for the addition of special syntax. Such syntax is supplied for the extra functionalities done by contracts to interact with other users and the blockchain. However, while such functionality is necessary, they makes the task of modeling the code's logic into formal verification languages difficult. Since such global functions and variables plays a part in the internal program logic of contracts, the specification model required for blockchain must include the global name-space of the Ethereum environment to include all the mentioned extra functionalities.

While such specification model built is enough for adequate verification, modeling Smart Contracts is not limited to the global name-space and can incorporate other features of the environment such as its non-deterministic behavior. This work introduces a parser and builds upon the work of a GitHub repository [9] to provide an effective specification model able to verify the run-time correctness of Smart Contracts in a formal verification language called Dafny by supplying the extra implementation necessary to model Smart Contracts.

Chapter 2

Literature Review

This chapter covers all the literature reviewed for this project in a comprehensive and informative way. The content includes various articles and journals related to the topic.

2.1 Introduction

The content for this research of formally verifying the Smart Contracts encompasses many related topics including the concepts of blockchain, the field of Formal Verification, Smart Contracts along with their vulnerabilities, and countermeasures. This section goes over all the related topics and subtopics of this new yet steadily growing field of study. Topics include the comprehensive background of the history of anonymous online transactions, starting with ECash technology, all the way to the blockchain revolution, and subsequently the Ethereun Platform and Smart Contracts in the form of distributed applications or DApps. The section starts by emphasizing the significance of Bitcoin that paved the way for purely decentralized environments to handle the exchange of assets with no immediate supervision which later allowed concepts such as Smart Contract and Distributed Application. After the history and significance, the section will give an extensive overview of the Ethereum platform and Smart Contract, outlining the platform's architecture and internal organization while providing a detailed listing of Smart Contract's functionality, significance, challenges, pitfalls, vulnerabilities, and security issues. The listing includes the major vulnerabilities plaguing Smart Contract illustrated by the major attacks that caused the most damage in terms of assets along with root causes, solutions, and recommended best practices to avoid and minimize such threats. After that, it will examine the main language used by the Ethereum platform to develop such applications while presenting arguments for the importance of bug-free code in the process of designing such applications.

After covering the background, the significance of those topics, and their relation to each other, the following literature review will discuss notable articles, publications, and white papers related to the field of formal verification and smart contract along with important ideas and frameworks discovered in the research process. The articles will be examined and discussed according to their relation, significance, and contribution to the thesis. Most relatable work are articles that go into

different methodologies and techniques use in the process of formally verifying Smart Contract along with the major issue and challenges faced and the platforms used in the verification process.

2.2 Background

In the early 1980s, David Chaum published a groundbreaking paper [10] outlining its basic idea in which to facilitate any online payment in a completely anonymous manner. In this paper, Chaum, being a world-renowned cryptographer, was able to avoid the pitfall of double-spend attack while maintaining transactional anonymity. Using the concept of blind signatures, Chaum came up with a way that allowed banks to issue money to customers without knowing their identity. However, while the premise was groundbreaking at the time, DigiCash, the service built using such technology, was ultimately a failure since it couldn't attract an adequate consumer base. While Chaum contributes DigiCash's mishaps to the lack of privacy awareness on the part of regular users, [11] indicates that the main contributing factor in the success of any payment gateway comes from the trust attributed to the system. While the technology solved the anonymity and privacy issues, the technology was completely centralized, and trusting the system implies the reliance on banks and financial institutions to facilitate transactions [11]. Nevertheless, the technology was significant since it introduced blind signatures. Building on such a concept for security and privacy, Bitcoin was born. The introduction of Bitcoin was in a whitepaper [2] in 2008 by an unknown identity under the pseudonym Satoshi Nakamoto. Bitcoin came as a truly disruptive technology to many industries including, but not limited to, distributed systems, banks, and the finTech community.

Bitcoin's innovation came from its ability to build upon many famous topics in the world of computer science and cryptography to automate the functionality of a distributed tamper-free database in a completely decentralized way [3]. Such automation implied the perfect distributed environment for the recording of transactions. Building on concepts related to fault tolerance, it utilized a consensus mechanism named Proof-of-Work based on incentivizing users to operate in a certain way. Along with a handful and well-known cryptographic concept, it was able to offer, for the first time, a purely decentralized distributed system capable of handling online transactions with no entity having implicit control over it while still able to maintain integrity, enforceability, and avoid attacks such as double-spend. The introduction of blockchain paved the way for a new paradigm and financial arrangement in the form of concurrency that includes all the characteristics needed for its verifiable decentralized exchange.

Bitcoin technology came as a proof of concept capable of handling online transactions in a purely decentralized manner by providing a distributed database in the form of a ledger able to maintain integrity and verifiability with no central entity. However, due to it being a new technology, it had numerous limitations. While Bitcoin is fully functional, it had to sacrifice complexity and scalability for decentralization. Such compromise left its architecture limited in terms of complexity. Not having addressed such challenges crucial to any distributed system, Bitcoin was only able to emerge as the platform capable of handling simple transactions in the form of cryptocurrency. However,

such limitations raised intrigue of the technology's potential in terms of complexity, scalability, and efficiency.

Building on what Bitcoin was able to prove, Ethereum emerged as the next distributed blockchain environment capable of handling more complicated online transactions in the form of computer programs [12]. The platform was ultimately able to overcome Bitcoin's limitation and achieve such properties by implementing unique concepts such as computation fees in the form of gas and by appending more layers to the blockchain such as the Ethereum Virtual Machine or EVM. With a new layer on top of the blockchain, the EVM enabled the Ethereum platform to be more than a distributed database. It enabled the platform to operate as one huge decentralized supercomputer [6] while in actuality being a collection of small machines capable of executing complex applications joined together by the blockchain and specialized algorithms to provide consensus. The extra layer of complexity added facilitated the deployment of distributed applications in the blockchain environment. Such applications made the perfect circumstances for the notion of contracts to be agreed upon and enforced on the blockchain given the name Smart Contracts.

The concept of online Smart Contracts that can facilitate the digital agreement between users by trusting an open distributed platform to abide by its integrity and enforceability characteristics is appealing. Such contracts can find their way into many domains that prosper in such environments. Applications such as insurance, supply chain, and IoT can all be modeled in such an environment to facilitate transparent communication between concerned parties.

Due to the immutability of the blockchain along with its transparency, decentralization, and clever consensus protocols, the Ethereum platform was the perfect environment for many applications to be modeled in the form of Smart Contracts. With that said, and since Smart Contracts are nothing more than computer programs or script is written in a high-level programming language deployed on an immutable open blockchain system that anyone can access, security must be taken into consideration when constructing such scripts since logical vulnerabilities can be exploited to cause serious damage and loss.

As mentioned, Smart Contracts once deployed are immutable by nature to be able to guarantee integrity and enforceability. However, the immutability property of it accompanied by Smart Contracts being computer scripts and having the Ethereum platform an open one makes Smart Contracts under threat of vulnerability exploitation on the level of contract code. Smart Contracts mainly hold real financial assets in the form of cryptocurrencies. Such value makes attackers very motivated to exploit code vulnerabilities seeking financial gain. Moreover, due to the immutable direct coupling a contract logic and code with the digital assets, contract exploitation can cause serious irreversible damages rendering the contract useless and untrustable while reflecting some serious financial loss like the case in 2016 with the DAO attack [13]. In contrast to such distributed application, in a normal system's security vulnerabilities and logic, errors can be easily modified and patched in a later version. However, Smart Contract once deployed on the immutable blockchain can never be modified in any form, which burdens the design process by making sure the code is secure and vulnerability free at runtime. Such reasons emphasize the importance of pre-deployment verification and testing techniques.

While patching and modifying the code to cover the vulnerability is off the table, many implementations can be used to maintain security and reduce vulnerabilities. Security vulnerabilities can be found on each layer on the Ethereum platform, however, when it comes to Smart Contracts, security vulnerabilities can be reduced and pitfalls can be avoided by implementing rigorous pre-deployment testing and verification. Also abiding by design best practices and design patterns, and learning from previous platform exploitation as to not repeat the same mistakes are important pre-deployment steps.

Since careful testing and verification must always take place at design time, tools for proving the correctness of the code can be crucial. With that being said, formal verification is one of the most prominent tools used in the correctness process. Formal verification is the process of proving the correct behavior of code by comparing them to the formal specifications. Formal verification is one of the most popular techniques used in Smart Contract verification. That is mainly due to its independence from the execution environment, which is perfect in the case of Smart Contract, and the fact the most security vulnerabilities are directly related to logical bugs making proving correctness a crucial step in the development process.

However, while formal verification is used there are many subfields of it, each with its methods and tools. While formal verification has the same goal in proving the correctness of code, the approach can diverge significantly with each having its methodology and tools, and languages. Hence, the process of formally verifying Smart Contract can also vary significantly. Subsequently, choosing the correct technique to fit the correctness requirement can be nontrivial as Smart Contract can differ from regular code verification due to its uniqueness.

Smart Contracts are typically written in a high-level scripting language. Solidity, a high-level language that is similar to JavaScript, is the most prominent language used to develop Ethereum contracts [14]. However, Solidity differs from mainstream programming or scripting languages in many ways. Mainly due to its unique purpose of developing distributed blockchain applications. The specific purpose of Solidity to design Ethereum Smart Contract gives it a space of flexibility to include the introduce special global functionalities and variables. Such functionalities are mostly implemented to facilitate contract functionality and to interact with the blockchain and users.

As priorly mentioned, formal verification techniques are numerous and usually very general to enable the verification of different languages and platforms while languages used to design Smart Contracts are usually unique and involve much special functionality. Due to the EVM playing a major role in the correctness of the algorithm, to model Smart Contracts using a formal tool a dedication specification model must be implemented that model the environment.

2.3 Blockchain and the Bitcoin Revolution

The idea of blockchain was not new, it was already used for temporal verification of digital documents [15]. However, the innovation was constructing a blockchain data structure to guarantee integrity and verifiability in a purely distributed system to automate the online exchange of currency in the form of Bitcoin. Bitcoin was the first cryptocurrency that was completely decentralized and relied on a clever

incentive mechanism to automate the behavior of its users to operate effectively.

Before Bitcoin, such an idea was deemed impossible. However, the real innovation of Bitcoin didn't come for its ability to verify data integrity nor to combine already known cryptographic primitives and computer science concepts to implement verification. It came from its ability to make participating nodes reach agreement on the creator of the next block in the chain along with data to be written. Bitcoin was the first distributed environment to use a clever consensus algorithm called Proof of Work or PoW to automate the behavior of a complete network of nodes while providing a tamper-free ledger for transactional integrity and cryptographic concepts for verifiability. Bitcoin worked by incentivizing independent nodes to complete for the creation of the next block that contains all the transactions that occurred recently.

While the consensus algorithm handles the role of maintaining parties' cooperation in the creation and verification of the next block, the links between the blockchain are maintained by the use of a cryptographic tool called Hash Pointer. A hash pointer is a key aspect of the blockchain, since the blockchain is a regular linked list data structure, hash pointers are added to guarantee integrity in the purely distributed system. Regular pointer main functionality is to maintain the connection between data by holding the address of the previous data, a hash pointer works the same way, however, it also maintains the hash of the data contained in the last block. Such a tool is implemented to detect any changes made since any modification contradicts the hash stored with the pointer making integrity directly verifiable.

Since the blockchain is a transparent environment with the complete ledger being maintained by each node. Each node can verify all the transactions in the system rejecting a false one with the concept of the digital signature introduced by Chaum. With such characteristics, Bitcoin was able to guarantee the full distributed automation payment gateway.

2.4 Ethereum and Smart Contracts

While Bitcoin was the original proof of concept in the form of a cryptocurrency, its functionality was limited. It was designed only to handle simple transactions with no immediate supervision. Bitcoin, being a distributed system, faced many scalability and complexity challenges. One of the top issues in such distributed environments, which wasn't inadequately addressed, was the halting problem, which can stop the whole system with one infinite loop. In 2014, building on the original proof of concept of Bitcoin, Buterin introduce in his white-paper [12] the Ethereum platform along with its virtual machine. Ethereum came to solve many problems faced in Bitcoin that made it limited in potential. It solved the halting problem by introducing the concept of gas per computation and a new layer on top of the blockchain to handle computer computation and transform the blockchain into a distributed computing platform rather than a merely distributed database.

The addition of such complexity and functionality paved the way for a range of new applications to be implemented in such a distributed environment. Application related to the Internet of Things, Supply Chain, and the Stock Market can all be shifted to the blockchain domain to benefit from its unique

Figure 2.1: Smart Contracts Procedures

attributes. Moreover, it made it one of the most hospitable environments for specific applications to thrive such as Smart Contracts. Smart Contracts are nothing but computer programs deployed on the blockchain and executed by the virtual machine. Such applications can facilitate the smooth collaboration between separate organizations and independent parties. They are called contracts due to the blockchain's ability to guarantee enforceability and specific predefined condition and agreement mimicking real-world contracts [16].

2.5 Formal Verification

Smart Contracts give a good example of applications where failure is unacceptable. Similar to the Ariane 5 rocket, simple software bugs can cause significant damage and loss [17]. While formal methods are not new, such software incidents are pushing more and more industries to consider formal verification as a means to eliminate whole classes of unpredicted errors and capture all possible behavior through specifying them [18, 19]. In contrast to dynamic verification, formal verification is the process of using static analysis based on a mathematical proof transformation to prove the correctness of hardware and software processes [20]. The process includes defining clear formal specification semantics in the form of a brief mathematical description of the algorithm to satisfy clear behavior properties. Such descriptions must outline clear and concise code behavior based upon mathematical models and theorems [17]. Often, the specification of high-level code can be translated into other high-level languages that support specification semantics-based and deductive and inductive reasoning. However, such specifications can be extremely hard to model accurately and deduce due to the generality of the system. Hence, finding an adequate design formalization technique in the form of a specification model can also be difficult [17]. While formal verification's end goal is to prove the correctness of algorithms and systems by implementing code specification, there are many approaches introduced in the field to tackle the challenges of correctness. However, one of the reasons

why formal verification is very suitable to verify Smart Contract application is due to its independence of simulation environments and to prove correctness [20].

Over the year there have been many new formal verification techniques that aim to prove the correctness [21], such techniques vary depending on the specification requirement of different systems. Such techniques include Logic-based approaches, Behavioral Modeling, and Formal Modeling, however, one leading and most popular techniques that have are the tools related to Theorem Proving and Model Checking [22].

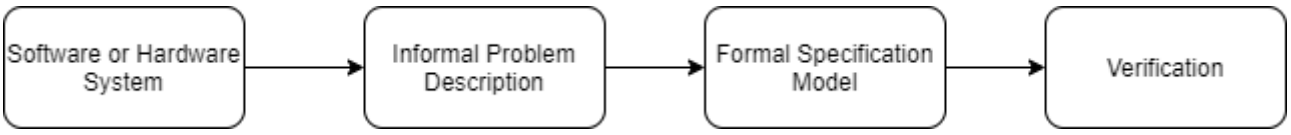

Figure 2.2: Formal Verification Steps

2.5.1 Theorem Proving

The technique of Theorem Proving, the most relied on formalization method, depends on proving software correctness through using rules of logical and deductive inference [21]. The method makes use of theorems and well-defined specifications that describe the behavior of the code through hypothesis and conclusion statements that must be satisfied initially and at the end to prove the code's correctness against its specification. Such statements constructed takes the form of assertions in high-level languages in the form of pre and post conditions to describe the hypothesis and conclusion respectively [18].

2.5.2 Model checking

Also known as Property Checking, Model Checking of using temporal logic semantics to formulate program properties to verify their adherence to predefined software specification models [22]. Such an approach utilizes the program state to build finite-state models to verify system correctness.

Such approaches of formal verification can prove very useful in minimizing security vulnerability. By defining the behavior of contracts and making sure they adhere to such specification, code exploitation through bug detection can be eliminated.

2.6 Smart Contracts Vulnerabilities

The Ethereum platform is the leading decentralized blockchain applications revolution in the form of Smart Contracts. Due to Smart Contract's special characteristics to provide efficiency and transparency, there has been a steady shift to such domains. Cooperations currently are reshaping their business model to fit such technology [5]. However, while this technology is attractive, it is far from perfect.

Scalability and the direct relation between immutable contract code and digital assets remain the major consideration when developing such applications [23] since code exploitation can directly lead to financial loss.

Security in such decentralized platforms can determine the faith of the technology. One simple vulnerability exploitation can raise questions regarding the whole system and cause the whole market demand to plummet [24]. Attacks such as the DAO can inflict serious damage on the entire system. Consequently, developers of Smart Contracts always try to maintain a high level of security consideration of best practice and design patterns.

One of the most important security considerations is learning from previous security vulnerabilities exploited in the system to avoid the same pitfalls. While the DAO attack is one of the most famous that have occurred on the platform since it caused the most serious direct and indirect financial loss since it caused the price of ether to drop by as much as 35 percent over one attack [23], there are many more security vulnerabilities plagued the platform which developer must be extremely aware of. Such vulnerabilities have been extensively examined over many article and assessments, however, one of the most popular vulnerabilities taxonomy survey, according to [23], is the one provided by [25] and was later further discussed and appended to by [13, 26] to classify the different layers related and root causes related to such vulnerabilities.

The Ethereum platform was built to operate on 4 different layers each with specific tasks and vulnerabilities [13]. Starting with the DAO attack on the application layer, such an attack was possible due to a vulnerability caused by the improper design of the interaction between contracts [27] that allows the bypassing of validity checks. In Ethereum, when an external contract invokes a function and then call back into the same one before it finishes leading the contract state not to be updated, opening up the possibility of the contract being drained if the invoked process doesn't update the contract balance while another function executes [23, 25]. Nevertheless, this problem can be solve using mutual exclusion and the use of formal verification models to implement best practices such as the use of send and transfer instead of call. While this is one of the vulnerabilities faced in the application layer of the Ethereum platform, there exists much more direct and indirect vulnerability to must be handled manually by the developer [13, 28]. Indirect vulnerabilities on the application layer also include the use of external unverified external libraries such as the delegate calls, mishandled exceptions, and transaction ordering dependency that can occur due to the blockchain layer [29].

2.7 Formal Verification of Smart Contract

The Ethereum platform, being comprised of more layers of and complexity, brought with a new set of security challenges and vulnerabilities that must be addressed [13]. Due to the uniqueness of the system and being a very new technology, many vulnerabilities don't have any counterparts in traditional applications and practices. Unlike other systems where vulnerabilities can be directly mitigated and corrected, the immutability of the blockchain environment offers no second chances [8]. Article [13] emphasize the importance of developing smart supporting tools for developers to design

Smart Contracts in a secure way such that vulnerabilities will be reduced to their minimum. Due to the complexity of the Ethereum environment and its different layers, security consideration must be maintained on each layer. With that being said, techniques used to test and verify Smart Contracts and the Ethereum platform are numerous and involve many different approaches. Article [13] outlines the major causes of vulnerabilities and causes in each layer.

Having such execution constraints in terms of its behavior and openness shifted the focus on formal methods as one of the optimal solutions to address many Smart Contract vulnerabilities. This is mainly due since most formal methods rely on the static verification of contract correctness irrespective of the environment [20]. However, while formal verification can serve to verify different aspects of such a complex platform, as specified in [13], most vulnerabilities related to Smart Contracts are on the application layer having to do with logic errors and bugs.

Parisi et. al [21] highlights that formal verification's main aspect verification is related to the verification of functionality correctness in which to eliminate the logic vulnerability. Most formal verification tools examined for the thesis uses code base detection of vulnerabilities. The even-though techniques of formal verification can vary significantly, from theorem proving to model checking, each with different tools and specification models, they are all aimed at proving the functional correctness of Smart Contract. This section examines the different formal methods and tools used along with the different frameworks and specification models constructed for the process of verification.

Other than the challenges of verification, formal verification relies on the construction of accurate specification models capable of modeling the system's specification so the adequate verification can take place. While formal verification tools and techniques have evolved to include many frameworks, usually more implementation needs to be considered when using the technique, especially when dealing with complex systems that rely on the construction of specification models for verification. Due to the global name-space of Smart Contracts, formal verification techniques are inadequate solely and rely on the addition of a dedicated specification model to address the gap in the functionality. Nevertheless, over the years, many frameworks, tools, and specification models were developed to model Smart Contracts using formal verification techniques.

F* has been used in more than one work to prove aspects of correctness for Smart Contracts. Article [30] uses such a language to prove functions termination in contracts. Such work is related since Dafny also used special syntax and verifier to prove termination. Dafny solves this the problem of termination ranking, so while the Ackermann function termination might not be verified in F* it can be verified in Dafny. This article provides insights on the work done using F* which is similar to Dafny in the context that it is well-typed and expressive. It uses a verifier to check the specification of code against its implementation and depending on the accuracy and the specification in defining the behavior it can be safely deployed. However, one of the most valuable works for this topic is [31], it also uses the same functional programming language. Article [31] introduces a novel framework that tries to analyze and verify Smart Contracts. The framework relies on proving Smart Contract's functional, runtime, and safety specifications by converting smart contracts into F*. The article contains many similar aspects to the work undertaken in this thesis since both works rely heavily on the translation of

Smart Contract to other formal verification languages while implementing a specification model for Smart Contract verification. The article specifies the technique used to model Smart Contracts with the implementations and assumptions need for adequate modeling of contracts. The article attempts to overcome such gaps by using some assumptions and built-in library calls to replace global variables and space. While such a technique is limited in its capability to prove full correctness, due to it's similarities to the work demonstrated in this project, it is one of the more related works.

Over the years, there have been numerous attacks to exploited Ethereum based Smart Contract, [13] provides an extensive survey of their security and the root cause that eventually led to the attacks. The authors list the major attacks that occurred on the Ethereum platform outlining their details, root causes, and consequences. Article [13] delved deep into one of the most infamous attacks, the DAO, which caused a hard fork of the Ethereum platform to regain over fifty million US Dollars worth of ether from the contract it was transformed to. Such an attack was later named the Reentrancy attack. In addition to such an attack, over the years there have been other famous attacks that caused significant financial loss and stressed the importance of security in Smart Contracts. Atzei et al. [25] provide an overview of some recent and major vulnerabilities in Smart Contracts related to two of the three classes of the theorem platform, Solidity, EVM, and the Blockchain. While [13] provide such taxonomy, [25] goes further into the source code evaluation of the contracts and their vulnerabilities. Also, [32] highlights many pitfalls faced in the development process by shedding light on the source code to point out code vulnerabilities. The two [25, 32] serve as the first articles encountered to offer real Solidity code in the process of taxonomizing the contract vulnerability which is a building block for testing the Dafny platform and was a focus kept in mind so that its template and the parser would be able to detect such code and offer best practice to avoid them. The vulnerabilities cause outlined include the famous reentrancy, typecast, gasless send, exception disorder, and stack size limit, all of which was the stepping stone in my research the building blocks of the project since they all deal with vulnerabilities on the solidity level. Such articles laid the foundation of the research and helped build a clear idea of the vulnerabilities are the root causes, as well as, giving a more in-depth understanding of the internal structure of the Ethereum environment.

However, one of the significances of this work is related to the emphasis given on the importance of designing a framework for handling the verification of Smart Contract while keeping in mind design patterns and gaps. Moreover, on the topic of failure to capture contract behavior due to its unique functionalities and characteristics, Abdellattif et al. [33], introduces an example of a Solidity contract formally verifying statistical model checking technique introduced in the article. The technique is based on the Behaviors Interacting Priorities Framework or BIP. The paper introduces a new approach for handling and verifying Smart Contracts while masking some of its gaps. It also covers the importance of the analysis of contract code on the blockchain while giving a clear depiction of the methodology followed in the verification process.

In addition to the mentioned frameworks that tackle source code specification, [34] introduces a technique that attempts to verify Smart Contract on the bytecode level. Using a framework named K along with some formal semantics, it attempts to specify the behavior of the code on the bytecode

level while giving an extensive overview of the internal organization of the EVM byte manipulation operation while highlighting the specific abstraction that must be addressed to further extended the limits of reasoning about Smart Contracts. Such abstraction can be classified as the specification model for such a project. The tool was implemented on different Smart Contracts vulnerabilities to test for accuracy. The tool was able to perform according to its specification in adhering to verify the contracts. While such work is on the bytecode level, it is one of the most relatable, since it relies on specifying the behavior of the contract using a technique that can be classified as theorem checking and best practice to minimize vulnerabilities.

Another very similar work was done by Hirai [35] to provide an interface for modeling Smart Contracts in higher-level languages. The paper's main contribution is related to the work done in this thesis since it offers a similar specification model to model smart contracts in any language. While our specification model allows the modeling of Smart Contract in Dafny, [35] constructs an interface that implements many of the interaction between Smart Contracts on their environment by defining an interface in Lem language and the necessary environments fields as specification. The framework is also used by [36] to implement for formal modeling of Smart Contract verification using an interactive theorem prover framework, Isabelle/Hol. The work done by [36] is significant since it attempts to answer similar questions related to the logical verification using expressive language while providing a valid formal model to such a complex environment.

Article [21] provides one of the most extensive and detailed systematic literature reviews of the work done on using formal verification techniques to verify Smart Contracts. The article achieves such a review by attempting to address the main questions related to the formalization approaches used along with the framework introduced and implemented in each one. While outlining the work, it highlights the challenges each work addresses while also giving an overview of the main security challenges that need addressing. Besides providing a good foundation for the literature review, this work is important since it emphasizes the importance of functional verification of Smart Contract which corrects an effective specification model which is the basis of my work.

Chapter 3

Method

This section covers the methodology used in the formal verification of Smart Contracts. After the survey of different formal tools, techniques, and languages, Dafny, a formal verification language, was used to verify the logical correctness of Smart Contract by reasoning about the contract's behavior. However, the process of formalizing Smart Contracts in Dafny is not trivial due to the interaction between Smart Contracts and their environment. A special specification model must be in place to model Solidity Contracts in Dafny. This section will provide a detailed overview of the work done in this thesis along with the methodology and approach of using Dafny language to formally verify Smart Contracts. It will cover the specification model used as well as the tools introduced to assist developers with such implementation.

3.1 Introduction

Smart Contracts are nothing more the computer programs running on a distributed transparent environment. The need for secure and bug-free code in the design process of Smart Contracts drove to demand to adopt static tools for security assessment, verification, and testing. Formal verification emerged as one of the leading techniques used for such programs. Its significance stems due to the independence of formal verification on dynamic testing environments, which, in such a distributed scenario, can be hard to emulate. Formal verification, as mention prior, attempts to eliminate unpredicted vulnerabilities by proving the correctness of code through the construction of formal models that verify the behavior of code by modeling it via a formalization technique and a specification model. Moreover, systems in the world of software and hardware can differ significantly in design specification and behavior. Such formal models must always be customized and well modeled according to the system's purpose, behavior, and environment. However, the process of attempting to model such a system can be extremely difficult since there is no precise guild line to follow when designing the verification model due to the uniqueness and challenges every system presents. Ethereum and the block-chain environment is still an infant field of study, and the uniqueness and openness of such environments make the process of formal modeling and verification a complex task. Nevertheless, there have been

many attempts at formal modeling of the Ethereum platform and Smart Contracts. Such attempts are usually achieved by attempting to use already well known formal verification tools and techniques. Such techniques are used as a basis to build specification models for the prospect of using them to model such environment and contracts. Such tools are discussed in the literature review as part of the research done for the project. In this project we built on the work of [9] who used the Dafny formal language as a tool to build an Object-Oriented Template to try to design a Smart Contract model in Dafny for the successful verification contract's correctness. Besides, Dafny and Solidity both implement many different design patterns that are recommended from both languages, and there is still the process of implementing the model and the design patterns. Hence, this thesis introduces a simple parser that attempts to convert Smart Contracts into a Dafny skeleton that implements all the correct design patterns and best practices for successful reasoning. By implementing the recommended design patterns and best practices of both languages along with the reasoning about the correctness of the code functionality at run-time, logical vulnerabilities can be reduced dramatically. This chapter covers the implementation, methodology, and utilization procedure of the specification model and the parser.

The contribution of this thesis can be divided into two parts, the first relates to the adjusting and adding on to the work of a GitHub repository [9] to supply a specification model template capable of bridging the gap between the Solidity global name-space for modeling Smart Contracts and mimic their real environment. The second part is related to developing a Java Parser that can convert Solidity Smart Contracts into Dafny by injecting the specification model and implementing Dafny recommended design patterns for the direct verification implementation.

3.2 Dafny

The concept of reasoning about algorithm and code is not new, however, formal verification has only recently been gaining popularity after more and more industries are realizing its importance in the process safety and system's security [17]. With that being said, more and more techniques, languages, and tools are being constructed to formally model and reason about systems, environments, and algorithms. One of the emerging technologies is a Formal Verification language tool named Dafny. Dafny is a formal high-level language that attempts to prove the correctness of code by implementing special syntax and automatic theorem checkers and solvers. Dafny is a formal language designed to specify, write, and verify programs [37]. It is an imperative, class-based formal verification language that uses verification conditions semantics to enable the construction of logical formulas considered as code specification, in the context of theorem proving, to reason about algorithms correctness. Such reasoning, however, must be done manually by the developer. While Dafny provides the syntax and the built-in solver for such a verification [38], developers must implement the necessary specification custom to each algorithm. The language supports special syntax and modifiers that facilitates automatic formal theorem proving using special semantic used as a hypothesis, conclusion, and axioms.

Through the use of such semantics, Dafny allows the construction of code specifications through the writing of logical formulas. Then it performs a condition verification of the logic of the code

written against such specification. The specification, while not trivial to implement, are expressions that define the exact behavior of the code, that are then compared to the code body. With the use of such special semantics. Any code can be verified for correctness by translating the algorithm to Dafny.

The Dafny language utilizes formal concepts in the form of clauses and axioms to enable the writing of specifications. Such specifications are then integrated with normal methods, classes, iterative and conditional constructs found in any programming languages to verify their correctness. The representation of theorem providing specifications such as the hypothesis and conclusion in Dafny is in the form of pre and post-condition, respectively.

```
method multiply(x: int, y: int) returns (r: int)
    requires 0 ≤ x ∧ 0 ≤ y
    ensures r = x * y
```

Listing 3.1: Example of Method Specification

Method 3.1 depicts the specification in terms of pre and post-conditions in a simple multiply method similar to formal verification using theorem proving. The code presents the syntax used for the representation of specification in Dafny. As the name indicates, the **requires** clause specifies the acceptable values required to the method to satisfy the post-condition specified. The post-condition is specified by the **ensures** clause which is a predicate that must be satisfied at all times [39]. The more accurate the post-condition specification expressions are, the more precisely they describe the behavior of the code and eliminate unforeseen scenarios and logic vulnerabilities. When the code is translated or constructed, the automatic verifier, built-in the language, tests the logic of the method's body against the specified behavior state for the presence of any run-time error and bugs.

Moreover, such special syntax is not limited to concepts concerned with theorem proving. The language is also equipped with more specification clauses such as the lexicographic termination tuple metric in the form of **decreases** clause that verifies the graceful termination of iterative programs and recursive functions.

In addition to the three method specification clauses specified, [39] mentions that Dafny enables the specification of the program state in an object-oriented sense. Article [40] introduces the theory of dynamic frames which enables the framing of specification in a modular form to maintain the idea of frames being modified, thus eliminating classes of unforeseen modifications. By implicitly specifying the segment of the code to be altered, bug elimination, and correct reasoning of the code frame can be established. Such framing is used in Dafny using the **modifies** clause which outlines the code reference that can be modified by a method so that no other indirect program state can be reached since the **modifies** clause account for all possible methods updates.

Also, a new concept in Dafny which is used in the project to implement the contract template is the idea of traits. Traits are similar to classes, however, they maintain a list of static codes without the need for explicit initialization and declaration.

3.2.1 Features

The Dafny language was built with correctness in mind [39]. With that building said, it contains many unique sets of features, concepts, and tools that facilitate the correct reasoning of code. It uses concepts such as Ghost State that can be in the form of variables and functions for maintaining coherence and links in reasoning throughout the proof. Such a feature is combined with the modifies clause to introduce a standard idiom, Repr, short for Representation Set, which is a ghost field that contains the set of classes and objects part of the encapsulated data being verified [38]. Using modifies Repr maintain the state of a class, methods are explicitly given the right to modify the set of objects that are part of the representation. Such design patterns can be very beneficial in the process of verifying Smart Contracts.

The verification features imply the correctness of the entire program by verifying the method in a modular fashion. At its core, Dafny is a language to prove functional correctness. As a part of the function included in a class of data structure, codes correctness can be implied by its ability to prove partial correctness [38]. When programs get larger it becomes harder to reason about their correctness in Dafny [41]. To help in abstracting over layers, methods, and modules for more coherent and linked correctness, Dafny supplies class definition with a recommended design patterns to reason about the entire class. Such invariant can help the maintenance of a dynamic frame and model more accurate specifications to define the behavior of it. Dafny idiom for classes and data structure invariants in Dafny is in the form of a predicate, a type of function in Dafny, named Valid() [41].

```
class C {
  ghost var Repr: set<object>;
  function Valid(): bool
    reads {this} ∪ Repr;
  { this ∈ Repr ∧ ... }
  method Init()
    modifies {this};
    ensures Valid() ∧ fresh(Repr - {this});
  { ... Repr := {this} ∪ ...; }
  method Update()
    requires Valid();
    modifies Repr;
    ensures Valid() ∧ fresh(Repr - old(Repr));
  { ... }
  ...
}
```

Figure 3.1: Dafny Class Design Pattern

The Dafny class 3.1, taken from [38], represents the use of class invariants in the verification process along with the design patterns recommended for best practice reasoning. Additionally, Dafny also utilizes ghost variables to facilitate smoother framing. In the form of a ghost variable name Repr, Dafny recommends a standard idiom for representing the different sets of objects in the program's states [41]. In simple terms, it is a set that contains all object references used in verification. After such initialization the Repr is included with the reads and modifies clauses that Dafny utilize to permit the read and writing the object references. This is done to give Dafny the power to implicitly define what reference can be used and modified. The Representation set can now be used as part of the coherent condition of the class invariant to ensure it's always satisfied. If the set is modified or any new references are added, it is explicitly specified. The final standard idiom for the correct representation of frame introduced by Dafny is included in the post-condition of any function that assumes modification of the frame. The condition is in the form ensures fresh(Repr - old(Repr)) which states that any modification of the frame must be in the form of a newly created object unless the Repr is implicitly modified [41]. This post-condition also introduces a special syntax fresh that indicated the creation of a new object reference in Dafny.

3.3 Dafny Template

As mentioned prior, the process of formal modeling doesn't rely on the execution environment for verification, it relies on the construction of the system logics by specifying their behavior using a specification model [29]. However, the implementation of an effective specification model can be a challenging concept. Since every system has its unique environment and modeling criteria, the process of formal verification can differ significantly based on the tools used and the specification model devised. Many tools and techniques have been built to address such variance in the specification and to attempt to effectively model Smart Contract. In this work, we build on the work of [9] to provide a Dafny specification model that attempts to statically verify Smart Contracts using Dafny standard idiom and best practices. The model achieves such modeling by implementing the global EVM namespace that enables the fake interaction between the modeled contract and its environment.

The specification model is nothing more than the extra implementation required to model a system using a formal verification technique. As mentioned in [21], the main aspect of Smart Contract formalization revolves around its functionality. However, such functionality relies heavily on the EVM environment and other contracts implemented in the language used to develop them such as Solidity. Hence, to accurately model Smart Contract in Dafny, the specification model should be based on such functionality gaps.

Building on the work of [9], the specification model devised for modeling Smart Contracts in this project is a Dafny Template that contains the core verified global name-space of the Ethereum platform. Since the interaction, many of Smart Contract's functionality relies on the interaction between them and the Ethereum environment. The specification model enables Smart Contract to access the global name-space of the Ethereum platform while verifying the contracts in Dafny. Such a specification

model provides Dafny with the functions and variables required for effective and adequate verification.

An overview of the template along with its uses and detailed importance is covered in this section. Snippets of the code are presented as listings and referred to so that a clear illustration can be depicted.

```
trait address{
    var balance: nat
    var msg: Message
    var block: Block
    ghost var Repr: set<object>
    predicate Valid()
        reads this, Repr
    method transfer(payer: address)
    method send(amount: nat, sender:address)
}
```

Listing 3.2: Contract template skeleton

The template, written and verified in Dafny, is comprised of one Trait named address and two classes, Message, and Block. The trait class models the address datatype used in solidity to present users and contracts [42]. The address datatype is the data-structure included in the specification model that enables the use of the global namespace of the Smart Contract environment. By extending the trait to the Dafny modeled Smart Contract, the contract can access functions such as transfer and send. Such functions are fully verified in the template and recommended for transactional interaction in Ethereum due to the evolving security consideration. The template also utilizes the balance global variable which allows the reasoning of the contract's balance while using the global functions. Listing 3.2 provides a clear outline of the global variables and functions used in the specification model. Moreover, the msg variable included in the trait is a representation of the msg.sender and msg.value found in the solidity language that acts as metadata for the user's interaction with the contract by invoking its functions. The attributes are supplied by the Message Class that include the two variables sender and value. Listing 3.5 depicts the Message class implementation in Dafny. Finally, the block variable is involved in the name-space by supplying the block characteristic and specification that may prove important in the contract reasoning.

The template serving as the specification model contains the functions that enable the interaction of Smart Contracts with their environment thus providing a more effective verification process by using the address trait data-structure. The trait essentially can be used in two ways. The first way includes extending the class into the Dafny modeled contract so that functions in the name-space can be accessed and used in the contract class directly. With such implementation, the class can utilize the global functions and variables as part of the contract name-space in Ethereum. Listing 3.3 illustrates the simple implementation of the specification model in Dafny by extending the template to serve as the super-class.

```
include "contract.dfy"
class SimpleAuction extends address { }
```

Listing 3.3: Extension to Dafny Smart Contracts

The second technique is using the address as a datatype to contract address. Solidity utilizes a datatype address for variables to need to hold other contract's and user information in the contract, by using the trait address in Dafny as a data type, the template can adequately mimic the addresses in Solidity and access its metadata in msg, Message, and global functions.

```
1     address public beneficiary;
```

Listing 3.4: Solidity DataType

The Example shows 3.2 a beneficiary address datatype in solidity. By using the address trait in Dafny, the variable beneficiary can have all the functionality address datatype has. With such declaration, the beneficiary can invoke the use of the global name-space functions and variables as if it was an address in the Ethereum environment.

Listing, 3.5, represents the Message class, mentioned prior, which enables the representation of the user's metadata in Dafny as a part of the global name-space. Also, it shows the Block class which contains attributes related to the block characteristics that might be useful in the verification process.

```
class Message{
    var sender: address
    var value: nat
    var data: nat
    ghost var Repr: set<object>
}
class Block{
    var timestamp: nat
    var coinbase: address
    var difficulty: nat
    var gaslimit: nat
    var number: nat
    ghost var Repr: set<object>
}
```

Listing 3.5: Template's Message Class

Moreover, as it is represented in the listings mentioned above, the classes and traits found in the template involve Dafny's design patterns discussed previously. Such design patterns are in the form of **Valid** found in the trait and class to maintain the condition for the representation set represented as Repr which maintains the frame reference of the classes involved. Such design patterns are implemented in the parser and included in the process of extending the template to model Smart contracts.

Thie section covered the methods and techniques used to implement the specification model in Dafny. While the specification models are not perfect and don't guarantee the complete verification of code, it significantly assists in the modeling of Smart Contracts in Dafny. Such modeling need also to rely on the security verification, design patterns of Dafny, and recommended best practices for the correct verification and the reduction of logical vulnerabilities.

3.4 Parser

While the template can serve as a good specification model for formally verifying Smart Contract in Dafny, it still needs extra implementation. The implementation required comes in the form of correctly extending the template package into the converted class, converting Smart Contract's function and variables into Dafny, and implementing the recommended design patterns and best practice of the Dafny language. The section covers in detail the parser constructed for more implementation assistance. It covers its primary purpose and objective as well as its design and methodology. Written in Java, the parser attempts to produce a Dafny contract class skeleton to cover all the extra implementation and design patterns so that verification can be directly implemented.

The parser is designed so that the specification model designed for the modeling of Smart Contracts can be easily implemented. The parser works by reading the Solidity written Smart Contract to extract the necessary information and output a correct contract class skeleton in Dafny. The parser initially checks the version of the Smart Contract, since the parser is not compatible with version breaking changes introduced by solidity after version 5.0 [43]. After checking the version compatibility, it starts to extract the required patterns. Such patterns are specified to extract the contract name, contract's functions, variables, and modifiers and keeps them in Java data-structures for one-to-one mapping to Dafny.

The code is written using Java Regular Expression or Regex to extract the necessary code pattern required for the conversion. Java Regular expression is a comprehensive library that requires a learning curve to be able to use. It is used to search for specific patterns in files to be extracted. In the parser, it is used extensively to extract the necessary data for the construction of a class skeleton that implements the design patterns and the specification model.

The Java language relies on two main classes for regex manipulation [44]. These libraries are the **Pattern** and **Matcher**. The first is for defining the pattern to be searched for while the second is for performing the search. Code 3.6 illustrate an example of the usage of the two libraries is defined and detecting patterns in Java taken from an online repository.

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class MyClass {
  public static void main(String[] args) {
    Pattern pattern = Pattern.compile("w3schools", Pattern.CASE_INSENSITIVE);
    Matcher matcher = pattern.matcher("Visit W3Schools!");
    boolean matchFound = matcher.find();
    if(matchFound) {
      System.out.println("Match found");
    } else {
      System.out.println("Match not found");
    }
  }
```

```
}
```

Listing 3.6: Example of Regex taken from [44]

The parser essentially utilizes such libraries to define regex patterns for the necessary information from the Smart Contracts. As mentioned, the parser attempts to extract the name of the contract, its functions, variables, and modifiers. Each extraction needs its dedicated expressions to detect and distinguish between the patterns and place them in the dedicated data-structure so they can be mapped and written into the skeleton Dafny file that models the contract. That parser is supplied with a class named FileParser that is equipped with the specific functions required for the matching and detection. However, the first step requires the complete filtering of all the contract's comments. Since we are using a pattern detection algorithm in Java, we must ensure the elimination of comments as not to interfere with the detection. The regex 3.7 code is used for the removal of all comments before extracting anything else. The code uses a famous String function to replace detected matches regular expression with another. In this case, we are removing them by leaving the second parameter empty.

```
String line = contract.nextLine()
                  .replaceAll("(?:/\\*(?:[^*]|(?:\\*+[^*/]))*\\*+/)|(?://.*)","");
```

Listing 3.7: Example of pattern used to remove comments

After removing the comments. The parser utilizes other functions in the FileParser Class to extract the name of the contract, variables, and functions. Listings 3.8, 3.9, and 3.10 depicts the regular expression patterns used to extract the contract name, functions, and variables respectively. After extraction, the code is design to place them into a dedicated data-structure to perform filtering and mapping of the variables and functions to Dafny utilizing another class that writes the extracted data in the skeleton Dafny file in the form of a class with the same name as the contract.

```
    Pattern.
        compile("(?:(\\w+(?:\\.\\w+)*)\\s*)?(contract)\\s*(.*?)\\w\\s*(.*?)\\{");
```

Listing 3.8: Get Contract Name

```
    Pattern.
        compile("(?:(\\w+(?:\\.\\w+)*))?(constructor|function|modifier)\\s*?)\\(");
```

Listing 3.9: Get Contract Functions

```
    Pattern
        .compile("(?=address|uint|mapping|bool).*\\s+[\\w+\\s+]+;$");
```

Listing 3.10: Get Contract Variables

The parser reads the Smart Contract file and extracts the name of the contract to name the Dafny file being created and the contract class skeleton being constructed. After it acquires all the contract's

functions and variables using pattern detection in Java Regex, it places them into an Array-List Java data-structure so that they can be written into the skeleton. The parser utilizes the class named **DafnyWriter** to output the Dafny class. The **DafnyWriter** is responsible for creating the Dafny skeleton in a specified directory always as to add the extra implementation need for direct verification of the model.

When all the functions and variables are fully extracted, the **DafnyWriter** creates a file with the Dafny extension in a specified directory, then after the contract file has been created, the main class invokes the dafnySkeleton function passing in the variables and functions after they have been transformed to Dafny so they can be written into the contract class. Listing 3.11 depicts the dafnySkeleton mentioned function that uses the Java **FileWriter** library to output the Dafny Skeleton contract. However, the function also injects the recommended design pattern and best practices for the class and functions and extends the template.

```
public class DafnyWriter {

    private File outputFile;
    private String contractName;
    public DafnyWriter(String contractName){
        this.contractName = contractName;
        this.outputFile = new File("C:\\Users\\"+contractName+".dfy");
    }
    public void dafnySkeleton(ArrayList<String> variablesUnfilered,
        ArrayList<String> finalFunctions) throws IOException{
        /** Create the Dafny Contract Skeleton */
    }
}
```

Listing 3.11: Dafny Writer Class

The dafnySkeleton skeleton function works as follows. To the output file specified, it initially imports the template and extends it to the newly created class holding the contract's name. Then it injects the contract with and Element Array and the Repr in Dafny used to model data-structure as recommended in Dafny along with the other variables declarations. Below the inside the class below the variables declaration it injects a standard idiom for the class invariant mention previously. For every function, while declaring it, it also specifies the specification best practice. This best practice involves specifying the invariant in the pre and post condition while including the modification of the Representation set and the fresh(Repr-old(Repr)) to specify any new modification of the frame.

```
    for(int i = 0 ; i < finalFunctions.size(); i++){
        dafnyWriter.write("\n\t"+finalFunctions.get(i)+"\n");
        dafnyWriter.write("\t\trequires Valid()\n");
        dafnyWriter.write("\t\tmodifies Repr, this\n");
```

```
        dafnyWriter.write("\t\tensures Valid() && fresh(Repr-old(Repr))\n");
    }
```

Listing 3.12: Function Specification Design Pattern

The listing above, 3.12, demonstrates how the Java code writes Smart Contract's function in Dafny. The loop represented the specification that is implemented in the Dafny class skeleton. The code represents the data structure that holds the function after extracting them using Regular Expression. The code loops over the functions implement the requires, modifies, and ensures specification of Dafny. The specification includes the class invariant for coherence and the Repr modification standard idiom to reason about the contract's state.

Chapter 4

Results

This thesis's main topic is to introduce the Java parser and specification model that allows the reasoning and formal verification of Smart Contracts in the Dafny language. The significance, literature review, and methodology are discussed in previous chapters. This Chapter includes the result and evaluation of using the template and the parser.

4.1 Introduction

The thesis introduced a parser that is capable of handling all the Dafny specification model implementation using a specification template which attempts to supply the Dafny language with the missing name-space of the Solidity and Ethereum environment. The parser was able to correctly covert Solidity and output a skeleton of the Smart Contracts into the Dafny class while extending the name-space. However, there was been some errors and challenges that have occurred. Also, while the Dafny template provides the popular and mostly used global functions and variables, the global name-space of Ethereum is far more complicated than what the template support. Over the years, and due to many security considerations, the global name-space and the Ethereum environment has been modified and updated. Mainly to prevent many vulnerabilities previously explored and for implementing security consideration. While our name-space can detect many of the vulnerabilities and logical bugs related to the name-space, many other vulnerabilities require other formal verification techniques and testing to detect. This chapter will cover the parser and the specification model's limitations and strengths. Besides, the chapter will also evaluate the success of the specification model against Solidity's security consideration [28]. The Chapter also covers the use of the whole framework in the process of reasoning about Smart Contracts.

4.2 Evaluation

The specification model template built was able to detect logical vulnerabilities found in well-known contracts as well to prove the correctness of vulnerability free contracts that were fully reasoned.

While the specification model doesn't completely emulate the Ethereum Environment, it servers as a sufficient specification model for many contracts. Also, implementing any new functions included in the name-space in the specification model is straight forward. Assuming the developers attempting to verify contracts are aware of the security consideration of Solidity [28], any Solidity function can be seamlessly added to the global name-space and verified then used to verify contracts.

The parser was able to output an applicable skeleton contract for various contracts, its capabilities are limited. The parser was designed solely to demonstrate the research done regarding the implementation of Dafny and Solidity design patterns when modeling contracts. While regular expressions are fast, simple, and flexible, they are hard to debug and may produce many errors and typos in the parsing process [45]. Also, many regex implementations must be in place to accommodate all Solidity Syntax. However, the parser doesn't cover all Solidity syntax. For example, the parser doesn't cover enumeration and structs. Also, some code discrepancies might occur while parsing modifiers and functions into Dafny. Such discrepancies include when there are a variable and a method with the same name and using modifiers in the method declaration.

4.2.1 Security Consideration

In the Solidity language documentation, there exists a dedicated page that outlines the most common pitfalls faced. It's to provide the developer with the recommended best security recommendation. The page also recommends the use of formal verification to model the correctness of contracts mentions the best techniques.

While the template cannot verify against all the pitfalls, it can still be beneficial in some aspects. Firstly, the models weren't designed to reason against the gas limit due to not including it in the specification model. However, Dafny features, such as its built-in termination metric can assist significantly. Also, Dafny doesn't assume any call-stack depth, however, any clever implementation of it can be considered in future work. Moreover, some vulnerabilities cannot be modeled in formal verification. However, using the template, Dafny was able to model the Checks-Effects-Interactions design pattern for the re-entrance vulnerability.

```
include "./contract.dfy"
class Fund extends address {
  var shares: map<address, nat>

  method withdraw(msg: Message) returns (r: bool)
    requires msg.sender in shares
    requires this ≠ msg.sender
    requires ∀ i • i in shares.Keys ⟹ shares[i] ≤ this.balance
    modifies this, msg.sender
  {
      var share := shares[msg.sender];
      shares := shares[msg.sender := 0];
      r := this.send(msg.sender, share);
  }
}
```

Listing 4.1: Reentrancy vulnerability modeled in Dafny

Listing 4.1 depict the design pattern for avoiding the re-entrancy vulnerabilities in Dafny. Moreover, the model can also protect against overflows and underflow. If necessary that Dafny language accepts the introduction of a newly defined datatype. This datatype can be used to define more Solidity datatype such as signed and unsigned integers of different ranges. Dafny supports module, such datatype introduced can be wrapped in such models and injected into the necessary contract. Such data can be of the form shown in listing 4.2, taken from [46], and can be then specified in the pre and post-condition to reason against overflows.

```
module SolidityTypes {newtype int64 = i: int | −0x8000000000000000 ≤i < 0x8000000000000000}
```

Listing 4.2: New solidity Variable in Dafny

As mentioned in the security recommendation documentation [28], there is only so much formal verification can provide. Formal verification is only concerned with match the behavior of the code to the intended specification. However, it is up to the developer to accurately outline the behavior and be well aware of the security pitfalls of the language beforehand.

4.2.2 Auction

Smart Contracts were designs to facilitate the smooth interaction between parties in a transparent manner. One of the ideas that can be implemented in Smart Contracts is an Auction Platform [47]. The Auction Contract found in the Solidity Documentation was modeled in Dafny using the specification model.

The contract attempts to mimic real auction by accepting Ether from users as bids while maintaining the highest bidder and a list of the address of unsuccessful bidders. when the contract expires, the money in the contract is returned to unsuccessful bidders.

```
predicate Invariant ()
  reads this, Repr
{
    this in this.Repr ∧
    Valid() ∧
    (∀ i • i in pendingReturns ⟹ this.balance ≥pendingReturns[i]) ∧ this.balance ≥ highestBid
}
```

Listing 4.3: Simple Auction in Dafny

As previously mentioned, one of the recommended best practices in Dafny when working with objects and classes is implementing a class invariant. Since the class invariant must be satisfied at all times, conditions related to the coherence of the entire class is specified in it. Dafny is an expressive language, it allows the use of conditional quantifiers to express specification in the form of theorems. In this case, one of the conditions specified relates to the balance. The condition states that balance should always be greater or equal than the highest bid which is logically valid. However, in this code, 4.3,

Dafny detects a vulnerability in the form of the class invariant might not hold. This vulnerability was caused since the withdrawal function written in Dafny correctly updates the highest bid while updating the balance. Such logical vulnerability could have been catastrophic in real-world scenarios. Such an example illustrates the potential and power of model Smart Contracts in Dafny.

Furthermore, the template was able to model more contracts in Dafny. It was able to prove the correctness of the Lottery contract that relies on the block and a hash function to randomly select the winner. However, we were able to reason about the code using the block specification while the hash was implemented in for of a function with the specification since the Dafny language doesn't support such features.

4.3 Parser

The parser was able to convert simple contracts to Dafny, However, the parser is far from perfect and still needs to implement many of the Solidity code translation using regular expressions. However, since the parser only means to implement Solidity code specification in Dafny, it performs as expected and worked as intended on simple code that encompasses all the regex expression specified.

The parser was build for the purpose of implementing the recommend Dafny features and standard idiom so that developer can have a ready template to choose what is needed in the verification. However, the parser still contains many limitations that can be addressed in future works. The way it's designed, it is unable to parser files with more than one contract. As mention previously it also only works of specific solidity version contract that is not adept at handling all the breaking changes introduced to solidity over the years. Also, many regular expression that detects advance data structures such as struct and enum are unavailable in the parser, and they will be some errors in translations. Other than such limitations, contracts that are independent of such implementations can be successfully parsed.

Chapter 5

Discussion and Related Work

The final chapter of this thesis will discuss the significance of the work done along with similar work in this domain.

Blockchain technology is still in a very young stage in terms of development and research, and it is yet to be expanded and modeled to many different industries since it is essentially a new architecture in the field of distributed systems. Consequent to Bitcoin, the Etheruem platform came as a platform that can facilitate the modeling of many business applications in the form of a Smart Contract. However, one of the main properties of Smart Contracts, that enables the cooperation of different entities, is the direct relation between open code and financial assets. Such coupling makes financial assets vulnerable to logical exploitation. Developing secure contracts is one of the most important considerations for developers when designing them. However, Smart Contracts are different from regular applications in which the behavior doesn't depend on a single compiler but the execution environment that is devised from thousands on the node that behaves non-deterministically. Normal debugging and testing techniques can be challenging to implement on Smart Contract given such an environment. Nevertheless, considering the potential of such technology, many tools and frameworks have been constructed to model and testing Smart Contracts against vulnerabilities and logical bugs. One of the most used tools for vulnerabilities assessment and code correctness verification is formal verification since it is independent of the nondeterministic execution environment of Smart Contracts and attempts to verify the code's correctness statically [20] by implementing a specification model. Much work has been done by the research community on techniques, tools, and specification models to model Smart Contracts. The work done in this thesis follows from such research to build upon the GitHub Repository [9] to supply effective modeling of Smart Contracts in Dafny.

The significance of such work relies on the specification model implemented to adequately model Smart Contracts in Dafny and by providing a parser that implements the recommended design patterns. With such a model, effective verification of Smart Contract's logical vulnerabilities can be performed which can minimize vulnerability and secure contracts. Formal verification, in general, is a difficult field that involves many challenging aspects to deal with such as implementing accurate models to verify systems and reason about them. While mimicking the execution environment for testing Smart

Contract, implementing a specification model for effective formal verification can also be challenging depending on the behavior to be verified. Smart Contracts are implemented on complex distributed systems that are relatively new in their behavior and properties. Such uniqueness implies that modeling Smart Contracts using formal verification tools and techniques can be challenging. The importance of such work is related to implementing a framework to can significantly remove the implementation load of the verification by implementing a proven specification model that accurately models the smart contract's environment in Dafny. Such a model and parser can remove many tedious implementations required by the developer so that direct verification can be possible using Dafny.

The Dafny language is a relatively new language designed to the reason about programs. When it comes to blockchain, the language is being used by many researchers to secure different aspects, layers, and protocols of it. The language has been used by many in the research community for reasoning about and verifying code. The first work is related to using Dafny to prove protocols that are will be implemented in the new version of the Ethereum platform [46]. It is being used to verify new chains on the platform such as the beacon chain that will be in the Eth2.0 specification. The second is related to using Dafny to prove the new Proof-of-Stake consensus algorithm [48] that is going to be apart of Eth2.0 specifications. The work was related to proving the correctness of a Smart Contract in Dafny named the Deposit Contract [49] that attempts to use dynamic programming to implement a Merkle tree algorithm that will host all the validators in the consensus algorithm. One of the most interesting parts of this works is that it takes you through the various assumption and specification models implemented to attempt to model the algorithm in Dafny.

Chapter 6

Conclusion

Smart Contracts and the Ethereum platform have been gaining a lot of traction recently due to the shift in the industry to model business accordingly. Smart Contracts are changing conventional cooperation and applications. However, Smart Contracts are strongly tied to the real financial asset so rigorous testing is mandatory. However, the process of testing and modeling Smart Contracts is difficult since modeling the correct specification for formal verification can involve many extra implementations. Also, modeling the execution environment can be difficult due to the non-deterministic and distributed nature of the blockchain environment. Consequently, formal verification techniques are being devised since they don't rely on the execution environment adding the extra implementation necessary for effective verification such as the specification model and design pattern can also be challenging. With that being said, in the thesis, we built upon the work of [9] to deliver a specification model and a parser that enables the modeling of the Smart contract using a formal verification expressive language that involves many helpful features that assist significantly in the process of reason about codes correctness that is implemented by the parser.

While such implementation can suffice for adequate verification, many adjustments, improvements, and plugins can be added in future work. One of the considerations for future work is to implement a more inclusive specification model that can in cooperate more verified functions and variables included in the global name-space of the EVM. Also, clever implementation of the concept of gas can be added to the framework to enable the developer to reason about is the gas computation in contracts. On the parser side, the parser is implemented using a simple regular expression that doesn't include the parsing of all the syntax of Solidity and the body, however, the addition more regular expression and the effective error consideration can render the parser very reliable. Moreover, the parser can be replaced by a compiler or interpreter that attempts to translate the contract of the bytecode level while implementing Dafny recommend best practices, features, and standard idioms. Also, more Dafny design patterns can be implemented using such enhanced parser.

Bibliography

[1] O. Jarkas, Formal verification of smart contracts (23/10/2020 2020).

[2] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system.
URL www.bitcoin.org

[3] J. B. Arvind Narayanan, Bitcoin and cryptocurrency technologies: A comprehensive introduction, Princeton University Press.

[4] J. J. Amulya Gurtu, Potential of blockchain technology in supply chain management: a literature review, International Journal of Physical Distribution Logistics Management 49 (2019).

[5] B. L. James Schneider, Alexander Blostein, Profiles in innovation blockchain: Putting theory into practice, The Goldman Sachs Group, Inc. (2016).

[6] M. W. Zdun, Uwe, Smart contracts: Security patterns in the ethereum ecosystem and solidity, 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)doi:10.1109/IWBOSE.2018.8327565.

[7] A. D. Reza M. Parizi, Smart contract programming languages on blockchains: An empirical evaluation of usability and security, International Conference on Blockchain (2018) 75–91doi:https://doi.org/10.1007/978-3-319-94478-4_6.

[8] L. P. Massimo Bartoletti, An empirical analysis of smart contracts: Platforms, applications, and design patterns, International Conference on Financial Cryptography and Data Security 10323.

[9] Chthonic7, Solidity-to-dafny (2018).

[10] D. Chaum, Blind signatures for untraceable payments, Advances in Cryptology, Springer US, pp. 199–203.

[11] B. Lim, S. Kurnia, Exploring the reasons for a failure of electronic payment systems: A case study of an australian company, Journal of Research and Practice in Information Technology - ACJ 39 (11 2007).

[12] V. Buterin, A next generation smart contract decentralized application platform, Ethereum White Paper (2014).

[13] H. Chen, M. Pendleton, L. Njilla, S. Xu, A survey on ethereum systems security: Vulnerabilities, attacks and defenses (08 2019).

[14] A. Tyurin, I. Tyuluandin, V. Maltsev, I. Kirilenko, D. Berezun, Overview of the languages for safe smart contract programming, Proceedings of the Institute for System Programming of the RAS 31 (2019) 157–176. doi:10.15514/ISPRAS-2019-31(3)-13.

[15] S. Haber, W. S. Stornetta, How to time-stamp a digital document, Journal of Cryptology 3 (2) (1991) 99–111. doi:10.1007/BF00196791.
URL https://doi.org/10.1007/BF00196791

[16] K. Christidis, M. Devetsikiotis, Blockchains and smart contracts for the internet of things, IEEE Access 4 (2016) 2292–2303. doi:10.1109/ACCESS.2016.2566339.

[17] T. Kropf, Introduction to formal hardware verification, Springer Science & Business Media, 2013.

[18] M. S. Nawaz, M. Malik, Y. Li, M. Sun, M. Lali, A survey on theorem provers in formal methods, arXiv preprint arXiv:1912.03028 (2019).

[19] E. Seligman, T. Schubert, M. V. A. K. Kumar, Chapter 2 - Basic formal verification algorithms, Morgan Kaufmann, Boston, 2015, pp. 23–47. doi:https://doi.org/10.1016/B978-0-12-800727-3.00002-2.
URL http://www.sciencedirect.com/science/article/pii/B9780128007273000022

[20] [link].
URL https://www.guru99.com/static-dynamic-testing.html

[21] A. Singh, R. M. Parizi, Q. Zhang, K.-K. R. Choo, A. Dehghantanha, Blockchain smart contracts formalization: Approaches and challenges to address vulnerabilities, Computers Security 88 (2020) 101654. doi:https://doi.org/10.1016/j.cose.2019.101654.
URL http://www.sciencedirect.com/science/article/pii/S0167404818310927

[22] X. X. L. Z. H. Y. Yue Liu, Qinghua Lu, Applying design patterns in smart contracts, International Conference on Blockchain 10974 92–106.

[23] A. Mense, M. Flatscher, Security vulnerabilities in ethereum smart contracts, iiWAS2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 375–380. doi:10.1145/3282373.3282419.
URL https://doi.org/10.1145/3282373.3282419

[24] S. Dutta, How ethereum reversed a 50 million dao attack!, gitconnected.
URL https://levelup.gitconnected.com/how-ethereum-reversed-a-50-million-dao-attack-

[25] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (sok), Principles of Security and Trust, Springer Berlin Heidelberg, pp. 164–186.

[26] A. Dika, M. Nowostawski, Security vulnerabilities in ethereum smart contracts, 2018, pp. 955–962. doi:10.1109/Cybermatics_2018.2018.00182.

[27] P. Daian, Analysis of the dao exploit, Hacking, Distributed.
URL https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[28] SolidityDocs, Security considerations (2020).
URL https://solidity.readthedocs.io/en/latest/security-considerations.html

[29] P. Praitheeshan, L. Pan, J. Yu, J. Liu, R. Doss, Security analysis methods on ethereum smart contract vulnerabilities: a survey, arXiv preprint arXiv:1908.08605 (2019).

[30] T. C. Le, L. Xu, L. Chen, W. Shi, Proving conditional termination for smart contracts, in: Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts, BCC '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 57–59. doi:10.1145/3205230.3205239.
URL https://doi.org/10.1145/3205230.3205239

[31] S. Z.-B. Antoine Delignat-Lavaud, Formal verification of smart contracts: Short paperdoi:10.1145/2993600.2993611.

[32] K. Delmolino, M. Arnett, A. Kosba, A. Miller, E. Shi, Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab, Vol. 9604, 2016, pp. 79–94. doi:10.1007/978-3-662-53357-4_6.

[33] T. Abdellatif, K.-L. Brousmiche, Formal verification of smart contracts based on users and blockchain behaviors models, 2018, pp. 1–5. doi:10.1109/NTMS.2018.8328737.

[34] D. Park, Y. Zhang, M. Saxena, P. Daian, G. Roşu, A formal verification tool for ethereum vm bytecode, 2018, pp. 912–915. doi:10.1145/3236024.3264591.

[35] Y. Hirai, Defining the ethereum virtual machine for interactive theorem provers, Financial Cryptography and Data Security, Springer International Publishing, pp. 520–535.

[36] M. B. Sidney Amani, Myriam Bégel, Towards verifying ethereum smart contract bytecode in isabelle/hol, 7th ACM SIGPLAN International Conferencedoi:10.1145/3176245.3167084.

[37] K. Rustan, Specification and verification of object-oriented software, Microsoft Research, Redmond, WA, USA (2008).

[38] K. R. M. Leino, Dafny: An automatic program verifier for functional correctness, Microsoft Research.

[39] R. Leino, Getting started with dafny: A guide, Microsoft Research (2020).

[40] I. T. Kassios, Dynamic frames: Support for framing, dependencies and sharing without restrictions, FM 2006: Formal Methods, Springer Berlin Heidelberg, pp. 268–283.

[41] L. Herbert, K. R. M. Leino, J. Quaresma, Using dafny, an automatic program verifier, in: LASER Summer School on Software Engineering, Springer, 2011, pp. 156–181.

[42] SolidityDocs, Security types (2020).
URL https://solidity.readthedocs.io/en/v0.5.3/types.html

[43] (2016-2018). [link].
URL https://solidity.readthedocs.io/en/v0.5.0/050-breaking-changes.html

[44] [link].
URL https://www.w3schools.com/java/java_regex.asp

[45] (2014). [link].
URL https://www.slideshare.net/niekschmoller/regex-external

[46] (2020). [link].
URL https://github.com/ConsenSys/eth2.0-dafny

[47] A. Contract, Security types (2020).
URL https://solidity.readthedocs.io/en/v0.5.11/solidity-by-example.html

[48] (2020). [link].
URL https://github.com/ConsenSys/deposit-sc-dafny

[49] D. Park, Formal verification of ethereum 2.0 deposit contract (part i), Report (2019).
URL https://runtimeverification.com/blog/formal-verification-of-ethereum-2-0-deposi

Appendix A

Appendix

A.1 Parser Code

Parser - Main Class

```
/**
* @author Omar Jarkas 45981799
*/
/** Packages */
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Stack;
/**
 * Class Parser
 * This class contains the main functions the control the parser
 */
public class Parser {
    public static void main(String[] args) {
        /** Variable to hold the contract's name */
        String contractName;
        /** Variable to hold the contract's version */
        String contractVersion;
        /** Data Structure to hold the map of variables name, values, and types */
        HashMap<String, String> varaiblesMap = new HashMap<String, String>();
        try {
            /** Input the Contrat name here */
            String contractNameInput = args[0];
```

```
/** Class for extracting text from the contract file */
FileParser fileParser = new FileParser();
/** Class for checking the syntax of the Contract File using regex */
SyntaxChecker syntaxChecker = new SyntaxChecker();
/** First step is to remove all comment as not to interfer with other
    patterns */
File contractFileCommented = new
    File("C:\\Users\\User\\Desktop\\UQ\\thesis\\
                                ConToBeProv\\"+contractNameInput+".sol");
/** Output commentless file */
File contractFile =
    fileParser.removeCommentsAndCreateTestFile(contractFileCommented);
/** Class to Manipulate Strings */
StringMan stringMan = new StringMan();
/** Getting contract's verision and placing it in variable */
contractVersion = fileParser.getContractVersion(contractFile);
/** Syntax and compatibility check */
if(syntaxChecker.checkContractVersion(contractVersion)){
    System.out.println("Version "+contractVersion);
} else {
    System.out.println("Error: Version incompatible against parser!");
    System.exit(1);
}
/** Getting Contract's name */
contractName = fileParser.getContractNameLine(contractFile);
/** Syntax check of Contract Name*/
contractName = stringMan.getContractNameFromLine(contractName);
if(syntaxChecker.checkContractName(contractName)){
    System.out.println(contractName);
}
/** Class for constructing the skeletion */
DafnyWriter dafnyWriter = new DafnyWriter(contractName);
/** Using the File Parser and StringMan Classes to
 * Parse and detect all the function declaration of variables and
     placing them in data strucutres*/
Stack<String> functions = fileParser.getAllFunctions(contractFile);
Stack<String> variablesUnFiltered =
    fileParser.getVaraibles(contractFile);
ArrayList<String> organizedVariables =
    stringMan.getOrganizedVariables(variablesUnFiltered);=
```

```
            ArrayList<String> filteredFunctions =
                stringMan.organizedFunctions(functions);
            ArrayList<String> organizedFunctions =
                stringMan.changeParameters(filteredFunctions);
            /** Output the contract after passing the variables and functions to be
                written*/
            dafnyWriter.dafnySkelton(organizedVariables, organizedFunctions);
        } catch (Exception e) {
            //TODO: handle exception
            System.out.println("An Error has occured.");
            e.printStackTrace();
        }
    }
}
```

File Parser

```
/**
* @author Omar Jarkas 45981799
/** Packages */
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;
import java.io.FileWriter;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.Stack;
import java.util.ArrayList;
/**
 * Class FileParser
 * This class contains all the functions for parsering data using regex
 */
class FileParser {
   /**
    * Getting the contract version
    * @param contractFile Contract Name
    */
    public static String getContractVersion(File contractFile) throws
        FileNotFoundException{
        Scanner contract = new Scanner(contractFile);
```

```
        Pattern versionPattern = Pattern.compile("\\bpragma solidity
            (\\p{Punct}\\d{1,4}.\\d{1,4}.22;)");
        while(contract.hasNextLine()){
            String line = contract.nextLine();
            Matcher versionCheck = versionPattern.matcher(line);
            if(versionCheck.find()){
                return versionCheck.group();
            }
        }
        return "Contract Version is either not found in incompatible with the
            parser!";
    }
    /**
     * Getting the contract functions
     * @param contractFile Contract Name
     */
    public static Stack<String> getAllFunctions(File contractFile) throws
        FileNotFoundException {
        Scanner contract = new Scanner(contractFile);
        Pattern versionPattern =
            Pattern.compile("(?:(\\w+(?:\\.\\w+)*)\\s*=\\s*)?(constructor|function|modifier)\\s
        Stack<String> functions = new Stack<>();
        while(contract.hasNextLine()){
            String line = contract.nextLine();
            Matcher versionCheck = versionPattern.matcher(line);
            if(versionCheck.find()){
                functions.push(line.trim());
            }
        }
        return functions;
    }
    /**
     * Getting the contract name
     * @param contractFile Contract Name
     */
    public static String getContractNameLine(File contractFile) throws
        FileNotFoundException {
        Scanner contract = new Scanner(contractFile);
        Pattern versionPattern =
            Pattern.compile("(?:(\\w+(?:\\.\\w+)*)\\s*=\\s*)?(contract)\\s*(.*?)\\w\\s*(.*?)\\{
        while(contract.hasNextLine()){
```

```
            String line = contract.nextLine();
            Matcher versionCheck = versionPattern.matcher(line);
            if(versionCheck.find()){
                return versionCheck.group();
            }
        }
        return "Contract Name not found!!!!";
    }
    /**
     * Getting the contract varaibles
     * @param contractFile Contract Name
     */
    public static Stack<String> getVaraibles(File contractFile) throws
        FileNotFoundException {
        Scanner contract = new Scanner(contractFile);
        Pattern versionPattern =
            Pattern.compile("(?=address|uint|mapping|bool).*\\s+[\\w+\\s+]+;$");
        Stack<String> variables = new Stack<>();
        while(contract.hasNextLine()){
            String line = contract.nextLine();
            Matcher versionCheck = versionPattern.matcher(line);
            if(versionCheck.find()){
                line = line.trim();
                variables.push(line);
            }
        }
        return variables;
    }
    /**
     * Removing all comments
     * @param contractFile Contract Name
     */
    public static File removeCommentsAndCreateTestFile(File contractFile) throws
        IOException {
        Scanner contract = new Scanner(contractFile);
        File myObj = new
            File("C:\\Users\\User\\Desktop\\UQ\\thesis\\testing\\Lottery.test");
        FileWriter myWriter = new FileWriter(myObj);
        if (myObj.createNewFile()) {
            System.out.println("File created: " + myObj.getName());
          } else {
```

```
            System.out.println("File already exists.");
          }
        while(contract.hasNextLine()){
            String line =
                contract.nextLine().replaceAll("(?:/\\*(?:[^*]|(?:\\*+[^*/]))*\\*+/)|(?://.*)","
            myWriter.write(line+"\n");
        }
        myWriter.close();
        return myObj;
    }
}
```

Dafny Writer

```
/**
* @author Omar Jarkas 45981799
/** Packages */
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.FileWriter;
import java.util.Scanner;
import java.util.ArrayList;
/**
 * Class FileParser
 * This class contains all the functions for parsering data using regex
 */
public class DafnyWriter {
    /** Path of output file */
    private File outputFile;
    /** Contract Name */
    private String contractName;
   /**
    * Constructor
    * @param contractFile Contract Name
    */
    public DafnyWriter(String contractName){
        this.contractName = contractName;
        this.outputFile = new
            File("C:\\Users\\User\\Desktop\\UQ\\thesis\\testing\\"+contractName+".dfy");
    }
```

```
/**
 * Outputing skeletion
 * @param variablesUnfilered Varibles to be translated on written
 * @param finalFunctions Functions to be translated on written
 */
public void dafnySkelton(ArrayList<String> variablesUnfilered,
    ArrayList<String> finalFunctions) throws IOException{
    FileWriter dafnyWriter = new FileWriter(this.outputFile);
    dafnyWriter.write("include \"./contract.dfy\"\n\n\n\n");
    dafnyWriter.write("class "+contractName+" extends address {\n\n");
    ArrayList<String> variableNames = getVaraibleNames(variablesUnfilered);
    if(variableNames.size() > 0){
        dafnyWriter.write("\n\tghost var Elements: seq<address>\n");
    }
    for(int i =0 ; i < variableNames.size(); i++){
        String temp = "";
        if(variablesUnfilered.get(i).contains("array")){
            String[] parts = variablesUnfilered.get(i).split(" ");
            String varType = parts[1];
            temp = "array<"+varType+">";
        } else if (variablesUnfilered.get(i).contains("map")){
            String[] parts = variablesUnfilered.get(i).split(" ");
            String varType1 = parts[2];
            String varType2 = parts[3];
            temp = "map<"+varType1+","+varType2+">";
        }else {
            String[] parts = variablesUnfilered.get(i).split(" ");
            String varType1 = parts[0];
            temp = varType1;
        }
        dafnyWriter.write("\n\tvar "+variableNames.get(i)+":
            "+temp.replace("uint","int").replace("uint256","int")+"\n");
    }
    dafnyWriter.write("\n\tghost var Repr1: set<object>\n");
    dafnyWriter.write("\n\tpredicate Invariant()\n");
    dafnyWriter.write("\t\treads this, Repr1\n");
    dafnyWriter.write("\t\tensures Invariant() ==> Valid() && this in
        this.Repr1");
    dafnyWriter.write("\t{\n");
    dafnyWriter.write("\t\tthis in this.Repr1 && this.msg in this.Repr1 && \n");
```

```
        dafnyWriter.write("\t\tthis !in this.msg.Repr && block in Repr1 && this !in
            block.Repr\n");
        dafnyWriter.write("\t}\n");
        for(int i =0 ; i < finalFunctions.size(); i++){
            if(finalFunctions.get(i).contains("constructor")){
                dafnyWriter.write("\n\t"+finalFunctions.get(i).replace("uint","int")+"\n");
                /** Constructor's Post Condition */
                String constructorPostCon = "\t\tensures ";
                finalFunctions.remove(i);
            }
        }
        for(int i =0 ; i < finalFunctions.size(); i++){
            dafnyWriter.write("\n\t"+finalFunctions.get(i));
            dafnyWriter.write("\t\trequires Invariant()\n");
            dafnyWriter.write("\t\tmodifies Repr, this\n");
            dafnyWriter.write("\t\tensures Invariant() && fresh(Repr-old(Repr))\n");
        }
        dafnyWriter.write("}");
        dafnyWriter.close();
    }
    public ArrayList<String> getVaraibleNames(ArrayList<String> variablesUnfilered){
        ArrayList<String> variableNames = new ArrayList<String>();
        for(int i =0 ; i < variablesUnfilered.size(); i++){
            String[] parts = variablesUnfilered.get(i).split(" ");
            String lastWord = parts[parts.length - 1];
            variableNames.add(lastWord);
        }
        return variableNames;
    }
}
/** Create the Dafny Contract Skeleton */
```

A.2 Dafny Code

```
/**
* @author  Omar Jarkas 45981799
* NOTE : This work was built of a github  repository  https: //github.com/chthonic7/solidity-to-dafny
*  Trait  Address
* This  class  is  used implement the  specification  model in Danfy
*/
    trait address{
        /** Balance global  variable  of any contract */
```

```
        var balance: nat
        /** User information */
        var msg: Message
        /** Block information */
        var block: Block
        /** Ghost Varaible Representation Set */
        ghost var Repr: set<object>
        /** Class Invariant Standard Idiom */
        predicate Valid()
            reads this, Repr
            ensures Valid() ⟹ this in Repr
            {
                this in Repr ∧
                msg in Repr ∧ this ∉ msg.Repr ∧
                block in Repr ∧ this ∉ block.Repr
            }
        /**
        * Call this event method when transfering Ether
        * @param payer address for interactiong with user and contracts
        */
        method transfer(payer: address)
        requires payer.msg.value < payer.balance
        requires payer ≠ this
        requires Valid()
        modifies Repr, payer
        ensures balance = old(balance) + payer.msg.value
        ensures payer.balance = old(payer.balance) − payer.msg.value
        ensures Valid()
        {
            assert old(balance) = balance;
            balance := balance + payer.msg.value;
            assert old(balance) + payer.msg.value = balance;
            payer.balance := payer.balance − payer.msg.value;
            assert old(balance) + payer.msg.value = balance;
        }
        /**
        * Call this event method when transfering Ether
        * @param amount specify the amount to send
        * @param sender specify the user
        */
        method send(amount: nat, sender: address)
            requires sender.balance > amount
            requires Valid()
            modifies Repr
            ensures this.balance = old(this.balance) + amount
            {
                this.balance := this.balance + amount;
            }
    }
/**
* Message Class
* This class is used implement users attributed
*/
    trait Message{
        var sender: address
        var value: nat
```

```
        var data: nat
        ghost var Repr: set<object>
    }
/**
* Block Class
* This class is used implement users attributed
*/
    trait Block{
        var timestamp: nat
        var coinbase: address
        var difficulty : nat
        var gaslimit : nat
        var number: nat
        ghost var Repr: set<object>
    }
```

A journey of a thousand miles begins with a single step.

Lao-Tzu,
Tao Te Ching