

Provas de Coerência Transacional para Smart Contracts em Blockhains

David Alexandre Aparício Bugalho

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2^o ciclo de estudos)

Orientador: Professor Simão Patrício Melo de Sousa

10 de Outubro, 2022

Declaração de Integridade

Eu, David Alexandre Aparício Bugalho, que abaixo assino, estudante com o número de inscrição M11031, do Curso de Mestrado em Engenharia Informática da Faculdade de Engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o **Código de Integridades da Universidade da Beira Interior**.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assumindo assim na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 10 /10 /2022

(assinatura conforme Cartão de Cidadão ou preferencialmente
assinatura digital no documento original se naquele mesmo formato)

David Alexandre Aparício Bugalho

Acknowledgments

The writing of this document stamps the end of the Master's Degree in Computer Science at *Universidade da Beira Interior* (UBI), academic year of 2021/2022, and the end of one's own academic pilgrimage. This section is intended to reflect upon the author's academic journey up until this point — how and why it started, its process, crossing the finish line, and plans for the future — but also to make the readers more familiar with the author as well. Unlike what is going to be seen throughout the document, this section will be written in first person, and a more personal speech.

NOTE: This section went on for longer than expected. Along the way, the speech became more meaningful than for the reader's exposition only, it also became a freeing, raw narrative of one's memory recounting its academic experiences up until yet another life turning point, greatly helping in soothing the mind in a time of need. If the reader makes it to the end, apologies are in place for overtaking their time in something that is not entirely related to the dissertation itself, in turn, it is also with sincere gratitude and appreciation that one rewards them for reading.

Firstly, I would like to formally introduce myself. I am David Alexandre Aparício Bugalho, birthed in 1999, being 23 years old at the time of writing. I come from a modest home in a modest city, where I lived the most significant part of my lifespan with my parents, older brother, and a couple of house cats. I consider myself to have had a pretty typical childhood, despite the fair share of issues that I think any family might have had. Nonetheless, I am eternally grateful to my parents for allowing me the chance to get where I am right now — they are both hard workers that over commit their time and health so that I can have a shot at succeeding more in life than they have ever had. Such a parental gesture, that sometimes goes unnoticed and unrecognized by me, their son, is truly despicable and inconsiderable. For this, I can only hope they forgive me, as I am confident that such gesture will only ever be truly recognized if someday I, too, were to become one.

My parents' household was one of the first to have a computer in the neighborhood, though, I have no recollection of my parents ever really making use of it, maybe they had it as a gift for my older brother. That being said, my earliest memories are from me watching my brother on the computer, mainly playing computer games. Sitting in the side chair beside the one occupied by my brother, my gaze was constantly attentive to the old CRT monitor, in awe, of what my brother was doing. Naturally, as time went on, I slowly became accustomed to being the one sitting on the chair across the computer.

I leisurely spent my time from grade to high school without too much hassle. I was an average student, exerting just enough effort to feel comfortable with my grades — not being pressured by my parents for scoring low, nor scoring so high as not to stand out and having to uphold a higher standard. During this time, I partook in a variety of hobbies and activities, but nothing truly stuck. Likewise, I made friends with some colleagues, to whom I can call friends to this

day, however, once I got home I would always go right into my comfort in either the console or the computer. The immersion provided by these medium were unrivaled by any other activity, yet, looking back, there was true passion in it.

I was simply sailing through time, not challenging the current, just doing what felt natural or acting accordingly to the feedback or lack of thereof from my elders, or whomever I had respect for. Never truly felt passionate or had a dream, not for pursuing a career in some field nor pouring my heart and soul into one specific hobby. I just knew sailing as I was would get me through my education and activities that I had committed to, and that, at some point, the current would cease, requiring me to decide upon a path, to hop on some other current of my choice.

When the time came to do so, I, very expectedly, felt lost, unwilling to choose a path. These were times of great stress, as watching my peers setting their courses made it feel like I was being left behind. It made sense, since I was not interested in pursuing any particular career, however, even if I were not to have a will or peculiar interest of my own, I could borrow them from those closer to me. My older brother had at the time just graduated from the military academy, so enlisting there felt just as natural — hadn't needed to decide on my own, just had to follow my brother's footsteps. With the help of both my father and brother, an application for the entrance exam was sent.

I did not pass the entrance exam, returning to square one. More stressful times were ahead, but somehow managed to be pressured into taking action. At that point, the second round of college applications had opened, so now I sought help from my family. They were always attentive of me spending lengthy times at the computer, so their first suggestion was "*Engenharia Informática*" (Computer Science). At first, I was very reluctant — most of my time in the computer was dedicated to entertainment only. If this were my path, then the computer would likely become a source for discontentment, possibly ruining my enjoyment of ever being in front of it. Despite the thought, I did not exert my opinion, simply accepting the opinion, now a duty, that my family had given me. Several applications for these courses were sent, the very first being in the city of Covilhã.

The application got accepted. Soon after, I ventured with my mother into this unknown land, at least to us. For the first time in my life, something bizarre was truly happening — am I really going to leave the bird's nest? Living on my own? — I started to become entertaining with the thought of living alone, sort of starting life anew. For the first time in what felt like ever, an event was truly exhilarating. I was about to start from life zero (although still have parental financial support), an opportunity to possibly discover, rediscover, or even affirm myself as someone else. In that same day, all the paperwork was sorted, only required to find a suitable place to call home for the next years. After searching high and low, a great place was found that, even 5 years later, is still considered home (but not for too long now). I am grateful for this house's landlord, which surprisingly provided the home with more than the optimal living conditions, as well as the house colleagues, which made the place lively (but not obnoxious). If the conditions/circumstances for the current place not as good, most likely the dissertation would have not happened.

The first few weeks nervously exciting — being unrecognized by none, being welcomed by

tradition, and in the process getting to know the academia community. I had the chance to act, be, and do things that would otherwise be impossible had I not pursued this path. The sense of freedom that was felt truly played a big part in discovering my own person. It made sense, before coming to Covilhã, life truly felt like simply going through its motions, almost mindlessly following the same routine, resulting in experiences from that time becoming one big blend. That is why taking hold of my actions and decisions was so impactful — what to do, what to eat, when to study, when to sleep, not feeling remorseful to do things out of order — such responsibilities of life, that were once micromanaged by my parents, were now mine to uphold, to figure out.

During the first year, I got to know several people, and got the chance to become closer friends with some. Though I had friends during my school years, the type of camaraderie that was obtained through college is an entire different experience. Perhaps it was the union through shared hardship, perhaps it was the bonding experience through former mundane and trivial activities that I hadn't paid attention before, perhaps it was the simple fact that everyday I wouldn't come back to my hometown household. The more time I spent here, the more I saw my life being given meaning. Every activity that I had thought mundane became most joyful when with friends. For those whom I call close friends, each and every one of you has filled me with meaning and strength to keep pursuing and finishing this academic journey. Crossing paths with you all made me realize life was much more than I had previously known. Once my academic duties are finished, I hope that, one day, we can live and remember the moments and memories we so hold dear, toasting to our reunion.

By the start of the last undergraduate semester, the unknowingly looming threat of the COVID-19 pandemic started to approach. Academic life carried on for but a brief moment before the pandemic instilled at large. I remember quoting this to a friend:

“I wish there was a way to know you're in the good old days before you've actually left them.”

No one had any semblance of idea the impact the pandemic would have, and, at the same time, class and friends were mentally preparing for the eventual parting at the end of the semester. However, preparation time was blindly robbed. We hadn't known that farewells would come much sooner. Before the official lockdown, friends brushed off the occurrence as a short-term farewell — “see you later”.

It was not short-term. As of mid-March of 2020, the life I had held so dear became but a memory.

The lockdown prevented the camaraderie created over the years from being properly celebrated. The lockdown prevented students from properly saying their goodbyes before the departure for their next step in life. Some janky laptop camera could not compare to properly conveying interpersonal emotion. For the sake of protecting those closest to us, however tragic the situation might turn out, it left me with a gaping wound in the chest. It was too cruel an end to the academic life I had cherished so much.

The most safe thing to do was going back to my hometown. Staying through more than a weekend, I quickly remembered how mundane and lifeless of a life I had had before I came to pursue something my family had chosen in Covilhã. Online classes really demonstrated me how important the social aspect of a learning environment is — sharing experience, knowledge, worries, difficulties, reassurance — all contribute to a student’s wellbeing. Personally, I had a hard time coping with everything, off camera, off classes, but I withstood.

I could not afford to give up, being so close to cross the finish line. Letting my peers down would have only rubbed salt in the already open wound. Despite the cruel, unfortunate ending, the academic year terminated with great success for the entire class. Strangely, a sense of accomplishment was felt, combined with anxiety and stress — “Am I supposed to feel confident about my abilities as a computer scientist?”. That was not the case, in fact, I had convinced myself that, after the 3 years of academic preparation, I was not ready to accept a job offering. Therefore, I did what any fresh undergraduate would do — keep progressing in academic life — now trying to clear my uncertainties, attempting to prove myself that I am competent by going up a notch in education. In the same year I graduated, I also applied for the Master’s Degree in Computer Science at the city in which my life was put in motion, Covilhã.

It is the start of the 2020/2021 academic year. As I come back and reach my home, after half a year, profound sadness commences to well up inside me. I might have been in the same place just 6 months ago, however, it did not feel the same. The wound hadn’t fully mended, any path that I had walked on before quickly became filled with echoes of the past. The kind of joy and contentment felt during those first 3 years in the city are never coming back — it finally sunk, the wound ached, those that I had formed close bonds with were no longer present in my daily life. It was a hard reality to accept. I could in no way replicate the life I had before, since I was in a city that I knew, not welcomed by tradition, and amidst a hybrid regime of on-site and online classes. Academic life in Covilhã soon began to regress to a mundane, lifeless school cycle.

However, not all was lost — even amidst what could be considered rough times, I had the opportunity to become more acquainted and closer to other people. Though, maybe only as an attempt to fill the gap left by the first wound. Regardless, the aforementioned year was my most successful in terms of academic prowess. I confronted the subjects and challenges from a different perspective and motivation, I successfully detached myself from bad habits acquired in the past, becoming more savvy and independent, which in turn helped rebuild newfound confidence in my own expertise, abilities, and knowledge.

Fast forwarding to the last year of the Master’s program, the year is entirely focused on producing a dissertation. The first year went reasonably well, I felt accomplished in the grades that I was given, self-doubt started to clear, and I became adjusted in setting good practices when it came to the learning process, I felt ready to tackle a project to my liking. In past semesters, one issue I had with projects was the fact that most of the time multi-juggling projects was required. Personally, I had always preferred to laser-focus on the task I have directly ahead, which in turn may very negatively affect things that are not of immediate importance. However, with the dissertation in sight, I finally found an opportunity to pour both

heart and soul into it, something I hadn't set myself to do for any kind of prior project. Laser focus could be exploited for an entire year to my advantage. I was excited to get started.

Professor Simão Melo de Sousa has always been a stern, strict, but generous teacher. I failed, however, to notice that until the later stages of my academic journey. Not taking full advantage of his earliest classes will be something that I will always regret. The insight I could have had then would have been pivotal in building my self-confidence and overall knowledge. This year, it was time to amend my regrets — despite not completely feeling up to the challenge, I approached the professor that I admired, and sought out to make my dissertation under his tutelage — not only to quell my regrets, but also because this would become my final test. Working under the professor would require me to push myself beyond my present capabilities. Even if at the start I feared not being up to the task, that sentiment would soon become the stimuli for my growth. I want to extend my gratitude to professor Simão for always being so kind and understanding per my own issues and struggles. I am certain that I made the correct choice in working under the professor, as it indeed stimulated my growth and made me learn about myself and my habits. Not only that, but the professor has been keen and active on introducing his students to future job opportunities, which is much appreciated by newly graduates, including me. I will not forget the professor's generosity, and hope one day I can make him proud by my accomplishments in the professional world.

The topic for the dissertation was completely new to me. No class had ever introduced the concepts of this area, so while I had my concerns, I also saw an opportunity to, once again, start with a clean slate — devoting and testing the entire learning process that I had come into contact within the past year — now with the confidence that this project could take my full time budget. However, things would soon turn in an unexpected manner.

Although I could dedicate my all, to completely laser focus on my project regardless of time, started a procrastination habit. At the beginning, I would dedicate my entire day into researching this new topic. I quickly became burned out. Slowed down my pace and, at some point, I started to take time for granted. This was one of the issues I have been tackling with, even now, as I am writing this text. As time went on, I would go for days without ever making any decent progress, since, after all, I could progress whenever I felt like it. There are more variables contributing to my procrastination issue, but the main one has always been my lack of self-discipline regarding managing time as a resource. I did not respect my time, for the entirety of this work, progression has always been done in big spurts either focusing for long periods of time, or wait until that time comes, or until I could not wait any longer (precisely right now, as I still have one last section in the dissertation to complete). I thought I could use my willpower to throttle my spurts, but all I learned is that I am weak against my own mental restraints, though, I also learned that when up caught between a rock and a hard place, I am able to dedicate inhumane hours. Of course, this is unfeasible if being done for days on end. When confronted with a real job, though, I am confident that circumstances will be able to redirect this kind of inhumane dedication into a useful working schedule.

All in all, I plan on giving my all during the little time that I have left (compared to the time I had over the year). I will finish this dissertation, and I will be someone that I can be proud of for finishing despite the hardships faced. Maybe I find the act of procrastination enjoyable (I

seriously wrote this section when I still have one main section left), or I simply prefer to pour significant work under pressure. Perhaps, pressure simply acts as a motivational catalyst for writing. Nonetheless, I am ecstatic and looking forward to writing the last word of this document, no matter how little time I have left.

As for plans for the future, the first thing should be to reset the strain on my mental and rest. Mental health played a big role, in which I neglected, throughout my academic journey, being especially prevalent during this last year during the dissertation. As I am truly finishing the document now, I feel like I successfully accomplished what I had sought out to do when enrolling to the master's program — cleared my own doubts, leveled up my expertise, and acquired newfound confidence, while still being conscious of my own faculties and capabilities — I feel ready and confident to provide and benefit others with my abilities, and I want to do so in the near future.

If the person reading reached this point, thank you. I poured my thoughts, feelings, past grievances and present challenges and future thoughts directly onto this section. It felt liberating to write this entire section without following any sort of reference, moreover when the rest of the document has a different approach in language. That being said, I for last, extend my gratitude to the dear reader — for taking the time to better know and understand the person behind the paper.

Abstract

Blockchain technology is an emergent topic based on decentralization and immutability, enabling mutually untrusting parties to fairly exchange assets without the need of a central authority. Recently, the addition of blockchain programs, known as smart contracts, enabled the technology to expand upon a variety of industry sectors, already known to traditional software. Many organizations and corporates saw a growth opportunity, extending their businesses into this domain — now, though, with the blockchain twist. However, the inclusion of computation exposed a weak link in the overall blockchain security, due to carrying not only traditional software bugs, but also never before seen ones. That way, smart contracts, especially valuable ones, became enticing for hackers to exploit, which resulted in a set of tragedies where funds were stolen, among other consequences. Soon after, smart contract security became a most valuable topic of research among blockchain platforms. The Tezos blockchain is a relatively new platform whose stance values security by construct infrastructure, in consequence of the past incidents. While many smart contract security solutions were devised over the years, these have not been properly adapted nor adopted for the average developer in the community. Due to various reasons, but for one, seamless integration with the smart contract development processes is one of them. This dissertation approaches the blockchain security problem through an indirect approach, providing the developer with better accessibility and conditions for working on one of Tezos’s state-of-the-art security tools. Although it is unorthodox, it is hoped for the solution to inspire and appeal other blockchain communities by shedding some light in this unknown direction.

Keywords

blockchain, Tezos, smart contract, smart contract development, smart contract security, smart contract verification, formal verification

Resumo

A tecnologia blockchain é um tópico emergente baseado na descentralização e imutabilidade, permitindo que entidades desconhecidas e não confiáveis consigam trocar bens e valores digitais de forma justa sem necessitarem uma entidade central. Recentemente, a adição de programas na blockchain, designados de *smart contracts*, permitiu que tal se expandisse sobre uma variedade de sectores industriais já explorada por programas tradicionais. Contudo, muitas empresas viram uma oportunidade de negócio bastante lucrativa, estendendo o seu negócio para este ambiente, agora incutindo as regras da blockchain. Embora oportunidades lucrativas tenham aparecido, problemas relativos aos programas tradicionais, bem como outros novos ainda não descobertos, também. Os *smart contracts* revelaram-se como um elo mais fraco para a segurança da blockchain e, tendo estes a capacidade de reter bastante valor monetário, tornaram-se um alvo aliciante para *hackers*. Não muito depois, notícias espalharam-se pela *internet* a anunciar crimes por entidades anónimas — roubo e congelamento de fundos, entre outras consequências, na blockchain. Após o primeiro grande incidente, a segurança na blockchain começou a ser um tópico bastante estudado por peritos e investigadores das várias comunidades. A blockchain da Tezos é uma plataforma relativamente recente, com uma postura relativa à segurança bastante madura, resultado dos incidentes passados. Enquanto várias soluções foram alcançadas para a segurança de *smart contracts*, estas não seriam ainda bem incorporadas pela comunidade, ou pelo menos para o engenheiro de contratos comum. Existem várias razões, porém, acessibilidade nos vários aspetos das ferramentas de segurança é uma delas. O trabalho realizado por esta dissertação passa por solucionar este problema, mais especificamente, solucionar o problema para uma ferramenta de segurança de programas na blockchain da Tezos. Este tipo de solução não é comum na literatura, contudo, espera-se que o trabalho realizado sirva de inspiração para que as comunidades possa explorar esta vertente mais indireta de segurança na blockchain.

Palavras-chave

blockchain, Tezos, smart contract, smart contract development, smart contract security, smart contract verification, formal verification

Resumo alargado

O universo da blockchain é uma tecnologia emergente, cujos conceitos principais são a descentralização e imutabilidade. Em conjunto com outras garantias, é um ambiente onde bens podem ser comercializados entre utilizadores pseudoanónimos, sem a necessidade de uma entidade central intervir diretamente. Recentemente, a computação *Turing-complete* passou a integrar estes ambientes, habilitando certas blockchains a desempenharem funções outrora vistas em mercados já conhecidos pela engenharia tradicional, contudo, as soluções para estes mercados seriam agora abordadas com recurso às regalias da blockchain. Porém, deveria ser calculado que, ambas vantagens e desvantagens seriam integradas — sim, a programação na blockchain revelou bastantes oportunidades novas para empresas e negócios, mas também os perigos associados a estes programas. Assim, esta integração fez com ambas blockchain e programas herdassem propriedades uma da outra, especialmente a imutabilidade. *Smart contracts*, a como se designam estes programas, revelaram-se um dos elos mais fracos da blockchain e, com a possível quantidade de fundos que poderiam conter, tornaram-se um alvo aliciante para agentes maliciosos se aproveitarem — tal resultou em várias tragédias como furtos e congelamentos de valores, entre outras consequências. Desde o primeiro grande incidente, investigadores e especialistas em segurança das várias comunidades das blockchains, começaram a apostar de forma exigente em métodos, técnicas e ferramentas para elevarem a segurança nestes ambientes, com a esperança de que tais tragédias não se repetissem. A blockchain da Tezos é relativamente recente, contudo, a sua postura quanto à segurança revela-se madura, sendo a sua conceção fruto dos incidentes ocorridos, daí, ela aposta numa infraestrutura e programas que sejam mais seguros por construção. A segurança nos *smart contracts* é uma área a ser bastante investigada, contudo, ainda há pouca tração das soluções encontradas serem adotadas e integrados com o processo de desenvolvimento de *smart contracts*, pelo menos para o engenheiro de contratos comum. Isto é, existem algumas barreiras e inconveniências que fazem das soluções existentes difíceis de trabalhar com, como corolário, existem otimizações a serem feitas neste aspeto. Tendo em conta os pontos anteriores, o contributo desta dissertação vem na forma de otimizar a acessibilidade de uma das ferramentas de segurança para programas na blockchain da Tezos. A solução passa pelo desenvolver de uma extensão para um ambiente de desenvolvimento usado pelo desenvolvedor comum, contribuindo para a visibilidade, compatibilidade e outros benefícios para quem usufrua deste ambiente de desenvolvimento para implementar os seus contratos. Este tipo de abordagem não é comum na literatura, por isso espera-se que as várias comunidades das blockchains se possam inspirar neste método atípico para combater os problemas supracitados nas suas plataformas.

Contents

Contents	xv
List of Figures	xvii
List of Tables	xix
Listings	xxi
1 Introduction	1
1.1 Motivation	1
1.2 Problem Summary	2
1.3 Objectives	3
1.4 Document Outline	4
2 Technical Background	5
2.1 The Blockchain	5
2.2 The Smart Contract	8
2.3 Tezos	10
2.4 Conclusion	11
3 Literature Review	13
3.1 Smart Contract Vulnerabilities	13
3.2 Real World Exploits	16
3.3 State-of-the-Art Security Tools	17
3.3.1 Smart Contract Security Mechanisms	18
3.4 Discussion	19
3.5 Conclusion	20
4 Problem Statement and Work Plan	23
4.1 Problem Statement	23
4.1.1 Problem	23
4.1.2 Solution	25
4.2 Implementation Roadmap	27
4.3 Conclusion	27
5 Implementation	29
5.1 Tools & Technologies Used	29
5.1.1 Michelson	29
5.1.2 LIGO	31
5.1.3 Visual Studio Code API	33
5.1.4 WhylSon	34
5.2 Extension	36

5.2.1 Prelude	36
5.2.2 Design	37
5.2.3 Development Loop	45
5.2.4 Features & Functionality	46
5.2.5 Issues	48
5.3 Conclusion	49
6 Practicality Assessment	51
6.1 Intra-Michelson Value Check	51
6.2 Deductive Verification	53
6.3 Conclusion	54
7 Final Remarks	55
7.1 Summary	55
7.1.1 Remarks	55
7.2 Future Work	56
Bibliography	57

List of Figures

2.1	Generic block structure.	5
2.2	Blockchain architecture for smart contract enabled blockchain.	9
4.1	Verification loop for proposed solution.	26
5.1	A generic stack structure with elements and operations.	30
5.2	Stack Overflow 2022 survey statistics for “IDE” topic.	33
5.3	<i>WhylSon</i> using <i>why3-ide</i> for a verification session of a <i>Michelson</i> Program.	35
5.4	Design of <i>godbolt</i> online compiler and <i>Markdown</i> extension, respectively.	38
5.5	Plausible <i>LIGO</i> project using proposed solution.	40
5.6	Simple example of <i>LIGO-Michelson Dual-View</i> .	41
5.7	Dual-View showing a compilation error.	42
5.8	Snippets available with the extension.	43
5.9	Condensed extension workflow through activity diagram.	47

List of Tables

3.1 State-of-the-art verification tools found for Ethereum.	21
3.2 State-of-the-art verification tools found for Tezos.	22
4.1 Task roadmap for implementation of the solution proposed.	28

Listings

5.1	Example of a simple <i>Michelson</i> program.	30
5.2	Example of a simple <i>LIGO</i> program.	33
5.3	Example of a <i>WhyLson</i> generated <i>WhyML</i> file.	35
5.4	Structure of an entry in <code>contracts.json</code> file	40
5.5	<i>WhyLson</i> annotation snippet example.	44
5.6	<i>WhyLson</i> annotations in <i>Michelson</i> .	44
5.7	<i>WhyLson</i> annotation snippet for contract with pre and post conditions.	44
5.8	<i>WhyLson</i> annotations in <i>Michelson</i> with pre and post conditions.	44
6.1	Using <i>WhyLson</i> annotations for simple value checks	51
6.2	Using <i>WhyLson</i> annotations for deductive verification - factorial.	53
6.3	Using <i>WhyLson</i> annotations for deductive verification - sum list.	53

Acronyms

API	Application Programming Interface
CFG	Control Flow Graph
DPoS	Delegated Proof of Stake
DSL	Domain-Specific Language
EVM	Ethereum Virtual Machine
FRESCO	Formal Verification of Tezos Smart Contracts
FSM	Finite State Machine
IDE	Integrated Development Environment
IO	Input & Output
IL	Intermediary language
JSON	JavaScript Object Notation
LPoS	Liquid Proof of Stake
PoC	Proof of Capacity
PoS	Proof of Stake
PoW	Proof of Work
RELEASE	Reliable and Secure Computation Group
SHA	Secure Hash Algorithm
SMT	Satisfiability Modulo Theories
SWC	Smart Contract Weakness Classification
UBI	<i>Universidade da Beira Interior</i>
UI	User Interface

Chapter 1

Introduction

1.1 Motivation

The introduction of blockchain technology has generated great interest in a variety of market sectors, as well as keen-eyed individuals. Its sudden emergence and ever so increasingly popularity is due to the incredible potential for netting large sums of profit. As such, numerous corporations and individuals have seized this opportunity to create or expand their businesses into this environment through the deployment of software emulating their business into the network, denominated a **smart contract**. This business approach has seen an explosive growth since the Ethereum network, the blockchain that innovated this practice, came online in 2015.

As with any piece of software, bugs are a concern for any of the parties involved, all the more so in a blockchain environment — with immutability being one of its key principles, the act of deployment should not be done carelessly — for any problem that arises post-deployment cannot be easily patched without a myriad of ramifications being associated with it, most notably financial losses and impairment of the blockchain’s credibility.

To minimize the aforementioned result, smart contract security countermeasures have been devised, such as code auditing, static analysis and formal verification. Despite the existence of these safeguards, faulty smart contracts are still being detected and taken advantage of past the act of deployment. There is a strong appeal from smart contract security experts and the broader blockchain community to reinforce and maturely adopt these countermeasures into the smart contract development cycle. Alas, to do so is not a trivial matter, as these measures must weave naturally and with great effectiveness as to maximize adherence from the community.

The Ethereum platform is, by far, the most popular and top grossing blockchain that supports smart contracts. Its record numbers are both its greatest strength and weakness — marketplaces thriving in activity and wealth, as well as the harmful actor looking to financially bank on an appetizing exploit — it has become a den for these actors whose ill reasoning has tunnel visioned into the platform given its prestige.

However, there are numerous other blockchains that now enable Ethereum’s innovation, such as **Tezos**, which is particularly interesting given its stance on security — for one, its smart contract language was designed specifically with security in mind, purposefully easing formal verification processes on its code. Second, a mechanism of self-amendment that enables reasonably regular updates to the network protocol itself. The latter is a feature unique in the blockchain realm, providing the platform with exceptional adaptability when it comes to impending vulnerabilities that have been on the loose for too long, as an example.

Tezos' steady growth progression cannot be yet compared to Ethereum's staggering success but, given its adaptability, updates are improving the platform cycle by cycle. It is reasonable to anticipate that this platform will receive a large influx of users down the line, whether honest or malicious. As a result, it is prudent to plan for such a phenomenon beforehand, learning from previous incidents.

Advisor and Professor Simão Melo de Sousa is a senior and lead researcher on the Reliable and Secure Computation Group (**RELEASE**) laboratory, within the computer science faculty in **UBI**. This environment is centered around research topics such as software reliability, security and formal verification. Within **RELEASE**, there is an ongoing project, Formal Verification of Tezos Smart Contracts (**FRESCO**), focusing on the research and development of security analysis tools for Tezos' smart contract language, **MICHELSON**. **RELEASE** presents a strong collaboration between its members and, considering that the current proposal faithfully aligns with Tezos' security stances, it is only natural to choose Tezos as the targeted platform for this dissertation's efforts.

To that end, this work culminates in providing the Tezos community, specifically smart contract developers, with a tool for their workbench, capable of verifying properties for their smart contracts. It is designed to satisfy simple user-defined properties, dispatching verification tasks to *WhylSon*, a deductive program verification tool. Amidst development, user-defined properties not only help the smart contract developer better comprehend design specifications, but also contribute to lowering the entry barrier for business corporates to collaborate with and aid developers in their task.

In short, the solution presented in this document will strive to **aid developers and business corporates alike to collaborate on the task of establishing contract specifications for their software in the Tezos platform** by setting a performant, reliable and effective security standard, seamlessly woven into the smart contract development cycle.

1.2 Problem Summary

This section precedes and acts as an introduction for the upcoming discussion in section **4.1.1**. The content is organized in bullet points for better clarity.

- With the introduction of Turing-complete computation in blockchain platforms, both possibilities and problems of traditional software are now present in these environments, one common, but severe problem, the inability for contract maintenance, or simply pulling the contract out of the network. Additionally, if developers do not account for emergency scenarios, tragedies might ensue where developers effectively can do nothing other than watch as their contracts are being exploited, which is the case for one of the most well known incidents in the Ethereum blockchain;
- Ethereum, being the current leader in smart contract applications, is an enticing target for hackers to explore exploitable contracts. However, exploitable contracts exist because security measures were not taken seriously, or factored into its development at

all. Being the development of smart contracts a recent practice, there has to be accessible guidance material for developers to access, in addition to security tools that are effective and intuitive to use;

- Blockchain platforms are rapidly evolving. It is very likely that other platforms become targeted as their value increases. As of right now, there has been a misguided overinvestment in research and efforts into security for the smart contract leader, where other platforms are severely lackluster in comparison. Other platforms must learn from past incidents by acting accordingly in the present, through investing in better security tools, designing safe languages, among other measures;
- The Tezos blockchain is firmly staking on security by construct infrastructure, in both its network protocol and its native language. It has the potential to become one of the leader smart contract platforms in the future, though it has not seen many developments in smart contract security tools. Furthermore, the existing tools are not seeing wide adoption from its smart contract development community;
- The efforts of this dissertation are paving the way for an indirect security approach, in which the common developer is encouraged to take the initiative in incorporating a valuable security tool into their workflow. The fruits of this labor, the dissertation's contributions, are presented in section [5.2.4](#);

1.3 Objectives

Throughout the development of this dissertation, one may assume two types of objectives — personal objectives as well as the project's objectives — Firstly, it has to be properly stated that the current work is the culmination of five academic years in [UBI](#), enclosing the master's degree. Unsurprisingly, this is yet the student's greatest academic challenge to overcome and conversely its biggest learning opportunity, with a chance to differentiate oneself and succeed in the areas explored in this dissertation. That said, the following can be considered **personal objectives**:

- Develop a masters' worth of knowledge on smart contract development and its security topics;
- Learn more in-depth about other blockchain related topics, such as its economical aspects;
- Become well versed and practiced in written English investigative nomenclature;
- Incorporate proper organization and investigative standards for the rest of its professional career.

Secondly, this **dissertation's objectives** refer to its contribution prospects after its completion. This dissertation document aims to:

- Briefly introduce the reader to general blockchain knowledge and terminology;
- Alert the reader of past and present incidents that befell on blockchain platforms;
- Raise awareness on the dangers associated with smart contract development;
- Expose the reader to current state-of-the-art smart contract security approaches and tools;
- Propose a solution capable of minimizing aforementioned dangers;
- Present and devise a realistic plan for the development of the solution;
- Detail the process leading up to the solution's conception;
- Demonstrate the solution's capabilities with practical examples.

1.4 Document Outline

- The first chapter — **Introduction** — briefly contextualizes how this dissertation topic came to be, presents this dissertation's main objectives as well as this document's outline;
- The second chapter — **Technical Background** — a complementary chapter that provides more detailed information regarding the building blocks of concepts associated with the dissertation;
- The third chapter — **Literature Review** — discusses state-of-the-art papers in the dissertation's research scope, gauging this area's research status, findings, problems, and directions;
- The fourth chapter — **Problem Statement and Work Plan** — addresses the main problem theme this dissertation aims to solve, in addition to presenting steps that confer it a plausible solution;
- The fifth chapter — **Implementation** — details the tools and technologies used throughout the entirety of the implementation phase as well as implementation details of the final product;
- The sixth chapter — **Practicality Assessment** — assesses the solution practicality with relevant applications;
- The seventh chapter — **Final Remarks** — the epilogue for this dissertation document, summarizing its contents as well as the author's final remarks on it. Additionally, it proposes future work and avenues for further improving the document and proposed solution.

Chapter 2

Technical Background

This chapter will introduce the preliminary concepts in which this dissertation theme is built upon. The introduced concepts are written concisely in a beginner-friendly manner for those who are unfamiliar with blockchain related topics. The relayed concepts do not encompass the entirety of the blockchain knowledge breadth nor its depth. For readers who seek more than what is presented in this chapter, consult the cited literature [1–4] for a comprehensive understanding of the topics. Section 2.1 will introduce the blockchain itself as a topic, section 2.2 will focus on smart contracts and lastly section 2.3 will present more details regarding the Tezos blockchain, the blockchain of focus for this dissertation.

2.1 The Blockchain

The blockchain is commonly known as a **decentralized distributed ledger** inlaid with **cryptographic security** over a **peer to peer network**, implementing a **consensus protocol** for synchronization over its peers. The concept of blockchain has become deeply linked with the network protocol it implements but, at its core, a blockchain is a data structure most similarly to an **append-only, back-linked linked list**, with a finite amount of nodes called blocks.

Figure 2.1 illustrates the contents of generic block.

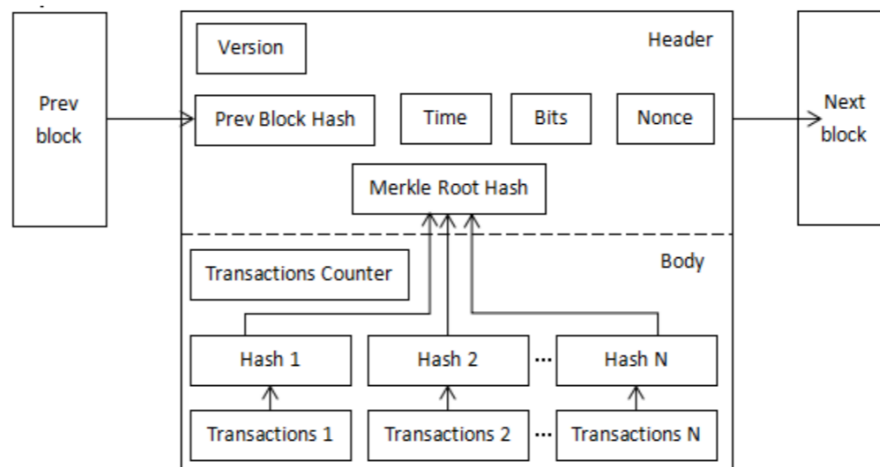


Figure 2.1: Generic block structure.

Each new block is linked through a cryptographic hash function, normally Secure Hash Algorithm (SHA) 256, to the previous one, just like a chain. Additionally, the more blocks there are, the more difficult it is to modify any data in the chain. This means that if a harmful actor

wants to cheat the system, they would have to modify everything from the first block ever created — called the “genesis block”. However, a cheat is able to be done if the harmful actor holds the majority of the network’s combined computational power (51%). This property is called as the blockchain being “Byzantine fault-tolerant”.

The history and principles of blockchain are deeply rooted with the emergence of Bitcoin [5] — an electronic cash system — looking to compete alongside fiat currencies [4], functioning free of control or manipulation from a centralized entity, that is, without the need of a mediating trusted third party. Bitcoin was the first protocol to ever develop a practical, working solution against the double spending problem [2], being able to become the first ever successfully implemented decentralized network protocol, whereas already having other protocols failed in the past [3].

The sense of community when participating in a blockchain can be a very inclusive experience, and as a community, problems are prone to happen. Being the blockchain immutable, if there ever is one small problem, it cannot be patched without doing what is called a blockchain fork [4]. These phenomenons can either be constructive, or destructive to the community.

Forks happen when a radical change to a network’s protocol is done, negating a part of the blockchain that is set in stone. Forks require users to manually upgrade to the latest version of the protocol software. Bitcoin has had a history of forks, some resulting in bitcoin derived coins such as Bitcoin Gold and Bitcoin Cash [5].

The concept of transaction is also redefined for a blockchain environment — users are able to change the state of the blockchain by sending transactions to each other’s addresses — in this protocol’s case, transactions comprised of sending a positive arbitrary number of coins to another address.

But transactions would not be processed immediately, in fact, they could be reverted. So, a new concept was introduced to refer to the act of processing/validating transactions in the blockchain — mining — for the user to mine a block they are required to solve a puzzle, in this case, it is called the zero prefix hash puzzle. Solving this puzzle is done by matching the output of computing the one way cryptographic hash function of the block the miner is trying to append to the blockchain, transactions included, with at least the number of zeros stated in the prefix. The catch here is, incrementally adding a zero to the target prefix exponentiates the odds of a miner ever solving the puzzle for that length. However, the odds are designed to be low, so that blocks are mined in a relatively constant rate. The adjustment of the puzzle difficulty is kept relative to the overall computational power of the entire network combined, and plays a crucial role for when hardware is updated.

By successfully mining a block, a user is rewarded with currency and the “change” of the transactions included in the block. There is a limit of issued currency for the blockchain, and in every mined block the rewards are following the downwards slope of an inverse function. However, even if the reward lessens overtime, the incentive is still supported by the

¹<https://www.ig.com/en/glossary-trading-terms/fiat-currency-definition>

²<https://www.investopedia.com/terms/d/doublespending.asp>

³<https://www.investopedia.com/tech/were-there-cryptocurrencies-bitcoin/>

⁴<https://www.investopedia.com/terms/h/hard-fork.asp>

⁵<https://www.investopedia.com/tech/history-bitcoin-hard-forks/>

transactional fees.

However, mining is not always the best for every user. Solving the puzzle taxes the hardware in the form of electricity, while some hardware are also more efficient than others in this job. As connoisseurs of the network kept trying to upgrade their computational power, heightening their chances of being the miner, so did the network's average computational power. Mining profitability becomes nil when the rewards associated with mining become lower than the cost spent to mine the block, on average.

Following the success of Bitcoin, many aspired developers started innovating with this highly grossing technology. Seeing the interest in this topic, new blockchains with differentiating features and capabilities have also emerged. One aspect that has also evolved with this technology is the user access. As such, these may have evolved respecting one of the following ideologies. From this point on, the focus is directed at public blockchains, since the others are of no relevance to this dissertation's objectives.

- **Private** – A blockchain that allows only selected entry of verified participants. These can join this type of network only through an authentic and verified invitation;
- **Public** – The most well known type, the one followed by bitcoin and many other top grossing blockchain platforms. Every user is free to join and participate in the core activities of the blockchain network;
- **Permissioned** – Permissioned blockchains allow for a mixed public and private user experience, these types support a customizable view options for its users, enabling users to be granted more or less restrictions.

Just like in ideologies, some have innovated in their consensus protocol. Some of the most common mechanisms are listed below.

- **Proof of Work (PoW)** – This protocol can be understood as a race condition to solve the puzzle of the zero prefix hash. It has shown to be very inefficient and damaging to the environment⁶. Despite the two biggest blockchains still adopting this method, new technologies are starting to reject it;
- **Proof of Capacity (PoC)** – This consensus first generates large data sets called "Plots". The more plots a miner records, the more lottery tickets he owns. Hence, the more hard-drive space (capacity) he has, the more chances of winning the rewards. This also mimics PoW mining: instead of accumulating hash power, you accumulate hard-drive capacity;
- **Proof of Stake (PoS)** – This protocol is a mechanism that rolls the dice for validators. The higher the amount of coins the validator has invested in the chain, the higher is its chance to be chosen as validator, appending the new block to the blockchain. However, the reward for this protocol only consists of revenue from transaction fees;

⁶<https://medium.com/logos-network/why-proof-of-work-is-not-viable-in-the-long-term-dd96d2775e99>

- **Delegated Proof of Stake (DPoS)** — Is an extension of **PoS**, with an added Delegation phase. During this phase, decentralized votes are made by the witnesses, electing a validator of choice. The witness group has the same function as the parliament in a parliamentary democracy;
- **Liquid Proof of Stake (LPoS)** — Is now an upgrade of **DPoS**, diluting the pool of bakers (validators) while increasing benefits for delegating coins to major bakers. Delegating coins is seen as a liquid investment, enabling the baker to revoke the delegation at any time. Delegators in the pool of the validator that won the dice roll are now rewarded according to a share proportional to the delegated value.

Following the explosive growth of blockchain topics, there was one platform that especially took over the Bitcoin spotlight with a one particular well-thought-out innovation — Ethereum [6]. This platform revealed horizons never thought to be seen in the world of blockchains. The introduced Ethereum Virtual Machine (**EVM**) is a Turing-complete system able to solve any computation problem using its low-level scripting language designated EVM bytecode, in short, a stack-based machine. Accompanying the bytecode, was a higher abstraction language called *Solidity* [7].

This opened endless possibilities for the applications of a blockchain. Having software being able to be executed on top of it meant automation of services was at hand through this network.

The impact was so large, that there is a terminology separating blockchains that enable the execution of code of those that cannot — first generational and second generational, respectively.

2.2 The Smart Contract

The software that is deployable to the blockchain is called a smart contract. This concept was, however, not outcast — in 1996, Nick Szabo [7] proposed the definition of a smart contract, even before Bitcoin’s emergence. Its definition is quoted as such,

“A smart contract is a set of promises, specified in digital form, including protocols within which the parties perform on these promises.”

A good analogy present in this Szabo’s approach is the similarities between a smart contract and a common vending machine.

- A reactive entity;
- Transparency in its process;
- Only reacts if its pre-conditions are triggered;
- Once it triggered, it agrees on what it was specified.

⁷<https://docs.soliditylang.org/en/v0.8.12/>

A person that buys something from a vending machine has to introduce money to start interacting with it. Then it has to choose which item to buy. If the item is available, then it can be bought. If there is just enough money, the vending machine only outputs the agreed upon item, else it also comes with change.

Smart contracts are called by users by referring transactions to the contract’s address. If the transaction is agreed across the network, all the existing peers have to execute the contract code with the current state of the blockchain with the relevant input parameters. Figure 2.2 illustrates the blockchain architecture prepared with smart contracts, extracted from [8].

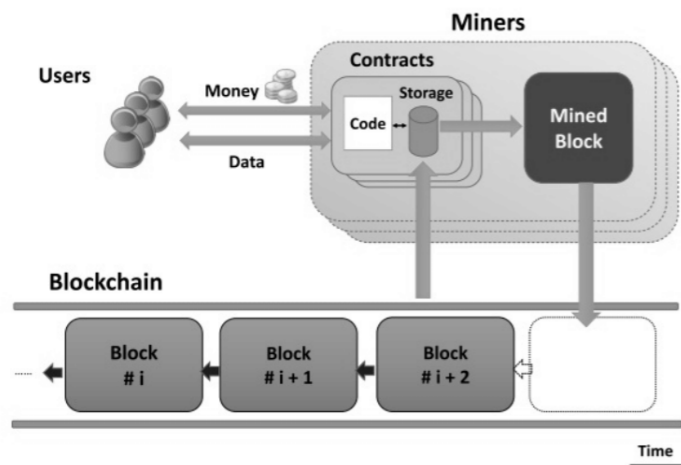


Figure 2.2: Blockchain architecture for smart contract enabled blockchain.

In other words, a smart contract executes a set of pre-defined instructions. Once deployed into the blockchain, it becomes immutable. Using transactions, a user is able to advance the blockchain state. To deploy a smart contract, one must also make a transaction for it to become permanent. Smart contracts are a highly specified bank account, which can hold a balance and also communicate with other smart contracts or users. So, in Ethereum, every address can be called a stateful account.

While having a turing complete language enables endless possibilities, it also opens many recipes for disaster, such as the halting problem. One measure to limit such dangerous situations, was the implementation of limiting factor when executing smart contracts, gas — gas is assigned a price in *wei* to every opcode in the EVM bytecode. For this end, running smart contracts costs money, but there is also an upper limit of how much a transaction can consume gas.

The development of smart contract gave rise to many organizations to become settled in the blockchain market but, as the scene had just started, the dangers of it were completely unknown. *Solidity*, when launched, was especially not the most mature language. It was presented as being very similar to Javascript and Python, which led to users creating unrealistic expectations and misunderstandings for this newly created language. The severity of this issue, among other reasons, lead to tantamount incidents, explored in section 3.2.

2.3 Tezos

The blockchain of focus for this dissertation is Tezos ^[9] platform. This section will proceed on introducing the Tezos ecosystem.

Tezos ^[9] is a public, smart contract-based, modular, open-source blockchain protocol relying on a low power consumption and energy-efficient consensus protocol. The protocol itself has never before seen mechanism — the self-amendment process, allowing continuous improvement and rapid innovation through community collaboration — preserving the integrity of its consensus for years to come, while also eliminating any uses for a fork. Tezos is also fundamentally designed to provide code safety and reliability through formal verification in its specification language, *Michelson* ^[8].

Through self-amendment, the protocol has undergone updates for more than 10 times, all improving on new features, quality of life changes, the self-amendment process itself, changes to its smart contract language, reduced gas costs, shorter validating time, etc. Each update has been largely successful, and with it, generating interest from the overall blockchain sphere and organizations from around the globe. An example portraying the full potential of this feature, was of a research paper having discovered a possible vulnerability in the consensus protocol ^[10], specifically during baking procedures. Once the community was alerted by it, it was quickly addressed and, the following patch, the vulnerability was no more.

The ease of formal verification for this platform is made possible by respecting the norms and philosophy of the functional language paradigm, in which all of Tezos' environment was built upon. It is a style of building and structuring the elements of computer programs, treating computation as an evaluation of mathematical functions, avoiding changing-state and mutable data. This defensive style of programming is an attitude towards the minimization of runtime errors while also ensuring the correctness of implementation.

Given its ability to assure soundness and reliability of programs, formal languages are strongly encouraged for financial and business use in software. Mathematical proofs, verification of predicates, are one of these languages' strong suit — avoiding monetary losses in any unpredictable program state, specifically making sure that business flows just as expected by design. The same is applied to *Michelson*. As of right now, the leading and most expanding sector in this platform are financial applications.

Alongside *Michelson*, the community has developed competitive higher level languages, abstracting the low-level stack-based into the common register one, for ease of development processes. Tezos officially recognizes *Archetype* ^[10], *LIGO* ^[11], and *SmartPy* ^[12] as the main higher-level languages for writing smart contracts in this blockchain. There are more choices available, such as *SCaml* ^[13] and *Fi* ^[14], however, these have yet to gain the traction and maturity to compete with the aforementioned choices.

⁸ <https://tezos.com>
⁹ <https://tezos.gitlab.io/michelson-reference/>
¹⁰ <https://archetype-lang.org>
¹¹ <https://ligolang.org>
¹² <https://smartpy.io>
¹³ <https://www.scamlang.dev>
¹⁴ <https://learn.fi-code.com>

In short, the following properties can be considered Tezos' highlights, as a platform, which are a byproduct of the two blockchains that have shaped the blockchain universe, and expanding in relation to security, reliability and being energy-friendly.

- **Self-amendment** nullifies the need for forks, contrasting with Bitcoin, later from Ethereum, that have suffered numerous forks and community partition;
- **LPOS is a mechanism that is resource friendly**, contrasting with Bitcoin and Ethereum's resource intensive **PoW**;
- **Michelson was a language especially designed with easing processes of formal verification**, contrasting with the incidents coming for earliest stages of *Solidity*, by having an already mature and secure first stance since the beginning.

During investigative time, the student was able to discern that this platform is very inclusive of its newcomers, having a wide variety of sources available where one may learn everything about this platform ¹⁵.

2.4 Conclusion

This section introduced the required concepts in order for the reader to better grasp the branching topics ahead. From general blockchain topics, to smart contract, to the Tezos platform.

¹⁵<https://www.opentezos.com>

Chapter 3

Literature Review

This chapter will present the current research status of smart contract security topics. Sections 3.1, 3.2 and 3.3 will respectively focus on smart contract vulnerabilities, exploits and security tools, first by listing its contents followed by a small description. Section 3.4 will discuss how the preceding issues relate with the dissertation’s objectives, as well as highlighting other pertinent papers that do not fall directly under any of the previous sections.

Literature for the aforementioned specified topics places much greater emphasis, and are also much more likely to be found, within the realm of Ethereum — the platform is the current leader in smart contract related topics, having been the first to introduce and put into practice the concept of smart contract, requiring greater efforts from the community to solve the problems as they arose. As a result, the security aspect of this platform’s community was and continues to be quite active, resulting in numerous publications focusing on Ethereum. Unless explicit, the concepts in the following sections will all implicitly allude to the Ethereum platform, including *Solidity* and EVM bytecode.

3.1 Smart Contract Vulnerabilities

Literature is very rich when it comes to documenting vulnerabilities for smart contracts. From research papers [11–17] to community driven websites¹, a joint effort from the community is made to spread awareness on these problems. The website for Smart Contract Weakness Classification (SWC) Registry is a trusted source for documenting vulnerabilities for Ethereum smart contracts, consisting of 36 known vulnerabilities. The next itemization includes those vulnerabilities cross-checked with those that are relevant for the reviewed surveys.

- **Reentrancy** — This phenomenon occurs when external contract calls are allowed to make new calls to the calling contract before the initial execution is complete, that is, before the initial contract logic attains closure. Such problem may lead to unexpected and unthinkable contract state changes in the middle of its execution, as a result of a call to an untrusted/harmful contract.

To avoid this vulnerability, any smart contract execution must attain closure for its internal logic before making any external calls. Also, some smart contract verification tools were designed to specifically identify reentrancy pattern and accuse them, such as ReGuard by Liu et al. [18] and Rodler et al. [19].

¹<http://swcregistry.io/>, <https://dasp.co/index.html>

The *Michelson* language is, by design, much safer from reentrancy attacks due to its semantics. External calls are unable to be made until full closure is obtained for the first contract execution;

- **Transaction order dependence** – This vulnerability stems from the blockchain’s own properties. Having any user be able to check the transaction pool at any given time, enables harmful users to take advantage of this knowledge to act accordingly to their ends, against other known users. It is especially prominent for situations in a race condition, where transactions with higher fees are generally executed first.

One way to avoid this problem is to mask the identity of the user, through a process called commit reveal hash scheme ²;

- **Call to the unknown** – *Solidity* contains a set of primitive call functions, those being `call`, `send` and `deletacall`. These functions can execute and trigger specified functions in external smart contracts or their fallback function. For a malicious user, the fallback function can be implemented in such a way to create a loophole in the caller contract. From those primitives, `delegate call` can be especially dangerous since the original contract context is used by external contracts, enabling such to maliciously alter the context.

This issue can be absolved by ensuring that such primitives are only called for trusted external contracts and functions;

- **Mishandled exceptions** – *Solidity* does not have a uniform way to handle exceptions, be it manually thrown via `throw` keyword or from unexpected runtime errors, such as the smart contract running out of gas or reaching its call stack limit. These situations become even trickier when the exception is triggered in an external call, having the language sometimes trouble when propagating the exception towards its original execution.

These issues can be prevented by taking preventive actions against unexpected outcomes from critical operations within the smart contract.

- **Timestamp dependence** – Smart contracts often use a block timestamp to trigger conditions to execute any sort of critical operations. Should be noted that timestamp values are given through the miner of the transaction. In this light, harmful actors can adjust the timestamp to a specific value that influences the timestamp-dependent condition, favoring them.

Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. An alternative is to use a trusted Oracle for timestamp operations, since they are unbiased towards the miners;

- **Integer overflow/underflow** – An overflow/underflow happens, akin to classic software, is when an arithmetic operation reaches the maximum or minimum size of a

²<https://swcregistry.io/docs/SWC-114>

given type. A paper by Torres et al. [20] introduces a tool that was tailored to identify integer bugs.

These situations can be avoided by using `SafeMath` library in *Solidity*, or the use of arbitrary precision types. Since version 0.8.0, these scenarios are much less likely to have any consequences — if arithmetic operations reach their type threshold, the transaction is, by default, reverted. Old behavior can still be obtaining through the use of a special keyword.

The *Michelson* language was implemented with integers of arbitrary precision, rendering nil any situation of overflow/underflow;

- **Greedy/Prodigal contract** — Smart contracts are called greedy or prodigal depending on how the contract's balanced can be manipulated by unwanted interactions — the first for when the contract's balance cannot be manipulated by anyone, not even authorized actors, while the second for when the balance can be manipulated by unauthorized users at will.

An article by Nikolic et al. [21] does a mass scouting of deployed smart contracts in the Ethereum blockchain, submitting them for verification in the MAIAN [22] tool, also one of Nikolic's works.

- **Weak field modifiers** — In *Solidity*, both functions and variables have public scope if not specified. If, during implementation, developers forget one of these's visibility, it can cause the smart contract to inadvertently be accessed by unwanted users in undocumented critical entry points.

This vulnerability is simply mended by the developer's conscious decision of modifiers — external, public, internal, or private;

- **Unprotected self-destruct** — Unwanted actors having the ability to self-destruct the contract due to weak access control modifiers.

To avoid these problems, removing the self-destruct functionality should be mandatory unless absolutely required. If there is a justifiable use-case, implement the self-destruct action of the smart contract through the approval of more than one user.

- **Predictable randomness** — The blockchain has no source of reliable randomness. More often than not, the operation solely relies on blockchain context attributes such as timestamp, current difficulty and block hash, which are values available to every user.

This vulnerability can be mitigated if the use of randomness is applied with the RANDAO [3] commitment scheme. Alternatively, the use of a trusted Oracle as a source of randomness can be used.

These are the vulnerabilities exposed by the literature. However, there is still one type of vulnerabilities that permeate all kinds of software and, arguably, can be seen as the root of

³<https://github.com/randao/randao>

these problems — **logical flaws** — these flaws can further be specified into the vulnerabilities reference by literature however, at their core, they exist because code practices are not consistently being practiced.

For example, the reentrancy vulnerability can mostly be solved if the external call is made as the last statement after reaching appropriate closure of the smart contract's internal logic for a function. Another example, such as weak field modifiers, is due to the developer not paying close attention to access control modifiers.

Logical flaws, without association to any specific vulnerability, can also have a tremendous impact on the smart contract. For example, implementing a condition that does not go according to specifications can go silent without any developers realizing.

These and many more vulnerabilities can be avoided by developers by following code practices and making conscious code decisions during development processes, as well always paying close attention to the specifications of the implementation. However, some vulnerabilities are still very difficult to notice, hence the need of security verification tools.

3.2 Real World Exploits

Hand in hand with the vulnerabilities listed, real world exploits were also properly documented, both in articles [11–15, 23–25] and across websites. An attack is only made possible if any vulnerabilities present are targeted and exploited. Most attacks are a combination of exploiting more than one vulnerability, having one leading to others.

- **The DAO (2016)** — Being referenced by every reviewed survey, it became and beacon that has shaped the current scenario of smart contract security and Ethereum itself. It also was a triggering factor for a hard fork in the chain, an incident marked in the history of blockchain technology for when security is taken as an afterthought. This smart contract served as an autonomous crowdfunding campaign seeking to help newly started organizations in the Ethereum platform. The implementation for this smart contract was detected with having a loophole in one of its functions, enabling a harmful actor to steal about \$50M dollars. A very in-depth explanation of the attack can be seen in this website ⁴;
- **Parity Multisig Wallet (2017)** — This incident was also often paired with the DAO attack for every survey document. The Parity Multisig Wallet was a wallet application, the contract that was exploited not only once, but twice, in the span of 4 months. The first attack resulted in the theft of \$30M dollars due to weak access modifiers for one of its functions, enabling any user to call a critical and supposedly restrictive function. The second attack resulted in the locking of \$150M dollars in some of its user's wallets due to non-authorized user being able to destroy the contract... by accident! The official smart contract developers acknowledged both attacks (first ⁵, second ⁶);

⁴<https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/>

⁵<https://www.parity.io/blog/the-multi-sig-hack-a-postmortem>

⁶<https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/>

- **King of the Ether Throne (2016)** — This smart contract was a simple game which consisted on the players accumulating points by being the king. Remaining king the longest would net more profit to the dethroned user, while the aspiring king would have to incrementally pay more to become the king. This smart contract contained a mishandled out of gas exception, reverting the payment to the current king, while it being dethroned all the same. This website ⁷ refers to the postmortem page of this smart contract;
- **BeautyChain Token (2018)** — The BeautyChain Token is an asset inherent to a sub-chain of Ethereum. The code of that sub-chain was detected with a vulnerability of integer overflow, in which a harmful user took advantage of to steal 10^{58} tokens or \$900M dollars worth ⁸. This website ⁸ details more about this specific integer overflow attack;
- **Govern Mental (2016)** — This smart contract played a Ponzi scheme for its players. Players join by sending Ether to the contract, if the clock goes 12 hours without adding a new player, the last player claims the jackpot. Each participant is appended to an array in the smart contract, claiming the jackpot clears the array, requiring its traversal until the end. At some point, the array became big enough as to the array not being able to clear itself in a single transaction, due to limited gas costs at the time. In short, this smart contract suffered vulnerability of mishandled out of gas exception, while simultaneously locking the smart contract. This forum publication ⁹ shows a discussion between the owner of the contract and the community;
- **SmartBillions (2017)** — This smart contract enabled players to gamble and play lottery. However, this lottery had a predictable source of randomness, in which a gambler took advantage of to win 400 Ether prize as a trial run, before the player could hit the jackpot on its next attempt. Before that happened, its developers rug-pulled the entire prize pool, walking away with 1100 Ether of the 1500 ETH jackpot. This forum publication ¹⁰ presents a discussion for this incident.

3.3 State-of-the-Art Security Tools

This section will list smart contract security tools that were found when reviewing literature, briefly introducing them. For better clarity and visibility, table ^{3.1} presents information regarding verification tools designed for Ethereum, while table ^{3.2} for Tezos.

⁷ <https://www.kingoftheether.com/postmortem.html>

⁸ <https://nvd.nist.gov/vuln/detail/cve-2018-10299>

⁹ https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/

¹⁰ https://www.reddit.com/r/ethereum/comments/74d3dc/smartbillions_lottery_contract_just_got_hacked/

3.3.1 Smart Contract Security Mechanisms

The tools introduced all had different backgrounds and mechanisms used in order to verify or grant some way of security to smart contracts. In that regard, it is wise to also understand their security mechanisms.

- **Symbolic Execution** — An analysis technique that replaces the values of program variables as symbolic expressions to uncover execution paths present in a **CFG** built from program code. The **CFG** is then studied by **SMTs** to pinpoint vulnerable paths;
- **Abstract Interpretation** — This method formalizes the idea of abstracting mathematical structures in order to achieve soundness in program analysis through over-approximating semantics of a program. For smart contracts, this technique ignores certain trivial operations while focusing more on order of operations and correct semantics;
- **Fuzzing** — A testing technique that provides random data into a program's entrypoint. This sort of testing enables smart contracts tools to simulate executions through possibly unexpected input values into smart contract entrypoints, testing their boundaries. Through it, certain vulnerabilities are more susceptible to be caught, but are not always guaranteed;
- **Runtime/Dynamic Verification** — Contrary to most methods, runtime verification is used for running programs, in this case, deployed smart contracts, preferably on test networks. It allows for analysis of program executions, while also closely monitoring any variables or values of interest. While this method does not require a precise modeling of the blockchain environment it is still not commonly used, for deploying into a test network still has resource limitations. This method has been commended by researchers since it has the potential to explore program states that may be more difficult to reach otherwise;
- **Model Checking** — Adopts the usage of **FSMs** to confront a given set of formal properties against the mathematical model of a program. Once the inputs are given, this approach automatically verifies a model through confronting it against the provided formal properties. If a property holds for all states, the program is considered to be formally verified for that property, otherwise, it is given counterexamples. This technique allows users to define their own properties instead of simply verifying common patterns;
- **Theorem Proving** — A method that performs program verification through transforming the programs and its requirements into a particular mathematical logic, deriving a formal proof of satisfaction of these requirements. This particular method is incredibly flexible and oscillating in terms of effectiveness, as it is as effective as the understanding and creativity of the user guiding the proof, being that the most skilled user should be able to arrive to some conclusion for any program. Aforementioned rea-

son is both the reason for this technique’s highly effectiveness and issue for not being commonly used;

- **Program Verification** — This semi-automatic verification method translates program source code into the specification language of a framework. The new program is then subjected through **SMTs** to check against program and semantic correctness or other known vulnerability patterns, whose specifications need to be formally stated. This approach is very similar to model checking, though there is proof that it performs better in a more realistic setting;
- **Machine Learning** — An unorthodox approach for detecting vulnerabilities in smart contracts is the usage of machine/deep learning. Models such as convolutional neural networks are fed with a wide variety of contracts labelled by vulnerability, however, this only allows such tools to detect known vulnerabilities.

The reviewed literature states that static analysis (static in a broad sense, in which it does not require a contract to be deployed in a network to be verified) methods are the most employed among smart contract verification tools, especially symbolic execution. This is the expected outcome due to the nature of the blockchain environment itself — programs should be as statically safe as possible before ever being deployed to the main network — and also the fact that the usage of **CFC** for the exploration of critical paths and possible bugs is a popular practice in traditional software. However, it is also stated that formal verification methods should become a more common practice.

One great distinction between static analysis and dynamic analysis is that the first detect vulnerable contracts, while the second detects ones that are exploitable. Perez et. al [57] conducted a massive research on deployed smart contracts on the Ethereum platform, finding that most contracts truly have some sort of vulnerability detected by verification tools, though, an interesting finding was that only a small portion was truly exploitable. Nonetheless, the results shown should not in any way alleviate pressure from developers for their contracts.

Most tools listed are freely available and open source, while some are not. Limiting security behind a paywall is one of the reasons such tools are not massively used by the smart contract development community. Additionally, auditing **A** is one existing method for granting smart contracts the required reliability. Though, such auditing firms charge a lengthy fee for their services, when their process normally subjects the smart contracts in question through the already explored tools. As of right now, clearing the audit might be the highest seal of certification for a smart contract, depending on the firm, but it is also the least accessible to the general community.

3.4 Discussion

One glaring fact that is instantly recognized is the sheer amount of existing verification tools for the Ethereum blockchain, moreover, the tools presented are only a small sample of the

¹¹<https://www.immunebytes.com/tezos-smart-contract-audit/>

ones explored. The referenced sources explored more than a combined of 100+ tools for this platform, while all Tezos verification tools/methods are all presented. Such proportions can also be estimated for other blockchains. This presents a problem in itself due to blockchain platforms rapidly evolving, since the majority of research resources and efforts are being poured into one single platform.

According to the referenced literature [16, 17, 24, 58, 59], directions point towards more than mindlessly developing smart contract verification tools — with the current over saturation of tools, specially in Ethereum, it is confusing and a time-consuming task to figure out which tools work best. Since different tools target different security risks, heterogeneity becomes a problem since the developer might require various tools for the contract to be considered safe, which is not an ideal scenario. Additionally, it is greatly recommended for tools to employ both static and dynamic analysis, known as hybrid analysis, since coupling these two approaches tackles the security concerns from two different angles, generally resulting in better efficiency.

Ideally, community effort should be guided into developing one single framework, or an ensemble tool, capable of verifying if possible all known vulnerabilities and the liberty to also specify other user-defined properties. For other platforms, there is indeed scarcity of tools. Learning once again from Ethereum, investigative resources and efforts should be poured into other platforms, now in a smarter manner.

Another topic is striking the security problem very early in development, that is, there should be standards and practices for smart contract development, and stricter, type-safe programming languages for this cause. The blockchain environment has glaring and serious differences compared to traditional software development, with developers crossing over fields, it is likely that misunderstandings might occur. Blockchain platforms must better promote guides, documentation and best practices for developers to be guided with. Another possibility for this would be reducing the manual input of developers by providing smart contract templates.

As a final note, a point that is not discussed within the literature is the implementation of supportive tools to connect the already known solutions to development environments. It is felt that solutions existing is simply not enough — smart contract security techniques must become an integral and infallible part of development. If security is ever implemented right, it must be free, accessible, effective, performant, and easy to use. Only then, will security truly be adopted by the masses.

3.5 Conclusion

This chapter presented the findings of the literature review done by the student. The most common vulnerabilities, known exploited attacks and listing security tools and how they work, all present concrete evidence for the problem to be described in chapter 4. The discussion here is still relatively tame, having the sections discussed the most important or common found points. For a complete in-depth explanation of every section, it is recommended to read on the referenced literature.

Designation	Author	Small Description
ESBMC-Solidity	Song et al. [26]	Formal analysis of vulnerabilities through translation of <i>Solidity</i> code to an Intermediary language (II) and consequent Satisfiability Modulo Theories (SMT) solvers.
EtherProv	Torres et al. [27]	Instrumentation of EVM bytecode through static and dynamic analysis of <i>Solidity</i>
EtherTrust	Grishchenko et al. [28]	Static analysis of EVM bytecode.
F*	Swamy et al. [29]	Equivalency analysis between <i>Solidity</i> and EVM bytecode.
FEther	Yang et al. [30]	Formal modeling of <i>Solidity</i> in Coq [31] framework, enabling Coq proof assisted verification.
FSolidM	Mavridou et al. [32, 33]	Use of Finite State Machine (FSM) to model smart contract, with consequent generation of <i>Solidity</i> code.
Isabelle/HOL	Amani et al. [34]	Formalization of EVM bytecode in the Isabelle/HOL framework.
KEVM	Hildenbrandt et al. [35]	Formalization of EVM bytecode through the K Framework.
Manticore	Mossberg et al. [36]	A framework capable of dynamic analysis and symbolic execution of EVM bytecode.
Mythx/Mythril	ConsenSys [37, 38]	Automatic vulnerability scanning of EVM bytecode as a service, through the use of symbolic execution, SMT solvers, and taint analysis.
Octopus	Ventuzelo [39]	Security analysis framework for various platforms, including Ethereum, verification through Control Flow Graph (CFG) analysis.
Oyente	Luu et al. [40]	Symbolic execution tool capable of flagging EVM bytecode of common vulnerabilities.
Securify	Tsankov et al. [41]	Automatic security analyzer, supporting user-defined properties, for EVM bytecode.
Slither	Feist et al. [42]	Automated static analysis framework through dataflow and taint tracking processes in an II converted from <i>Solidity</i> .
SmartCheck	Tikhomirov et al. [43]	Static analysis tool for common vulnerabilities in <i>Solidity</i> code.
SODA	Chen et al. [44]	Online detection framework by instrumentation of EVM bytecode.
Solidifier	Antonino et al. [45]	Bounded model checker for <i>Solidity</i> language, through the formalization of a subset of the language itself.
Vandal	Brent et al. [46]	Security analysis framework supporting verification of EVM bytecode by converting it into semantic logic relations.
VeriSolid	Mavridou et al. [47]	A model-based approach for generating correct by design, formally verified <i>Solidity</i> code.
VerX	Permenev et al. [48]	Verification of functional properties in an II, similar to <i>Solidity</i> , through symbolic execution and delayed predicate abstraction, with consequent compilation to EVM bytecode.
Zeus	Kalra et al. [49]	Perform formal verification for an II translated from <i>Solidity</i> , through abstract interpretation, symbolic model checking and constrained horn clauses' verification checking.

Table 3.1: State-of-the-art verification tools found for Ethereum.

Designation	Author	Small Description
Albert	Bernardo et al. [50]	An IL for <i>Michelson</i> , uses Mi-Cho-Coq's compiler to obtain formally verified Michelson code.
Helmholtz	Nishida et al. [51]	Automated verification tool of <i>Michelson</i> code through SMT .
Mi-Cho-Coq	Bernardo et al. [52]	Formal specification of <i>Michelson</i> in the Coq [81] framework.
Tezla	Reis et al. [53]	Static analysis framework for <i>Michelson</i> .
WhylSon	Horta et al. [54, 55]	Automated formal verification of <i>Michelson</i> through the Why3 [56] framework.

Table 3.2: State-of-the-art verification tools found for Tezos.

Chapter 4

Problem Statement and Work Plan

The fourth chapter of this document entails on a segment for a more comprehensive breakdown of the problem statement surrounding the entirety of this project. The other segment is an educated estimate of the procedural steps for obtaining a working solution for the problem described.

4.1 Problem Statement

4.1.1 Problem

Throughout the entirety of this document, the problem statement has constantly been underlined and not directly addressed with proper reasoning. Reflecting on chapters' [2](#) and [3](#) topics, it is now possible to coherently reason behind the problem statement of “proving transactional coherence for smart contracts in blockchains”. It can be seen as a two-part question:

1. **What is the current problem with smart contract development?**
2. **Why should the problem be solved in the Tezos platform?**

To answer the first question, one should highlight how smart contracts should behave within the blockchain environment.

Smart contracts should only and only be able to fulfill its specifications, expecting no unknown behaviors.

If all software were to conform to what is specified in its code, no software would need to be submitted to any sort of security mechanism. Assuming no external factors can influence a piece of software, and that every specification is correctly implemented but also confers to what the developer envisions, then every piece of software developed is secure. In this argument, three distinct problems run at large that contribute to faulty software:

- Manual input of code is often riddled with bugs and logical errors due to possible misconceptions by developers;
- External factors do influence software, mainly from unexpected inputs from an ill intentioned actor (one may also argue that this problem is covered by the first one);
- The specification itself can be incorrect.

From these, the first is one of utmost relevance for the present work — the area of smart contract development is fairly recent, with it accompanying often a lack of knowledge that leads to doubts and misconceptions — not being experienced or knowledgeable enough in a language/environment may amplify the chances of a smart contract being identified with a vulnerability by an ill actor, and later exploited, evidenced by the incidents presented in section 3.2. The second problem may be considered related to the first one, since the developer is also tasked with specifying their software’s entry points and consequent restrictions, p.e. access modifiers, type, range of values, etc. The third is a considerable problem during requirements gathering stages, that is, a problem that comes from flawed business logic. In this case, if these flaws take too long to be detected, the software may have serious implications in the long run, since it is most unnoticeable if not detected prematurely.

Aforementioned problems are further amplified if the software resides in the context of a blockchain. In off-chain software, flaws detected post-shipping have a chance for the problem to be repaired with, however, having its associated loss in both money and product credibility. On the other hand, smart contracts, due to blockchain immutability, cannot simply follow the stages of immediate retraction, patching and re-deployment. Just as in classic software, any vulnerabilities detected post-deployment lead to an exponential rise of patching costs, more so on smart contracts, where there is a risk of the contract already being taken advantage of before vulnerability detection. This is a seriously impactful problem, consequences for this problem are, for instance, the already recorded incidents of asset theft, locking of funds, sudden destruction of the contract, etc.

The recorded events also contribute to questioning the credibility and viability of the markets settled in the blockchain. Actors may question themselves if theirs and/or their organization’s funds are truly safe.

In short, **smart contracts execute what is written in them, not what is expected by its developers.** It is the developer’s responsibility for these expectations to perfectly match the smart contract’s behavior under any circumstance. To minimize any future tragedies regarding one’s smart contract, the existence of verification tools greatly aid developers in this task, certifying their software of correctness and soundness properties, as well as the alignment of specifications with expected behavior.

The second question, is one most pertinent. Appeals from researchers state that Ethereum is the one platform that requires utmost attention to security norms — there are an exorbitant amount of organizations whose software is already deployed on the platform, as well as a constant flux of new users that are driven by the appetizing numbers and successful stories. In spite of such, why develop for Tezos? Should not the priority be to focus on minimizing the problem in the Ethereum platform?

A very fair argument, but the contextual reasons stated in section 1.1 are also taken into account for in this decision — the efforts for this dissertation are best directed towards the Tezos platform. It attempts to contribute beyond the current trend, especially when the state-of-the-art surveys on smart contract verification reveal not only the vast amount of already existing tools, but also heterogeneity regarding the security practices of said tools in Ethereum. Learning from past incidents and the present state of both security topics in Ethereum and

Tezos were vital to understand the best directions in which to devote current efforts.

Foreseeing the upwards trend on the Tezos platform, given its security stances, constant updates with new features and quality of life changes, the time is ticking for enforcing better security measures in this blockchain. Current efforts focus on delivering and contributing to the platform in order to minimize the chances of a blockchain hazard to occur again.

4.1.2 Solution

Having explored the problem at hand, it is now the appropriate time to explain the solution proposed. First, special emphasis should be given to the solution's purpose.

A plug-in capable of formally verifying user-defined properties for their *Michelson* smart contracts.

From the above statement, some specifications can be directly outlined, while some become mandatory for assuring the tool's quality and relevance:

1. The tool should understand *Michelson*'s context;
2. The tool should enable the developer to formally specify properties within *Michelson*'s context, without being intrusive;
3. The tool should be able to verify the specified user-defined properties;
4. The tool should be able to validate or refute the given user-defined properties — the former validating the smart contract code of the specified properties, the latter present feedback with a counterexample;
5. The tool itself should be flexible, performant, reliable and effective.

For the first point, Visual Studio Code should be the code editor of choice — the tool is one of the most well known amongst younger developers — with a variety of features that make it rise above its competition, such as, performance, User Interface (UI) visibility and clarity, and customizability. The last one is, by far, its most important trait, and the reason for enabling the development of the proposed solution in this environment in the first place. Not only does Microsoft add extensions themselves, but also enables and encourages the community to do so. The company is very supportive of this feature, creating and sharing a variety of resources of its Extension Application Programming Interface (API). The community is very welcoming of this feature, making this code editor the ideal environment for the proposed plug-in.

Items second to fifth present the solution itself. Up until this section, not much has been disclosed on as per how the operations within the tool will act. Reason being, the solution will require the help of an already existing and established tool — *WhyLSON* — already mentioned in section 3.3. Reiterating on what was disclosed in said section, *WhyLSON* is a state-of-the-art deductive verifier for *Michelson* programs, developed by Horta et al. — a PhD student and

veteran member of the **RELEASE** lab — whose thesis is being developed under the **FRESCO** project. More about this tool will be detailed in **5.1**.

WhylSon is a tool powered, in itself, by *Why3*, a well known framework for automatic deductive verification. In turn, *Why3* dispatches its verification processes into external provers, known as **SMT** solvers. Solvers are at the end of the chain, where, for the devised solution’s case, user-defined properties will be verified for their satisfiability or not. Feedback is then retroactively sent back towards the start, where the proposed solution will present if the specified properties could be verified or not.

One doubt still remains, which is how to represent the specified user-defined properties. Recently, *WhylSon* has made advancements in developing its own specification language, which is an augmented *Michelson* syntax with annotations. This question is yet to be cleared, but a solution shall definitely be reached at the time of the solution’s design.

Figure **4.1** illustrates a very plausible architecture for the verification loop of the proposed solution.

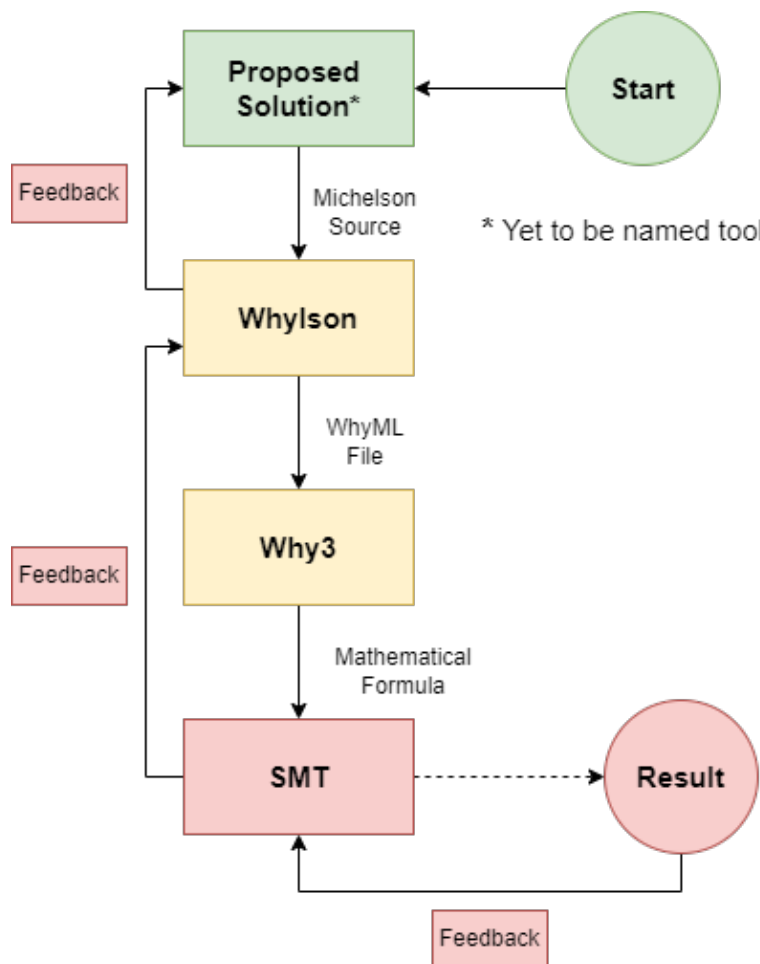


Figure 4.1: Verification loop for proposed solution.

Having presented the solution, one may ask two questions.

1. **Why not simply use *WhylSon* for meeting the problem statement’s ends?**
2. **What are the benefits of using the proposed solution?**

For starters, it should be clarified the roles of both the proposed solution and *WhylSon*. The latter can distinguish itself for being a state-of-the-art *Michelson* formalization tool, designed with its main purpose of being able to perform verification on formal specifications. The former is designed as a supportive tool for the latter, granting greater accessibility and ease of use by sliding a layer of abstraction between the act of smart contract development in *Michelson* and verification of specified properties of code on-the-fly.

The solution also **seeks to promote a better involvement of business entities** with establishing business rules in contracts through the specifications during development cycles, that way, both specialized developers and corporates can actively contribute to the smart contract code. Conversely, without the solution, business entities could be overwhelmed with the highly specialized tools, holding a higher entry level for development cycle contribution.

Of course, using the solution does not in any way obscure the usage of *WhylSon* — the tool should be required to be installed in the developer’s machines for the plug-in to be fully functional. If developers feel comfortable enough to be using this tool without the abstraction layer granted by the solution, then they have the chance to do so, after all, one of the solution’s main purpose is to promote collaboration between the entities responsible in defining contract rules and its implementation.

That being said, the second question pretty much reiterates on what was addressed for the first question, however, for visibility, the next items answer the second question.

- Greater tool accessibility with plug-in architecture;
- No real learning curve, but still requires knowing the correct representation for the user-defined properties;
- Ability to remain focused on smart contract development, while always having the option to verify the defined properties;
- Assurance of safety for used-defined properties in case of positive feedback, and having clear counterexamples for when feedback is negative.

4.2 Implementation Roadmap

Having the end goal set, a plausible estimation of the development process of the solution can be made. May the table [4.1](#) become a general guide for the tasks ahead for the implementation phase of the dissertation.

4.3 Conclusion

This section engaged in specifying the underlying problem statement that this work proposes to address, as well as a realistically plausible solution for addressing it. At the end of this chapter, an estimation of the roadmap is made, comprising the tasks ahead, in order to completely fulfill the implementation of the proposed solution.

Task	Description
T_0	Stay alert on any relevant articles on Tezos and smart contract security research scene.
T_1	Acquire a practical understanding of implementing formal verification processes.
T_2	Acquire a practical understanding of <i>WhylSon</i> , mainly batch mode usage.
T_3	Learn how to develop a generic extension for Visual Studio Code.
T_4	Specification and design of the tool proposed.
T_5	Implement the proposed solution (now with due preparations).
T_6	Use formal verification methods and testing to grant reliability, performance, and effectiveness of the implemented solution.
T_7	Write the dissertation paper, upgrading the current one with the chapters for the tasks above (if relevant enough).

Table 4.1: Task roadmap for implementation of the solution proposed.

Chapter 5

Implementation

The fifth chapter in this document will expand upon the implementation process of the solution teased throughout the entirety of the document, which was refined in section [4.1.2](#). Section [5.1](#) presents the tools and technology used, while section [5.2](#) describes the solution's development process.

5.1 Tools & Technologies Used

Before diving into the implementation process, it is necessary to understand the foundation upon which the extension is built. The solution is a Visual Studio Code extension written in the Typescript language, making extensive use of the *vscode* [API](#). The extension interacts with Tezos' smart contract language, *Michelson*, and *LIGO*, a higher abstraction programming language that compiles to the first. An external call to the *WhylSon* tool is made via the extension, allowing the formally specified Michelson contracts to be verified against automatic provers in *Why3*.

5.1.1 Michelson

Michelson is a **Turing-complete** Domain-Specific Language ([DSI](#)) inspired by classic functional programming languages such as *Forth*, *ML* and *Scheme*, and designed specifically for and with the Tezos blockchain, the means by which this network performs on-chain computation. Just as the platform itself, the language was built from the ground-up with security by construct ideologies in mind — operating under a **stack-based approach** (last-in-first-out fashion), value immutability, and also having particularly strict **static type checking**. Along with garbage collection, these key features confer an incredible amount of robustness to this language's programs, minimizing the chances of *Michelson* ever suffering a runtime error due to internal execution. Figure [5.1](#) illustrates the basic structure and operations of a stack [1](#).

A stack is a linear collection of data whose basic instructions consist of *push* and *pop* — pushing allows elements to be added on top of the stack, while popping removes elements from the top as well. Most of the instruction catalog in *Michelson* employ these two constructs to form more complex ones, commonly popping elements before pushing. A simple example would be arithmetic instructions, which consume the two topmost elements and insert the instruction result at the top. Other than push and pop, more constructs exist to facilitate processing and manipulation of data [2](#), for instance swapping element positions, locking the topmost elements, or even function declaration/application.

¹<https://www.opentezos.com/michelson/smart-contracts>

²<https://tezos.gitlab.io/michelson-reference/>

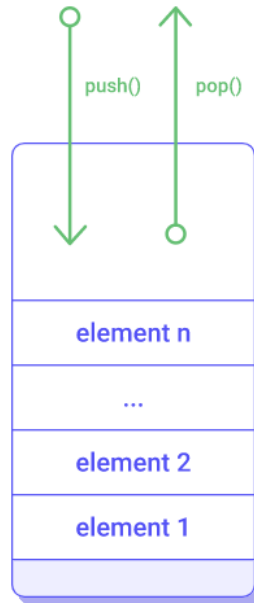


Figure 5.1: A generic stack structure with elements and operations.

However, *Michelson*'s stack is incomplete without types. Instructions require not only their inputs to match certain types, but these must also respect precise ordering, much like completing a puzzle. If these conditions are not met, the type checking of the program fails and the execution of the smart contract stops. Nonetheless, the blockchain only accepts contracts that have successfully passed the type check, meaning developers will always have to work towards a working contract — it completely negates the possibility of an on-chain runtime error due to an instruction being greeted by a stack of unexpected length, contents, or element types. In other words, program states are fixed and monomorphic in respect to their types, making internal execution deterministic.

Despite this, there are some cases in which the execution of a contract may result in failure. External transactions have greater margin of error, for these have no way to be verified by the type check. For instance, a transaction that incorrectly specifies a contract's address and/or contract type, and an incorrect entrypoint and/or argument for a contract. Yet, there are also failures that occur during a contract's own execution, such as it running out of gas, having insufficient amount of tokens for an operation, and the program programmatically fail through a `FAILWITH` instruction. Regardless of the failure that occurs, when it happens, none of the contracts involved end up in an inconsistent state — internal and external failures both result in execution and effects being reverted to the contract's last consistent state.

On one hand, developers might find the act of programming in *Michelson* to be somewhat restrictive, unlike the commonly used store-based languages. On the other hand, this restriction, along with value immutability, greatly contributes to simpler and clearer language semantics, making the language easier to reason with.

A *Michelson* program is composed by three distinct sections. Code fragment [5.1](#) illustrates the basic structure of a *Michelson* program.

```

1 parameter nat ;
2 storage nat ;
3 code { UNPAIR ;
4     ADD ;
5     NIL operation ;
6     PAIR ; }

```

Listing 5.1: Example of a simple *Michelson* program.

The three sections play a vital role in understanding how smart contracts work in the Tezos blockchain.

- **Parameter** — Describes all the contract’s possible entrypoints, and respective arguments. It can range from singular, simple values, to multiple, complex values. The `or` type is made of ample use in this section due to it behaving exactly like a *Variant* type;
- **Storage** — Describes the contract’s persistent memory space associated with it when deployed. It can range from simple to complex structures of data. The `pair` type is made of wide use in this section due to the ability to nest fields within it, similar to a *record* type;
- **Code** — Constitutes the actual code of the smart contract. It is a finite series of instructions that are run in sequence, with each instruction receiving as input the previous stack state, resulting in a new one.

A contract’s initial storage value is specified during origination, while parameter values are specified in each entrypoint call to a contract. The persistent storage will only be updated after a successful execution, whether the value before the current transaction is different or not. Other than having their storage updated, contract’s also have built-in information regarding other variables of interest that might be updated on transactions, such as its balance.

A well-typed *Michelson* program implements a function which transforms a *pair* of values representing the transaction parameter and the contract storage, and returns a *pair* containing a list of operations and the new storage. Upon terminating execution, the operations stored in the list are initiated via transactions to other accounts or contracts. Furthermore, just like in Ethereum, *Michelson* is restricted by gas usage — every instruction has a gas cost associated with it, with the amount spent somewhat relative to the complexity of the instruction.

5.1.2 LIGO

The *LIGO* language is one of the three high-level languages officially recognized by Tezos. Developers implement their contracts in *LIGO* and compile them down to *Michelson*, a typical developer workflow. It was designed to fulfill the security promises of *Michelson* by softening some of its constraints while also adding features that are seen in more conventional languages. One of this language’s trademarks is the option to use different syntaxes — *PascalLIGO*, *CamelLIGO*, *ReasonLIGO*, *JsLIGO* — in an attempt to make the development more familiar for developers already used to a certain dialect.

Though this flexibility is a great feature for attracting developers from varying backgrounds, it may conversely become a pitfall, since developers are highly susceptible to misleading and misunderstanding concepts that may be deemed as present since, after all, the language is “similar” to a language one may be familiar with. Moreover, while *LIGO* is considered a functional language, some syntaxes allow the use of imperative style approach for programming, negating the immutability feature, at least in the front-end of the compiler.

Nevertheless, the *LIGO* compiler is strict, meaning any erroneously preconceived notions given by the familiar syntax will be forcefully deconstructed at contract compilation time. The language is made in such a way that a contract that is correctly implemented in the various dialects will always generate the same *Michelson* contents. Ultimately, developers have the freedom to pick the syntax they prefer to work with, since there are no drawbacks in choosing one over the other.

LIGO satisfies a good middle ground of code simplicity and clarity without compromising security, which are not satisfied by *Michelson*’s stack-based approach — the latter’s programs became more difficult to interpret, both read and write, as contract complexity increases — garbage collection is one common feature, but the former has presented solutions to various of the latter’s drawbacks while also adding some utilities:

- **Store-based language** — Allows binding expressions to identifiers, contrary to *Michelson* that requires the developer to always keep track of the stack’s state.
- **Type inference** — The language is statically typed with a degree of type inference. When the compiler does not infer type during compilation, it has to be specifically annotated;
- **Segregation of code** — Implementation can be separated into various files. Compilation still results in a single *Michelson* file;
- **Utility API** — The language has an extensive [API](#) that provides functionality to manipulate and process simple data and data structures, and also invoke relevant environment variables;
- **Test environment** — Provides a built-in unit testing framework, simulating the Tezos on-chain environment, to test functionality in contracts;
- **Modules** — Functionality and logic can be encapsulated through modules, and consequently used in other scopes, or even files through a typical *include* (à la C);
- **Package management & versioning** — Recently added, package management allowing for *LIGO* code to be easily shared and maintained with the community.

Code fragment [5.2](#) presents the same program as presented in [5.1](#), demonstrating a basic *LIGO* file.

```

1 type parameter = nat
2 type storage = nat
3 type _return = (operation list) * storage
4
5 let main (p,s : parameter*storage) : _return =
6     let ns : int = p + s in
7     ([ : operation list), ns

```

Listing 5.2: Example of a simple *LIGO* program.

It has become best practices in *LIGO* to declare the contract’s parameter, storage, and return types, much like *Michelson* itself.

5.1.3 Visual Studio Code API

Visual Studio Code is one of the most renowned tools of the text editor scene. It is a free, multi-platform, open-source, lightweight yet powerful text editor created by *Microsoft* developers for developers. These traits contributed to the tool remaining competitive with other text editors, however, it was given the edge by its critically acclaimed community driven aspect, the community that has given it much care through the development of everyday-use extensions — the customizability provided by extensions allows developers to envision and develop their own tools for the job, sharing them with the entire world in the process through publishing.

According to Stack Overflow’s 2022 survey statistics³, Visual Studio Code is the leading code editor for developers, as illustrated in figure 5.2.

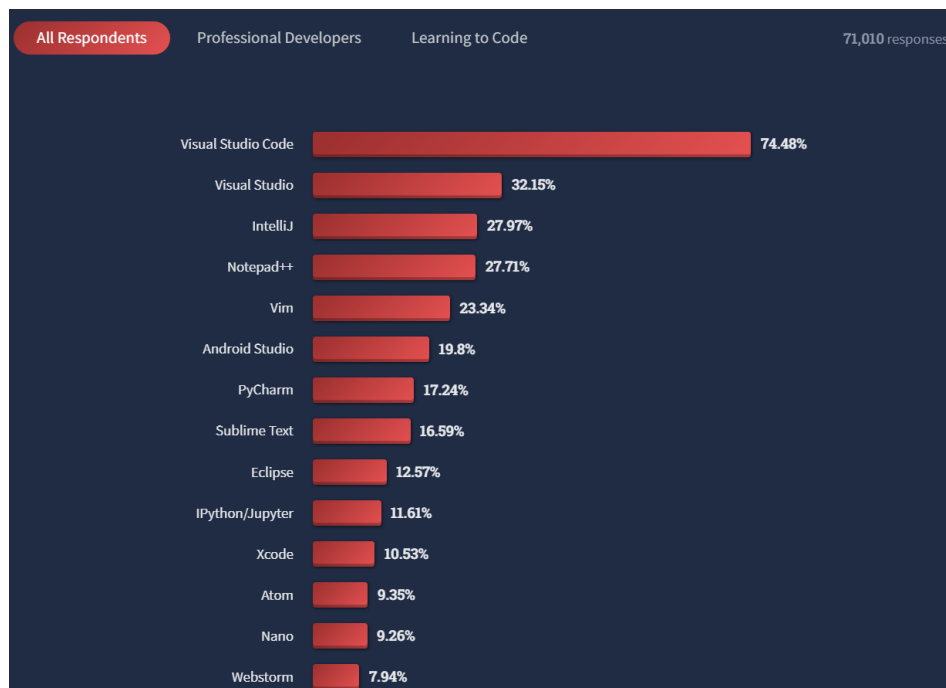


Figure 5.2: Stack Overflow 2022 survey statistics for “IDE” topic.

³<https://survey.stackoverflow.co/2022/>

Most Integrated Development Environment (IDE)s are confined to a field, a specific workflow with a specific language. Customizability through extensions and user experience distinguishes this tool apart from its competitors, since it has evolved into a hub for various distinct workflows, fields, and language supports to intersect.

Visual Studio Code's extension API⁴ is what allows developers to thoroughly tap the tool's capabilities. For starters, the documentation reference is clean, accessible, and detailed⁴, moreover, it features a GitHub repository holding numerous extensions samples⁵, allowing anyone to simply have a hands-on experience without fully committing to starting an extension from scratch.

The editor was designed with extensibility in mind, with many of the tool's components customizable and augmented through the extension API. As a matter of fact, the team integrated the tool's core features through extensions by using the extension API itself. One of its built-in features is the support for the *JavaScript* language, as well as its reliable superset *TypeScript* (also developed by *Microsoft*), which are the languages through which one can access the extension API. The language support provides developers many features that make developing in the supported language substantially more efficient, causing extension development particularly more accessible and easier to reason with.

The following can be accomplished via the extension API:

- Add user-defined commands;
- Customize and add functionality to new UI elements;
- Launch external processes;
- Automate tasks, such as building projects;
- Create a custom editor/window or even integrate an HTML page through webview;
- Upload new themes/icons for the editor;
- ...
- And much more, which can be seen in the extension API documentation.

5.1.4 WhylSon

This section will introduce *WhylSon*, a deductive verification tool for Tezos smart contracts [55], which has been teased across parts of this document's chapters. It is a tool that provides automatic formal verification on *Michelson* smart contracts through the means of its underlying framework, *Why3*. That is, *WhylSon* is built on top of *Why3*'s foundation, making use of its powerful verification functionalities that would otherwise be inaccessible for the *Michelson* language. In figure 5.3, it is seen *Why3* graphical user interface opened with a *Michelson* program. The left side of the interface is dedicated to the program's verification conditions generation.

⁴<https://code.visualstudio.com/api>

⁵<https://github.com/microsoft/vscode-extension-samples>

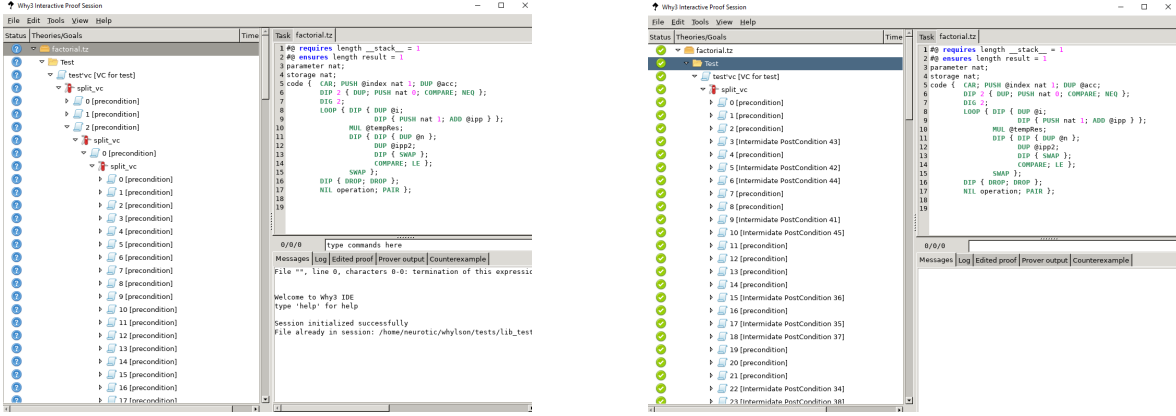


Figure 5.3: *Why3Son* using *why3-ide* for a verification session of a *Michelson* Program.

Why3 uses its own programming language, *WhyML*, which is highly specified and geared towards deductive verification. In that regard, *Why3Son*'s sole objective is to successfully generate a *WhyML* file that is equivalent to the desired *Michelson* input program. The process in question is not so linear, as there is a considerable amount of effort made by the tool so that program equivalency can, in fact, be achieved:

1. Parse deductive verification specific artifacts, in the form of *pre/post* conditions and *variant/invariant* from *Michelson*;
2. Translate *Michelson* program, plus artifacts, into a *WhyML* program through the *Why3 API*;
3. Emulate *Michelson*'s types and semantics in *Why3*;
4. Apply provers in the newly equivalent *WhyML* program.

For more detailed description of each of these steps, refer to the literature. Basically, *Why3Son* works as a plug-in to the *Why3* framework that allows *Michelson* programs to be formally verified through equivalency. Code fragment 5.3 presents the results of *Why3Son*'s translation process from the code fragments shown in 5.1 and 5.2.

```

1 use axiomatic.AxiomaticSem
2 use dataTypes.DataTypes
3 use seq.Seq
4 use int.Int
5 let contract (__stack__: stack_t) (__fuel__: int) : stack_t
6   requires { (length __stack__) = 1 }
7   requires { __fuel__ > 0 }
8   requires { (typ_infer (d (__stack__[0])))
9     = (Pair_t (Comparable_t Nat_t) (Comparable_t Nat_t)) }
10  ensures { (length result) = 1 }
11  ensures { (typ_infer (d (result[0])))
12    = (Pair_t (List_t Operation_t) (Comparable_t Nat_t)) } =
13  let __stack__ =
14    let __stack__ = unpair __stack__ __fuel__ in

```

```

15     (let __stack__ = add __stack__ __fuel__ in
16     (let __stack__ = nil_op __stack__ __fuel__ Operation_t in
17     (pair __stack__ __fuel__))) in
18     __stack__

```

Listing 5.3: Example of a *WhylSon* generated *WhyML* file.

Although the tool’s functionality is promising, there are some lackluster aspects that are hindering its use. Conclusive remarks on the tool’s paper points to the existence of performance issues for more complex contracts, as well as the fact that developers would still require to manually write the verifications artifacts themselves, which may not be as intuitive as expected unpracticed deductive verification users.

Aside from functionality difficulties, its accessibility is slightly hampered — directly using *Michelson* as its input program makes it fairly awkward for the common developer to use, especially as Tezos’s smart contract developers have better options to implement their smart contracts on, such as *LIGO*. Though, the fact that this tool employs *Michelson*, any higher-level language can utilize *WhylSon* as an intermediary step for verification, which is precisely the approach this dissertation is walking towards. In that regard, it would be interesting to see any of the available language platform incorporating *WhylSon* into their testing infrastructure.

One other glaring issue that *WhylSon* suffers from is its installation process. The tool is distributed through *opam*, *OCaml*’s package manager, which is not the issue in of itself — the codebase demands a considerable amount of dependencies that are very sensitive to conflicting with version mismatching, whether from other co-existing packages, the manager’s version, or even *OCaml*’s version. This is a technical aspect that must be streamlined and automated, since this type of issue can be very off-putting to developers, decreasing adherence before ever attempting the tool’s features.

Additionally, it is fair to say that the latest paper on *WhylSon* does not cover its latest developments. Without relaunching a paper that is up to *WhylSon*’s current status, it becomes difficult for it to become visible and feasible to the community. Overall, the tool shows great promise, but still requires polish on for it to become a standard on itself.

5.2 Extension

This section will describe the efforts and technical process leading up to the solution’s conception, introduced in section [4.1.2](#). Reiterating on the core thematic of this dissertation, discussed throughout the document, what this dissertation seeks to provide is not a verification tool in itself, but in fulfilling to achieve a method in which smart contract verification becomes more approachable to the common developer.

5.2.1 Prelude

Before getting more in-depth on the solution’s design, the author dedicated some time on studying on Tezos’ smart contract languages. Due to *Michelson*’s nature, it is best if the solu-

tion were to work directly with one of its higher-level equivalent. At some point, an opportunity to do practical smart contract development arose, by the form of a workshop, arranged by *nomadic labs*⁶. After acquiring hands-on experience with smart contract development in Tezos, it was concluded that *LIGO* would become the targeted language for working with the solution, due to the author's personal preference in its simplistic nature and shared similarities/inspiration in OCaml, one's language of mild comfort.

Though a wide variety of smart contract verification tools exist, as seen in section 3.3, the fact remains that security/verification is not properly coupled into the smart contract development workflow, else the blockchain exploits would have not been present. Most of the literature's future directions, discussed in section 3.4, point towards developing better tools and their integration in the development cycle — the former is being worked on, with the literature expanding every year with a repertoire of new tools, while the latter seems to be neglected — answers to the smart contract security problem exist, but the delivery is not being properly done to the smart contract development community.

Therefore, solution's approach sprouted from the thought of:

How do I make existing verification tools more accessible to developers? Additionally, why is adoption of these tools difficult?

Upon reflecting on the matter, the pieces started to fit naturally. The question is not simply directed at them, developers, but also at oneself. For some time, answers were sought out on the outside, when all it required was to look inwards — the problem had to be faced from the common developer's perspective, that is, the author himself — the design of the solution were to become something that the author would find truly useful and something he would use in the future, were he to work in smart contract development area.

From personal experience, Visual Studio Code editor has been a reliable workbench for several years, akin to a table large enough to fit all the tools one requires and many more. The aforementioned workshop participation incited a search for Tezos extensions in the tool's marketplace, which revealed positive — *LIGO* has its own extension that greatly improves developers' experience in this platform — however, there were no existing security or verification related tools. That is to say, a Tezos smart contract security related extension has yet to be released on Visual Studio Code, which means this dissertation presents a novel feature for Tezos smart contract developers that use this platform.

5.2.2 Design

The foundation for this extension lies in the **interactions between a *LIGO* source file and its *Michelson* counterpart**. A large section of the code is present in order to maintain a unidirectional cohesion from the source to the compilation, but also for this to remain an entirely optional feature.

The inspiration for this relationship lies in the critically acclaimed *godbolt*⁷ online compiler and Markdown extension in Visual Studio Code. These tools take advantage of the user hav-

⁶<https://research-development.nomadic-labs.com/training.html>

⁷<https://godbolt.org>

ing visibility of both a source file and a compiled file, where changes done in the source are reflected on the compiled counterpart. Though, this change is not always immediate — the online compiler triggers compilation after no changes are detected on the source file within a specified amount of time, normally between 0.5s, while in *Markdown*'s case, its engine processes the raw content immediately.

This feature allows users to have a glass window over the compiler's work, making it an interesting feature for those who are seeking to improve their program's efficiency, for the former, or to see formatted *Markdown*, for the latter. Figure 5.4 presents the appearance of both aforementioned tools.

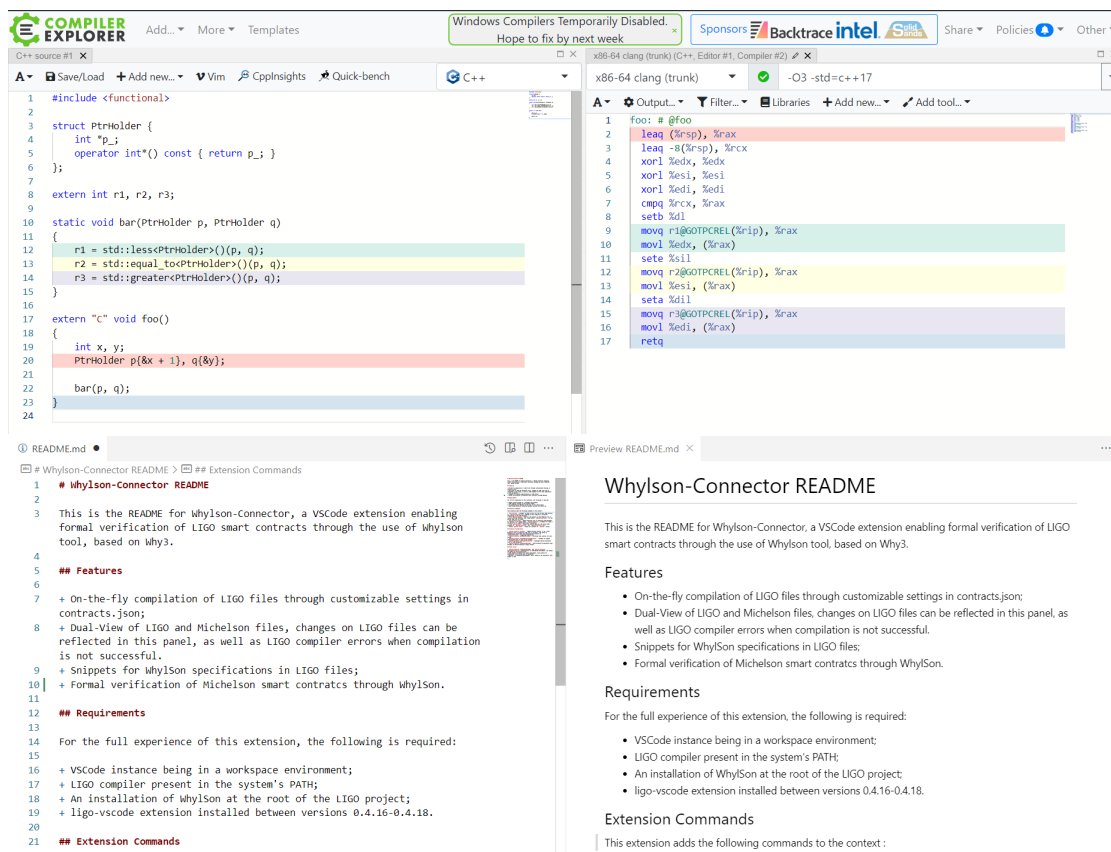


Figure 5.4: Design of *godbolt* online compiler and *Markdown* extension, respectively.

The extension's appearance has now reliable references, but there is still more substance under the hood of these tools that needs to be adapted to this dissertation's solution. Doubts and issues will start to appear as implementation advances, however, for this document, not every implementation detail will be documented. For the entirety of the implementation, it is used the Visual Studio Code [API](#) using *TypeScript* as its language. The following points are the relevant implementation details.

1. **Automated Compilation** — Automation of *LIGO* source file compilation;
2. **Workspace Specific Extension Folder** — Implement data persistency across project workspaces;

3. **Michelson View** — Show the compiled *Michelson* of the focused *LIGO* file, changes in second are reflected in the first;
4. **WhylSon Annotation Snippets** – Small code templates for potentially reusable code, these snippets will semi-automate the input of verification conditions, though the user will require to specify values and/or formulas;
5. **Launching WhylSon** — Launch a detached WhylSon process in which contracts can be submitted to formal verification using why3-ide.

5.2.2.1 Automated Compilation

The first highlighted feature is automated compilation. It is a feature of extreme importance due to the fact that the compilation of *LIGO* source files would always require the input of more than the file — the file itself, and an entrypoint — specifying the latter every time one were to compile the source would severely deteriorate the experience of the developer, meaning, there is most definitely an optimization to be made along this path.

Referencing the documentation from the *LIGO* compiler:

```
ligo compile contract SOURCE_FILE
```

Compiles a contract to Michelson code. It expects a source file and an entrypoint function that has the type of a contract:

```
parameter * storage → operationslist * storage.
```

Although not shown, this command can also accept numerous other compilation flags, also, the entrypoint argument is not present in the command. It is implied that, when not specified, the entrypoint will be “main”, which has become the standard for the language. However, anyone can still change their entrypoint name, as long as it conforms to the contract type.

Such task can be automated with relative ease, however, developers might feel a need to change their entrypoint name and modify other additional flags. This becomes an issue, for with this command alone there is no flexibility. This was one of the motivations for adding some data persistency to the extension’s context.

5.2.2.2 Workspace Specific Extension Folder

The workspace specific folder is the manner in which the extension acquires persistency on contract metadata and the *Michelson* contracts themselves, after they are successfully compiled. A plausible file structure for a *LIGO* project using this extension could be as shown in figure [5.5](#).

To better understand the workspace specific folder, it is best to dissect the example.

- **.whyson** — This is the folder that is specific to the extension, automatically generated in the workspace root of the project once the extension is activated. There are two entities present, a directory for hosting the compiled contracts and a file. The


```
Ligo-Project/
|-- .why1son/
|   |-- bin-contracts/
|   |   |-- contract1.tz
|   |   |-- contract2.tz
|   |   `-- ...
|   `-- contracts.json
|-- src/
|   |-- contract1.mligo
|   |-- contract2.mligo
|   `-- ...
|-- why1son/
|   `-- Why1Son-project-files
`-- other-project-files
```

Figure 5.5: Plausible *LIGO* project using proposed solution.

`contracts.json` file is a JavaScript Object Notation (**JSON**) format file that contains metadata on every unique contract that has been at least successfully compiled once through the extension. Code fragment 5.4 presents the type of objects that are within the file (in TypeScript notation). The file is considered as a list of objects of this type.

```
type ContractEntry = {
  title: string,      // Short, identifiable title for smart contract
  source: string,     // Absolute path for LIGO source file
  onPath: string,     // Absolute path for compiled Michelson file
  entrypoint: string // Specified entrypoint for contract compilation
  flags: string[]    // Additional flags array
}
```

Listing 5.4: Structure of an entry in `contracts.json` file

Through this entry, all the information is needed to compile a *LIGO* source file, while at the same time granting persistency and flexibility to the compilation, since its readability allows the user to modify any entry as they see fit. However, such is not wholly recommended, as there are flags that might not work as intended with the extension, such as the output flag.

The subfolder `bin-contracts` is simply where the compiled contracts are being held. Through this folder, they are programmatically fetched for either presentation as a read-only document, or for launching a *Why1Son* session with.

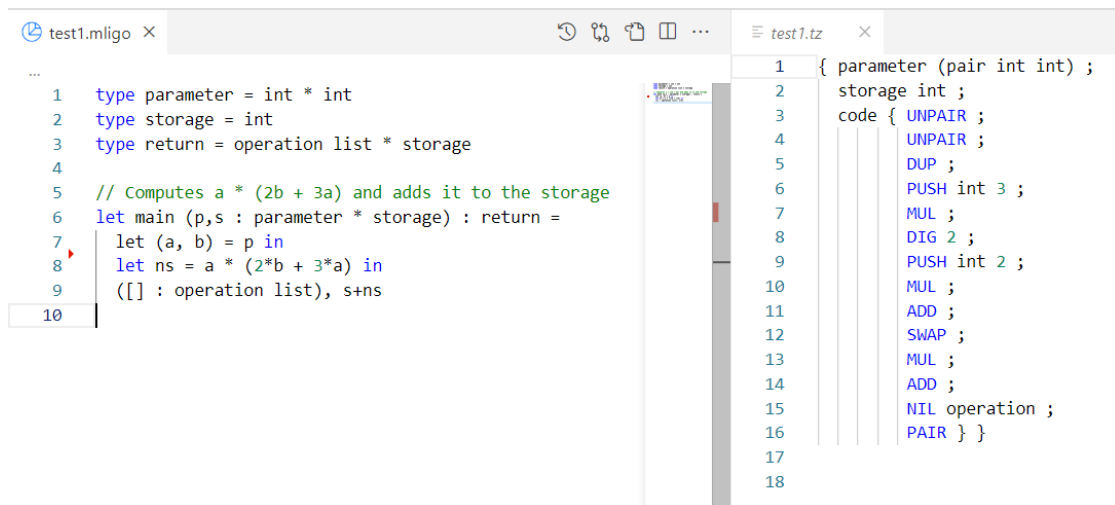
- `src` — This folder is dedicated to hosting *LIGO* source files. Project dependent, file hierarchy within should not influence extension;
- `why1son` — A local installation of the *Why1Son* tool. Although the tool uses the *opam* package manager, it is yet to be added as one. It provides some of the contents that are

required to launch the tool. Folder should not be manually modified.

The extension specific folder is crucial for the extension to maintain cohesion, meaning if it is ever manually removed, some problems might ensue. In case of remaking or just deleting specific sections of the folder, the extension provides commands either to remake the entire folder, resetting all contents, or even remove specific contracts from the contract entry file.

5.2.2.3 Michelson View

This feature is a centerpiece for the extension, it is what allows it to resemble the tools of reference. Tezos smart contracts require every optimization possible for a production environment, since it not being optimized may very easily sway the economic aspect of an entire organization or entity. Thus, the **LIGO-Michelson Dual-View** becomes specially important for developers who are interested in keeping their smart contracts optimized by always having their *Michelson* contract visible, updating in real time to the response of changes in the respective *LIGO* source file they are developing on. Figure 5.6 illustrates the design for the Dual-View (due to the nature of the exposition medium, it becomes impossible to feature the dynamism of real-time file updates).



```
test1.mligo x
...
1 type parameter = int * int
2 type storage = int
3 type return = operation list * storage
4
5 // Computes a * (2b + 3a) and adds it to the storage
6 let main (p,s : parameter * storage) : return =
7   let (a, b) = p in
8   let ns = a * (2*b + 3*a) in
9   ([ : operation list), s+ns
10

test1.tz x
1 { parameter (pair int int) ;
2   storage int ;
3   code { UNPAIR ;
4         UNPAIR ;
5         DUP ;
6         PUSH int 3 ;
7         MUL ;
8         DIG 2 ;
9         PUSH int 2 ;
10        MUL ;
11        ADD ;
12        SWAP ;
13        MUL ;
14        ADD ;
15        NIL operation ;
16        PAIR } }
17
18
```

Figure 5.6: Simple example of **LIGO-Michelson Dual-View**.

However, before this view makes its entrance, the *Michelson* counterpart of the focused *LIGO* file must exist in the `bin-contracts` folder, otherwise, a first compilation process is initiated that must result in a successful compilation. Only then, will the Dual-View become available to the *LIGO* source file in question.

This aspect of the extension is possible due to the virtual document **API** — the *Michelson* file located on the right tab in figure 5.6, is not exactly a file located in a physical medium, but an entity in memory. A feature for these entities is that their content cannot be manually changed by the user, making it a **read-only document**. Content can only be changed through the usage of the **API**, exactly what was required in order to maintain the one-way relationship from *LIGO* to *Michelson*. After all, if changes were to be made in the latter, they

would either be rather difficult to edit on the *LIGO* source, or the modified content would simply be overwritten by the next compilation of the former.

Early prototyping for this feature used the conventional file through the file system [API](#), which caused optimization concerns due to an overhead of Input & Output [IO](#) operations on disk. In other words, every *LIGO* compilation resulted in the creation of a new file in the `bin-contracts` folder, overwriting existing ones, and consequently read and displayed on screen. The virtual document [API](#) optimized this process by allowing content to be directly loaded into memory — when the output location is not specified in the compilation command, the result is found within the `stdout` channel, which can be read as a `string` — this way, compilation does not result in writing a file on disk, but to simply result in a `string` which can be directly loaded onto the virtual document. This adjustment resulted in very minimal usage of [IO](#) operations for the Dual-View feature.

The rate at which the *Michelson* material is displayed on screen is another element at this stage of development. Through trial and error, the *godbolt*'s approach worked better, as compilation and consequent display could not keep up with the normal rate of document change (keystrokes). In that regard, a compilation throttling mechanic was implemented, in which the compile & display process is only triggered after the extension does not detect changes within the *LIGO* source after a specified time interval has elapsed. The threshold value can be changed in through the extension's configuration settings, but its base value is set to 0.75 seconds.

Another neat detail that the Dual-View provides is showing compilation errors. Normally, errors are shown in others [UI](#) elements within the code editor, but this dynamism allows to users to always be on the lookout for their code. An example is shown on figure [5.7](#).

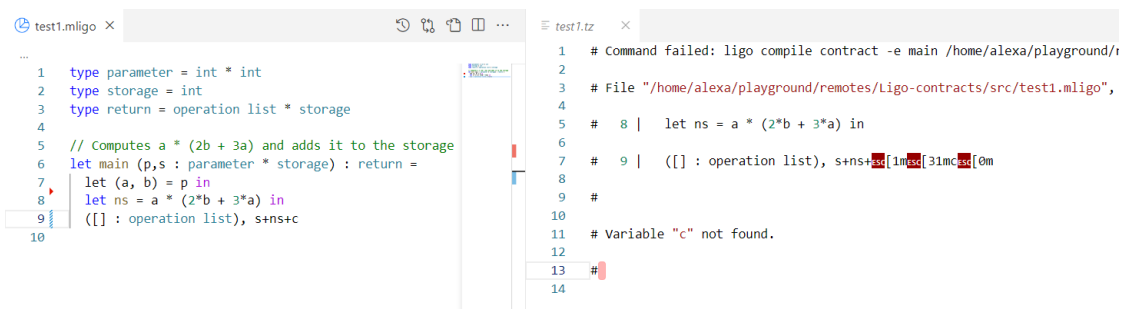


Figure 5.7: Dual-View showing a compilation error.

Lastly, the Dual-View feature is entirely optional. For example, if the *LIGO* source file is not supposed to have an entrypoint, that is, the file is not supposed to behave as a stand-alone contract, but as a module that will be included in other *LIGO* files instead, then all it takes is to ignore the activation process of the Dual-View. Additionally, even if the *LIGO* source file is a contract that has already been compiled through the extension's functionalities, developers can also ignore the automatic compilation feature through the extension's settings.

5.2.2.4 WhylSon Annotation Snippets

Snippets are a form of semi-automatic insertion of code fragments, often used when the structure of the fragment in question is routinely made use of. A good example, for most conventional languages, are snippets for control statements and function definitions — instead of manually typing the entire sequence, snippets generate the structure, leaving a cursor placeholder with tab-cycling for the sections in the fragment that require the user’s input.

In order to facilitate the user’s experience with inserting *WhylSon* artifacts in their contracts, snippets were implemented. For the remaining of this solution, the values that are to be worked with on snippets are the numerical (`int`, `nat`, `tez`) and string-based (`string`, `address`) values. Figure 5.8 presents all the available snippets. In Visual Studio Code, to invoke the snippet context, it is normally typed `Ctrl + Space` (in Windows/Linux), followed by typing a character that matches the snippet prefix.

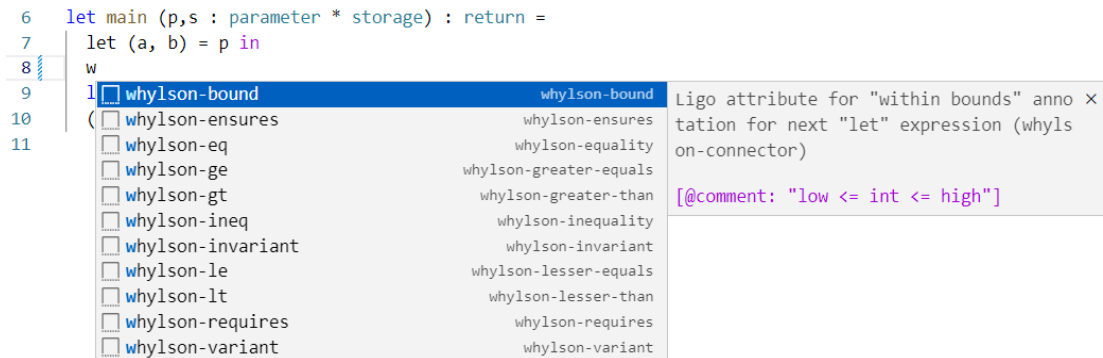


Figure 5.8: Snippets available with the extension.

However, one important issue is brought up with this feature — how do these verification artifacts carry over to *Michelson*? — suffice to say, this matter should be tended to by the *LIGO* compiler. The manner in which these snippets function is through *LIGO*’s attributes, in this case specifically, comment attributes. These constructs signal the compiler that whatever is within the delimited quotation marks are to be inserted verbatim in the generated *Michelson*.

Every *WhylSon* annotation snippet comes confined within the *LIGO* comment attribute. Though, these comments have to follow a specific pattern according to the next or previous `let` expression. Code fragments 5.5 and 5.6 are the same contract, the first is the *LIGO* implementation, the second its compiled output. Both exemplify the usage of these snippets and *WhylSon* annotations, and how these are inserted in contract code.

```
// Compute a * (2b + 3a)
type parameter = int * int
type storage = int
type _return = (operation list) * storage
let main (p,s : parameter * storage) : _return =
  let (a, b) = p in
  [@comment:"@ 0 <= int <= 100"]
  let ns = a * (2*b + 3*a) in
  ([ : operation list), s+ns
```

Listing 5.5: *Why3* annotation snippet example.

```
{ parameter (pair int int) ;
  storage int ;
  code { UNPAIR ;
        UNPAIR ;
        DUP ;
        PUSH int 3 ;
        MUL ;
        DIG 2 ;
        PUSH int 2 ;
        MUL ;
        ADD ;
        SWAP ;
        MUL ;
        # @ 0 <= int < 100
        ADD ;
        NIL operation ;
        PAIR } }
```

Listing 5.6: *Why3* annotations in *Michelson*.

The detail here lies in that the *LIGO* **comment attribute is being placed above** the "let ns = ..." expression, while on the right, the *Why3* **annotation is inserted right after the bound expression** to the identifier ns is computed in *Michelson*. Having the annotation inserted right after the expression is formalized makes it so that, during analysis, the top of the stack contains the value bound to the ns expression, granting *Why3* a context advantage when it comes to verifying the specified property.

Although this is the case for these annotations, there are two special cases in which they are supposed to be inserted before the expression, rather, the entrypoint function. In *LIGO*, the `requires{}` and `ensures{}` artifacts are inserted above the entrypoint function, in *Michelson*, these should be outside the contract code section. An example of these artifacts' usage can be seen in code fragments [5.7](#) and [5.8](#), for *LIGO* and *Michelson*, respectively.

```
// Compute a * (2b + 3a)
type parameter = int * int
type storage = int
type _return = (operation list) * storage
[@comment:"@ requires{a > 0 /\ b > 0}"]
[@comment:"@ ensures{result >= 0}"]
let main (p,s : parameter * storage) : _return =
  let (a, b) = p in
  [@comment:"@ 0 <= int <= 100"]
  let ns = a * (2*b + 3*a) in
  ([ : operation list), s+ns
```

Listing 5.7: *Why3* annotation snippet for contract with pre and post conditions.

```
# @ requires{a > 0 /\ b > 0}
# @ ensures{result >= 0}
{ parameter (pair int int) ;
  storage int ;
  code { UNPAIR ;
        ...
        MUL ;
        # @ 0 <= int < 100
        ADD ;
        NIL operation ;
        PAIR } }
```

Listing 5.8: *Why3* annotations in *Michelson* with pre and post conditions.

This structure shares similarities with the *Why3* tool, where pre- and post-conditions are

also specified outside the code itself.

The entire set of snippets provided by the extension can be seen on figure [5.8](#), but the next points summarize their usage.

- Comparative binary operators — "(In)Equals", "greater/lesser (equals) than", and a range operator that enclaves a numeric value between a specified lower and higher bounds;
- Pre- / Post-conditions — `requires{}` and `ensures{}`;
- Variant/Invariant specification — `variant` and `invariant`.

5.2.2.5 Launching WhylSon

Having a formally specified contract, *WhylSon* now requires delivering upon its functionality — having a specified *Michelson* contract, it can formally verify its properties by confronting the specifications with automatic provers. Unfortunately, the entire extension experience cannot be present solely within the Visual Studio Code editor, since *WhylSon* does not work without the graphical interface.

With an installation of *WhylSon* in the root of the *LIGO* project, a *WhylSon* proof session can be started with the following command:

```
why3 ide -L ./whylson/lib [path_to_michelson]
```

Where `path_to_michelson` is automatically filled with the absolute path to one of the files contained within the `bin-contracts` folder. The extension provides an existing command in the instance context, requiring only for a *LIGO* source file to be focused. Once the command is executed, the extension compiles the source, creating a new file in the `bin-contracts`, and launching *WhylSon* with said contract.

5.2.3 Development Loop

Now that the entire extension has been detailed, this section will propose how the development loop with this extension installed can be approached. Assuming Visual Studio Code, *WhylSon*, and *LIGO* have a working installation on the machine... additionally, the *WhylSon* installation must remain located on the project's root.

1. Install the extension through the marketplace;
2. Open or create a *LIGO* development workspace;
3. Open a *LIGO* source file. The extension begins its activation routine once it detects that a *LIGO* source file has been opened, generating a `.whylson` folder with its respective contents if not already created at the project's root;

4. Implementation loop — Program in *LIGO* to the heart's content by either or not using the features provided by the extension — open Dual-View through an icon on the [UI](#) or by the command in the context and use snippets to help insert *WhylSon* specification artifacts;
5. Verification loop — Once comfortable enough with the specifications, start verifying with *WhylSon* — a graphical interface will be presented, with the *Michelson* program.

To verify a program, an automatic prover must first be selected, normally one of Alt-Ergo, CVC4, and Z3. The prover will then confront the specifications that were inserted in *LIGO* through the snippets or not, which were translated into the *Michelson* comments, and are now readable by the tool after *WhylSon* converted the contract into an equivalent *WhyML* program.

After some time, the results of the verification will be presented in the *why3 ide* interface, showing which specifications every of the specification's status — passed or failed — the latter being provided with counter-examples. Once the session is over, the implementation loop may resume, where one would start adapting the *LIGO* source file according to the feedback provided by the verification tool.

Figure [5.9](#) condenses the workflow into an upgraded version of what is seen in figure [4.1](#), that is supposed to describe the architecture and their relationships.

5.2.4 Features & Functionality

The previous section provided a narrative approach in extension's inner workings and design. This section will describe what is provided by the extension in bullet point fashion.

The extension's main features are:

- On-the-fly compilation of *LIGO* files through customizable settings in `contracts.json`;
- Dual-View of *LIGO* and *Michelson* files, changes to *LIGO* files can be reflected in this panel, as well as *LIGO* compiler errors when compilation is not successful;
- Snippets for *WhylSon* specifications in *LIGO* files;
- Formal verification of *Michelson* smart contracts through *WhylSon*.

For the user to experience the full functionality of this extension, the following is required:

- Visual Studio Code instance of version 1.67.0 and above, opened in a workspace environment;
- *LIGO* compiler present in the system's `PATH`;
- An installation of *WhylSon* at the root of the *LIGO* project;
- `ligo-vscode` extension installed between versions 0.4.16-0.4.18.

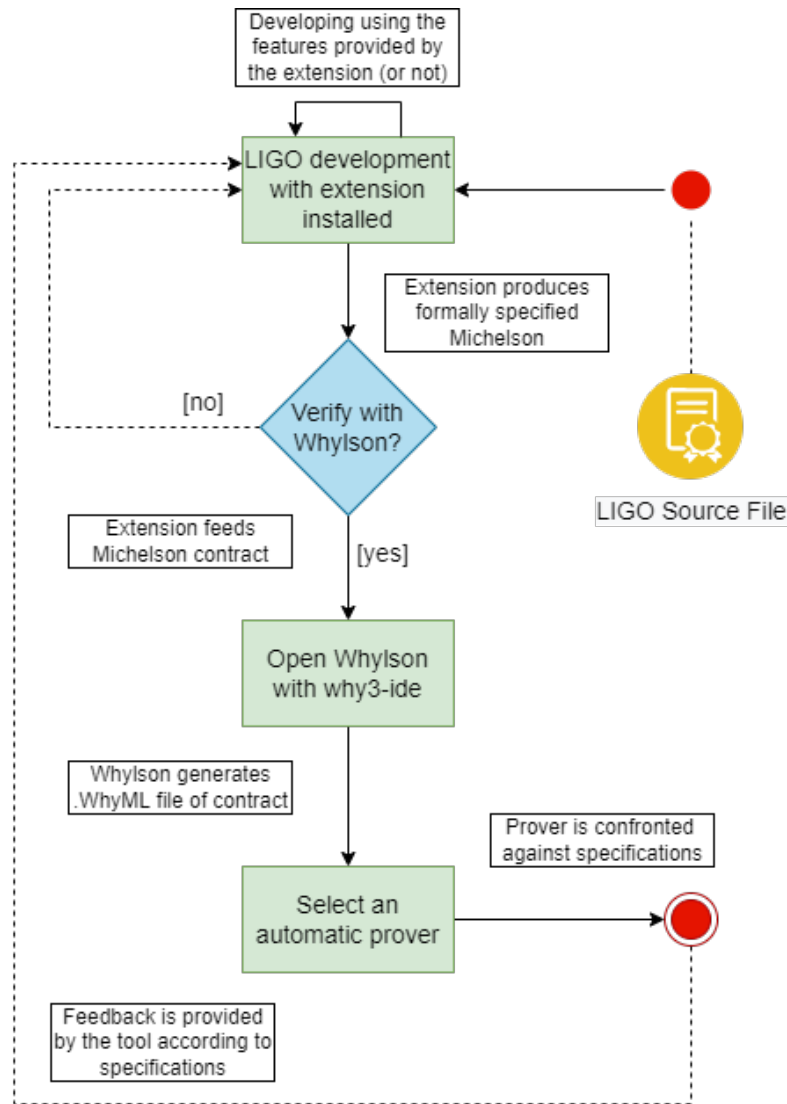


Figure 5.9: Condensed extension workflow through activity diagram.

The extension adds the following commands to the Visual Studio Code instance context:

- **Save Contract** — Attempts to make an entry for the current LIGO contract in `.why1son/contracts.json`. Making an entry requires a successful compilation of the *LIGO* document;
- **Start Whylson Session** — Starts a new process in which Whylson runs a session with the Michelson file, found within `.why1son/contracts/`, of the active *LIGO* file on screen;
- **Open Michelson View** — Opens Michelson file of respective LIGO document. If the contract is not found within `.why1son/contracts/`, attempts to create a new entry for it, opening the view if successful. This command is also available through an icon on the editor title **UI**;
- **Erase Contract Data** — Erases the contract data for the active *LIGO* document in `.why1son/contracts/` and `.why1son/contracts.json`;

- Remake `.why1son` Folder — Erases all contents of `.why1son/` folder.

The next points are relative to the possible configurations the user can make in the extension customization section in their Visual Studio Code instance.

- `autoSave` — Toggle autosave feature. If on, **and Michelson view of respective *LIGO* file is visible**, the latter is automatically saved after the specified time interval in `autoSaveThreshold`, triggering compilation;
- `autoSaveThreshold` — Throttled time interval for auto saving;
- `onSaveBackgroundCompilation` — Attempts to compile *LIGO* document even if view is not visible;
- `highlightAnnotations` — Highlight why1son annotated lines in *LIGO* documents;
- `showOutputMessages` — Have extension occasionally send messages on “Why1son-Connector” output channel.

5.2.5 Issues

The prior sections described and exhibited the potential of the extension, however, just like with any piece of software, there are some issues that refrain the proposed solution to perform as expected. The next items include main issues that are present, until the time of writing, in the proposed solution:

- Comment attribute is still undergoing development with the *LIGO* team, though the concept and mechanic has already been arranged and set in stone through an open issue in *LIGO*’s gitlab repository ⁸;
- Though the verification features in *Why1son* seem promising, it cannot be understated that the tool’s lack of redistribution is holding back its usability by Tezos’ smart contract development community.

For once, the installation process is not linear, requiring a multitude of dependencies and workaround fixes to get the tool working. It is extremely unlikely that a developer sticks and pours time into learning this tool when the first hurdle it is its installation process.

Secondly, the tool not being able to run in batch mode, unlike *Why3*. A graphical window retracts from the extension’s experience, specially when there is no feasible way for communication between both *why3 ide* and the Visual Studio Code instance. Being able to run in batch mode, or through `API` calls to *Why3*, would allow the entire verification loop and other process to be confined within the Visual Studio Code, not detracting from the development space and workflow.

⁸<https://gitlab.com/ligolang/ligo/-/issues/1447>

- Some of the extension's configuration that have been thought of are still yet to be properly implemented, namely highlighting and quick jump between *WhylSon annotations*, personally modifying the elapsed threshold for compilation to be triggered, and enabling/disabling logging (though the logging console is not intrusive).

5.3 Conclusion

This chapter was focused on dissecting the entire process that led to the proposed solution's final implementation. There was a thorough discussion of the motives, inspirations, details, features, and issues. Not every implementation detail was mentioned, since the code in itself is of little relevance for the exploration.

Chapter 6

Practicality Assessment

This section is dedicated to analysing the practicality of the proposed solution, discussed in the previous chapter. Several made up examples will hopefully shed some light on the usage, benefits, and downsides of using the extension when developing in *LIGO*. Unfortunately, for the examples shown, none can be reproduced through extension usage — it would be a difference in the hypothesis and thoughts shown if the *LIGO* annotations could carry over to compiled *Michelson*, and also if *WhylSon* could parse newly introduced verification artifacts, consequently verifying them.

It should be noted that the values chosen for the snippets are the simplest in both targeted languages — `int`, `nat`, `tez`, `string`, `address` — numerical values especially, are by far the most used for simple applications, while also being the basis for financial applications. Composite and other complex types, such as `pair`, `list` etc., do not make an appearance for this iteration of the extension, but they can still theoretically be used by *WhylSon* tool standalone.

6.1 Intra-Michelson Value Check

In the context of this work, an intra-michelson value check is the functionality of *WhylSon* asserting that the value in question is the one specified for the specific *Michelson* stack state. The provided snippets, combined with the *LIGO* comment attribute, are able to insert such annotations at the precise stack location where that value is required to be checked. Code fragment [6.1](#) shows the complete code of a contract that at the end of execution sends balance in its storage. Through this example, there is some room for *WhylSon* assertions to grant the program some reliability.

If this were in Visual Studio Code, the highlight feature on the *WhylSon* annotations would be extremely useful in order to spot them quickly amidst code.

```
1 type parameter = tez
2 type storage = unit
3 type _return = (operation list) * storage
4
5 let div_tez_decimal (a,b : tez * tez) =
6     let (q,r) = Option.unopt (ediv a b) in
7     1tez*q + r
8
9 (* Obtain p% of b in tez *)
10 let percentage_of_balance (b,p : tez * tez) : tez =
11     [@comment: "@ 0.01 <= tez <= 1"]
12     let p : tez = p in
```

```

13 (* balance * 0.01 -> balance / (1/[0.01-1]) *)
14 if p > 1tez || p < 0.01tez then
15     (failwith "Invalid percentage" : tez)
16 else
17     let inversed_p : tez = div_tez_decimal (1tez, p) in
18     div_tez_decimal (b, inversed_p)
19
20 [@comment: "@ requires{balance >= 0 /\ balance <= 10000} "]
21 let main (p, _ : parameter * storage) : _return =
22
23     (* Extract p% of current contract *)
24     [@comment: "@ 0 <= tez <= 10000"]
25     let value : tez = percentage_of_balance (Tezos.get_balance (), p) in
26
27     [@comment: "@ tez >= 0"]
28     let bal : tez = (Tezos.get_balance ()) - value in
29
30     (* Send the p% from this contract to the current transaction initiator *)
31     let op : operation = Tezos.transaction () value (Tezos.get_sender ()) in
32     ([op], ())

```

Listing 6.1: Using *WhylSon* annotations for simple value checks

The program is quite lengthy, but it wants to portray some cases in which the annotations might prove useful or hard through *WhylSon* alone. The next items describe the usage and possible thought behind the annotations.

- Line 11 — One issue that hasn't been addressed, is having access to a function's arguments in verification context. The fact that the intra-michelson annotations attach to the next `let` binding, makes it impossible to assert function argument values directly. However, one thing that may be done is rebinding (might require paying attention to variable shadowing) the argument immediately after the function declaration in a `let` binding. As a workaround it might work, but it is not experimented with. This annotation will attempt to check if the input value for the function is between the specified boundaries;
- Line 20 — This pre-condition is a *WhylSon* artifact to guarantee that whatever things it is trying to verify will be under the assumption that the contract's balance can only be between 0 and 10000. This kind of pre-condition might further help the intra-michelson annotations;
- Line 24 — The `percentage_of_balance` function fetches the `tez` corresponding to the percentage encoded for what is seemingly a decimal number — such type is not native in Tezos, but they can be emulated through the `tez` type. Through the pre-condition it assumes the input will stay within the specified range, meaning, given a percentage of the balance, should return also a value from within the pre-conditioned range;
- Line 27 — A simple check if the `tez` that going to be extracted through the operation, formalized in line 31, is non-negative. Since line 28 attempts to subtract an amount of

tez from the contract's balance, it would be wise to prevent the contract from failing its execution. Most of the time, making a stand-alone check like this might not prove sufficient for *WhylSon*, hence it might be required a set of good pre- /post- conditions artifacts to help such situations.

6.2 Deductive Verification

This next example is a classic for formal verification. We attempt to add *WhylSon* pre- /post- conditions in order for *WhylSon* to fully verify the program. First off, these annotations are generally harder to use than intra-michelson annotations, since it requires for the user to have some knowledge in deductive verification, which is a barrier in itself, but not too difficult for simple formulas. Code fragment [6.2](#) presents a *LIGO* implementation for a factorial program.

```

1 type parameter = nat
2 type storage = nat
3 type _return = (operation list) * storage
4
5 [@comment: "@ requires {p >= 0n}"]
6 [@comment: "@ ensures {result = p!}"]
7 let main (p, _ : parameter * storage) : _return =
8     let rec fact (n, acc : nat * nat) : nat =
9         if n <= 1n then acc else fact (abs (n-1), n*acc)
10    in
11    [@comment: "@ nat >= 1]
12    let r : nat = fact (p, 1n) in
13    ([], r)

```

Listing 6.2: Using *WhylSon* annotations for deductive verification - factorial.

Lines 5 and 6 show a correct usage of *WhylSon* pre- /post- condition artifacts — this program functions should function for every natural input, and given one, will always output a natural above 1 — after all, the lowest possible input would result in 1, with higher inputs being consecutively higher. While line 11 shows a possibly useless usage of an intra-michelson verification check, since the initial assertions might be enough to completely verify the program.

Figure [5.3](#) shows a factorial implementation in *Michelson*, however, it was manually implemented on a not recent version of *Michelson*, not affected by the compiler's optimizations compared to when obtaining through *LIGO* compilation. It would be interesting comparing both performance and verification assessment in *why3-ide*, since both programs represent the same computation, but with different implementation steps associated with it.

Similarly, code fragment [6.3](#) shows the implementation of a program that sums the elements of a list of natural numbers, storing it after execution, which now seems to be leaning towards abusing the current capabilities of the verification tool, however, ideally, this kind of implementation and specifications should work.

```

1 type parameter = nat list

```

```

2 type storage = nat
3 type _return = (operation list) * storage
4
5 [@comment: "@ requires {p.length >= 0}"]
6 [@comment: "@ ensures {result = List.sum p}"]
7 let main (p,s : parameter * storage) : _return =
8     let sum (acc, i : nat * nat) : nat = acc + i in
9     [@comment: "res >= 0"]
10    let res = List.fold_left sum 0 p in
11    ([], s+res)

```

Listing 6.3: Using *WhylSon* annotations for deductive verification - sum list.

Since the program deals exclusively with type `nat`, it should be easy enough for the tool to verify that any sum of the list result in an always non-negative number, that is, another natural.

6.3 Conclusion

Due to the issues presented in section [5.2.5](#), it was not possible to completely show the full potential of the extension. It is incredibly defeating to have an “incomplete puzzle” at the time of writing. Naught can be done other than imagining the remaining piece’s effects and behaviors, without fully tangible results. The entire workflow is laid out with the assumption of these features, yet, what is shown in this section are partial hypothesis on how the mentioned features may fit together into the final, “finished puzzle”. Nonetheless, the results shown are encouraging — revisiting the extension with all pieces, as future work, would be interesting to prove the hypothesis made in this section.

Chapter 7

Final Remarks

7.1 Summary

This dissertation proposal inlays the foundation of this soon to be graduation dissertation. Chapter 1 relays how this problem came to be, its severity, and general guidelines for minimizing it. Chapter 2 introduces the general knowledge on blockchain, smart contract and the Tezos platform. Chapter 3 presents an analysis of current state-of-the-art research literature on this dissertation's area. Chapter 4 develops on the problem statement, solution, and the working plan for the solution proposed. Chapter 5 details the entirety of the implementation process, starting by introducing the tools and technologies used, followed by the implementation process of the solution as well its pertinent points and issues. Chapter 6 presents practical approaches in which the proposed solution can be of help in smart contract development on Tezos. Lastly, chapter 7 presents conclusive remarks and future work for this dissertation.

7.1.1 Remarks

Blockchain technology is a phenomenon that has attracted many actors into a scene with incredible opportunity as well as steep landslides. Although it has passed its infancy, it is still a relatively new area that requires avid shaping in the security department. Such is a phenomenon still present in traditional software, where security is often an afterthought, not being taken into consideration in its earliest stages of development, however, circumstances in blockchain platforms revealed the severity of incidents when lackluster security measures are evident.

On one side it has been shown the very compelling, appetizing aspects of participating in the blockchain market, however, on the other, a small misstep is enough to jeopardize all of what has been taken for granted — smart contracts should only and only be able to fulfill its specifications, expecting no unknown behaviors — such should be the philosophy when approaching the act of smart contract development and consequent deployment, especially in financial sectors where losses are potentially more severe.

To that end, smart contract security countermeasures were devised, and many tools based on such approaches were purposely developed. Though, what was found within the literature was an over commitment and unorganized effort in solving the problem — many are the security tools developed, and while most are effective, the heterogeneity of solutions made the overall scene convoluted and confusing not only for newcomers to smart contract development but also the average non-expert in security developer. Not only that, but there has been little to no attention to the development of tools that bridge these security tools themselves

with the actual development workspaces and pipelines, making the integration of security tedious and unfeasible.

Thus, an opportunity was seen in attempting to pave a new way of confronting security integration with development, which is the main theme of this dissertation.

The proposed solution provides what seems to be a novel way to working with a smart contract verification tool — the development of a plug-in tool in one of the most used development workspaces worldwide. Through this solution, the verification tool itself will gain visibility, while also granting various benefits for smart contract developers who decide to integrate the extension into their workflow. One of such benefits is the ability to work directly with a high-level language for Tezos smart contracts that was otherwise impossible for the security tool in question. This change may also increase adoption by developers, while also boosting possible inclusion of businessman and stakeholder in directly participating in the establishing business properties in their smart contracts due to increased readability.

Hopefully, this unorthodox method inspires blockchain communities to also approach security in this manner, since it is a small yet significant security optimization through the simple fact of tools having better integration, visibility, and accessibility.

7.2 Future Work

The implemented solution contributes with something unique among the Tezos smart contract development community, though, like all software, underlying issues are deducting from the expected outcome.

- The extension is merely an extension. That is, it has limited functionality by itself. It is supposed to interact flawlessly with concerned tools. For that, the suggested is for the dependencies to be streamlined and properly worked on with the parties involved;
- It relies on a feature for the development language that is still under development as of time of writing. This imposed a great obstacle in properly testing one of its main functionalities;
- The verification tool is also undergoing internal restructuring, making it additionally troublesome to add a completely new mechanics to it. One being the ability to read intra-code verification artifacts, and the other being the only option to use the verification tool through an external window. For the first mechanic, users still require some knowledge on deductive program verification, imposing an entry barrier that was not expected at first. Secondly, the entire verification experience should have been confined within the development workspace in order to not deter from the experience;
- There are some features in the extension that were thought of, but not implemented. Though they are not fundamental, some are still a quality of life improvement that would positively affect the extension's experience.

In that regard, as future work, this dissertation should be revisited at a time when the issues stated can be properly confronted by all parties involved.

Bibliography

- [1] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, “An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends,” in *2017 IEEE International Congress on Big Data (BigData Congress)*, Jun. 2017, pp. 557–564. [6](#)
- [2] D. Yaga, P. Mell, N. Roby, and K. Scarfone, “Blockchain Technology Overview,” Tech. Rep., Oct. 2018, arXiv:1906.11078 [cs]. [Online]. Available: <http://arxiv.org/abs/1906.11078> [6](#)
- [3] V. Allombert, M. Bourgoïn, and J. Tesson, “Introduction to the Tezos Blockchain,” in *2019 International Conference on High Performance Computing Simulation (HPCS)*, Jul. 2019, pp. 1–10. [6](#)
- [4] X. Yi, D. Wu, L. Jiang, K. Zhang, and W. Zhang, “Diving Into Blockchain’s Weaknesses: An Empirical Study of Blockchain System Vulnerabilities,” p. 23, 2021. [6](#)
- [5] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Cryptography Mailing list at https://metzdowd.com*, 03 2009. [6](#)
- [6] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger.” [8](#)
- [7] N. Szabo, “Smart contracts : Building blocks for digital markets,” 2018. [8](#)
- [8] B. Baby, A. Sunil, and N. Thomas, “A review analysis on smart contract vulnerabilities using blockchain.” [Online]. Available: https://www.ijsr.net/conf/ICIPR2021/ICIPR2021_12.pdf [9](#)
- [9] L. M. Goodman, “Tezos : A self-amending crypto-ledger position paper,” 2014. [10](#)
- [10] M. Neuder, D. Moroz, R. Rao, and D. Parkes, *Selfish Behavior in the Tezos Proof-of-Stake Protocol*. [10](#)
- [11] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (SoK),” in *POST*. [13](#)
- [12] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi, “Smart contract security: A software lifecycle perspective,” vol. 7, pp. 150 184–150 202, conference Name: IEEE Access. [13](#), [16](#)
- [13] P. Praitheeshan, L. Pan, J. Yu, J. K. Liu, and R. Doss, “Security analysis methods on ethereum smart contract vulnerabilities: A survey.” [13](#), [16](#)
- [14] N. F. Samreen and M. H. Alalfi, “A survey of security vulnerabilities in ethereum smart contracts,” p. 10. [13](#), [16](#)
- [15] S. Sayeed, H. Marco-Gisbert, and T. Caira, “Smart contract: Attacks and protections,” vol. 8, pp. 24 416–24 427, conference Name: IEEE Access. [13](#), [16](#)

- [16] S. Kim and S. Ryu, “Analysis of blockchain smart contracts: Techniques and insights,” pp. 65–73. [13](#), [20](#)
- [17] C. Benabbou and □. Gürcan, *A Survey of Verification, Validation and Testing Solutions for Smart Contracts*. [13](#), [20](#)
- [18] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, “ReGuard: Finding reentrancy bugs in smart contracts,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 65–68, ISSN: 2574-1934. [13](#)
- [19] M. Rodler, W. Li, G. O. Karame, and L. Davi, “Sereum: Protecting existing smart contracts against re-entrancy attacks.” [Online]. Available: <http://arxiv.org/abs/1812.05934> [13](#)
- [20] C. F. Torres, J. Schütte, and R. State, “Osiris: Hunting for integer bugs in ethereum smart contracts,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. Association for Computing Machinery, pp. 664–676. [Online]. Available: <https://doi.org/10.1145/3274694.3274737> [15](#)
- [21] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, “Finding the greedy, prodigal, and suicidal contracts at scale.” [15](#)
- [22] I. Nikolic. Maian. Original-date: 2018-03-12T07:58:25Z. [Online]. Available: <https://github.com/ivicanikolicsg/MAIAN> [15](#)
- [23] A. Miller, Z. Cai, and S. Jha, “Smart contracts and opportunities for formal methods,” in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer International Publishing, pp. 280–299. [16](#)
- [24] J. Liu and Z. Liu, “A survey on security verification of blockchain smart contracts,” vol. 7, pp. 77 894–77 904. [16](#), [20](#)
- [25] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, “Verification of smart contracts: A survey,” vol. 67, p. 101227. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S1574119220300821> [16](#), [17](#)
- [26] K. Song, N. Matulevicius, E. B. d. L. Filho, and L. C. Cordeiro, “ESBMC-solidity: An SMT-based model checker for solidity smart contracts.” [Online]. Available: <http://arxiv.org/abs/2111.13117> [21](#)
- [27] S. Lino, S. Ray, and N. Stakhanova, “EtherProv: Provenance-aware detection, analysis, and mitigation of ethereum smart contract security issues,” in *2021 IEEE International Conference on Blockchain (Blockchain)*, pp. 1–10. [21](#)
- [28] I. Grishchenko, M. Maffei, and C. Schneidewind, “EtherTrust: Sound static analysis of ethereum bytecode.” [21](#)

- [29] N. Swamy, C. Hrițcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, “Dependent types and multi-monadic effects in f^* ,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’16. Association for Computing Machinery, pp. 256–270. [Online]. Available: <https://doi.org/10.1145/2837614.2837655> 21
- [30] Z. Yang and H. Lei, “FEther: An extensible definitional interpreter for smart-contract verifications in coq,” vol. 7, pp. 37 770–37 791, conference Name: IEEE Access. 21
- [31] The coq proof assistant. [Online]. Available: <https://coq.inria.fr> 21, 22
- [32] A. Mavridou and A. Laszka, “Designing secure ethereum smart contracts: A finite state machine based approach.” [Online]. Available: <http://arxiv.org/abs/1711.09327> 21
- [33] —, “Tool demonstration: FSolidM for designing secure ethereum smart contracts,” in *Principles of Security and Trust*, ser. Lecture Notes in Computer Science, L. Bauer and R. Küsters, Eds. Springer International Publishing, pp. 270–277. 21
- [34] S. Amani, M. Bégel, M. Bortin, and M. Staples, “Towards verifying ethereum smart contract bytecode in isabelle/HOL,” pp. 66–77. 21
- [35] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, “KEVM: A complete formal semantics of the ethereum virtual machine,” in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 204–217, ISSN: 2374-8303. 21
- [36] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1186–1189, ISSN: 2643-1572. 21
- [37] MythX: Smart contract security service for ethereum. [Online]. Available: <https://mythx.io/about/>,lastaccess26/05/2021 21
- [38] Mythril. Original-date: 2017-09-18T04:14:20Z. [Online]. Available: <https://github.com/ConsenSys/mythril> 21
- [39] P. Ventuzelo. Octopus. Original-date: 2018-06-13T15:06:12Z. [Online]. Available: <https://github.com/pventuzelo/octopus> 21
- [40] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. Association for Computing Machinery, pp. 254–269. [Online]. Available: <https://doi.org/10.1145/2976749.2978309> 21
- [41] P. Tsankov, A. Dan, D. D. Cohen, A. Gervais, F. Buenzli, and M. Vechev, “Securify: Practical security analysis of smart contracts.” [Online]. Available: <http://arxiv.org/abs/1806.01143> 21

- [42] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts,” in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15. [21](#)
- [43] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, “SmartCheck: static analysis of ethereum smart contracts,” in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB ’18. Association for Computing Machinery, pp. 9–16. [Online]. Available: <https://doi.org/10.1145/3194113.3194115> [21](#)
- [44] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He, Y. Tang, X. Lin, and X. Zhang, “SODA: A generic online detection framework for smart contracts.” [21](#)
- [45] P. Antonino and A. W. Roscoe, “Formalising and verifying smart contracts with solidifier: a bounded model checker for solidity.” [21](#)
- [46] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, “Vandal: A scalable security analysis framework for smart contracts.” [Online]. Available: <http://arxiv.org/abs/1809.03981> [21](#)
- [47] A. Mavridou, A. Laszka, E. Stachtari, and A. Dubey, “VeriSolid: Correct-by-design smart contracts for ethereum.” [Online]. Available: <http://arxiv.org/abs/1901.01292> [21](#)
- [48] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, “VerX: Safety verification of smart contracts,” in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, pp. 1661–1677. [Online]. Available: <https://ieeexplore.ieee.org/document/9152791/> [21](#)
- [49] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, “ZEUS: Analyzing safety of smart contracts,” in *NDSS*. [21](#)
- [50] B. Bernardo, R. Cauderlier, B. Pesin, and J. Tesson, “Albert, an intermediate smart-contract language for the tezos blockchain,” in *Financial Cryptography Workshops*. [22](#)
- [51] Y. Nishida, H. Saito, R. Chen, A. Kawata, J. Furuse, K. Suenaga, and A. Igarashi, “Helmholtz: A verifier for tezos smart contracts based on refinement types,” pp. 262–280. [22](#)
- [52] B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson, “Mi-cho-coq, a framework for certifying tezos smart contracts.” [Online]. Available: <http://arxiv.org/abs/1909.08671> [22](#)
- [53] J. S. Reis, P. Crocker, and S. M. de Sousa, “Tezla, an intermediate representation for static analysis of michelson smart contracts.” [Online]. Available: <http://arxiv.org/abs/2005.11839> [22](#)

- [54] L. Horta, J. Reis, S. Sousa, and M. Pereira, “A tool for proving michelson smart contracts in WHY3 *,” pp. 409–414. [22](#)
- [55] L. Horta, J. Reis, M. Pereira, and S. Sousa, *Why3Son: Proving your Michelson Smart Contracts in Why3*. [22](#), [34](#)
- [56] Why3 - where programs meet provers. [Online]. Available: <http://why3.lri.fr> [22](#)
- [57] D. Perez and B. Livshits, “Smart contract vulnerabilities: Vulnerable does not imply exploited,” pp. 1325–1341. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/perez> [19](#)
- [58] X. Cao, J. Zhang, X. Wu, and B. Liu, “A survey on security in consensus and smart contracts,” vol. 15, no. 2, pp. 1008–1028. [Online]. Available: <https://doi.org/10.1007/s12083-021-01268-2> [20](#)
- [59] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, “A survey of smart contract formal specification and verification,” vol. 54, no. 7, pp. 148:1–148:38. [Online]. Available: <https://doi.org/10.1145/3464421> [20](#)