Electronics and Computer Science
Faculty of Engineering and Physical Sciences
University of Southampton

German Nikolishin

December 5, 2023

Folidity - Safe Functional Smart Contract Language

Project Supervisor: Prof. Vladimiro Sassone
Second Examiner: Dr Butthead

A project progress report submitted for the award of
BSc Computer Science

Contents

Contents **i**

1 Introduction **1**

2 Security and Safety of Smart Contracts **2**

2.1 Overview 2

2.2 Vulnerability classification 3

2.3 Setting the scene 3

3 Current Solutions **4**

3.1 Overview 5

3.2 Solidity: the good and the bad 5

3.3 Formal Verification Tools 7

3.4 Safe Smart Contract Languages 8

3.5 Problem statement 9

4 Proposed Solution **10**

5 Project Planning **11**

A Gannt Chart **12**

References **14**

1. Introduction

The idea of "smart contract" [SC] was first coined by Nick Szabo as a computerised transaction protocol [1]. He later defined smart contracts as observable, verifiable, privity-applicable, and enforceable programs. [2]. In other words, it was envisioned for smart contracts to inherit the natural properties of the traditional "paper-based" contracts.

It was only in 2014 when SCs were technically formalised at the protocol level by Dr. Gavin Wood, as an arbitrary program written in some programming language (Solidity) and executed in the blockchain's virtual machine of Ethereum (EVM) [3].

Ethereum Virtual Machine (EVM) iterated over the idea of Bitcoin Scripting allowing developers to deploy general-purpose, Turing-Complete programs that can have own storage, hence state. This enabled the development of more sophisticated applications that grew beyond the simple transfers of funds among users.

Overall, SC can be summarised as an *immutable*, *permissionless*, *deterministic* computer programs that are executed as part of state transition in the blockchain system. At the time of writing, Solidity is still the most widely used SC language (SCL) [4].

After a relatively short time, SCs have come a long way and allowed users to access different online services in a completely trustless and decentralised way. The applications have spanned across financial, health, construction[5], and other sectors.

2. Security and Safety of Smart Contracts

2.1 Overview

With the increased adoption of decentralised applications (DApps) and the increased total value locked in DApps, there has been evidence of numerous attacks and exploits focused on extracting funds from SCs. Due to the permissionless nature of SCs, the most common attack vector is exploiting the mistakes in the SC's source code. Specifically, the attacker can not tamper with the protocol code due to consensus mechanisms. Instead, they can cleverly tamper with the publicly accessible parameters and interfaces to force the SC into an unexpected state, essentially gaining partial control of it.

A notorious example of such attacks is the DAO hack when hackers exploited unprotected re-entrance calls to withdraw **$50 million worth of ETH**. This event forced the community to hard-fork the protocol to revert the transaction provoking a debate on the soundness of the action [6].

Another less-known example is the "King of the Ether" attack which was caused by the unchecked low-level Solidity `send` call to transfer funds to a contract-based wallet [7]. The "King of the Ether Throne" contract was not able to recognise the failed transaction on the wallet side. Instead, the contract proceeded with the operation, incorrectly mutating its internal state.

Other issues involve the *safety* and *liveness* of SCs. The term *safety* is used to describe the *functional safety* and *type safety*. It is often used interchangeably with the *security* of code as compromising the former has effects on the latter. When talking about *liveness*, we describe the business logic of a DApp, in particular, whether it transitions into the expected new state.

Safety and *liveness* can be compromised due to the programmer's mistakes in the source code that can result in the SC entering the terminal unexpected state preventing users from interacting with it[8].

2.2 Vulnerability classification

There has been an effort in both academia and industry to classify common vulnerabilities and exploits in SCs in blockchain systems [9][10][11]. Some of the work has been recycled by bug bounty platforms growing the community of auditors and encouraging peer-review of SCs such as *Code4rena*[1], *Solodit*[2], and many others.

Analysing the work mentioned above, SCs vulnerabilities can be categorised into the 6 general groups that are outlined in Table 2.1.

Note that we do not evaluate the listed vulnerabilities based on their severity. As far as this paper is concerned, all vulnerabilities are considered to be of equal weight for the reasons described in Section 2.3.

2.3 Setting the scene

Numerous deployed DApps allowed the community of developers and auditors to learn from the mistakes and the past, and generally improve the code quality and security of SCs. Audits are now an essential part of the release cycle of any DApp.

However, even with the raised awareness for the security and safety of SCs, recent reports from ”code4rena” still show *SCV:3*, *SCV:4* and *SCV:5* commonly present in the recent audit reports[12][8][13].

In particular, in [13] a relatively simple calculation mistake resulted in other SC users being unable to withdraw their funds.

It can be seen that SC Vulnerabilities illustrated in Table 2.1 are still evident in modern SCs resulting in opening them up to vulnerabilities of different severity levels. Looking at the mentioned reports, there is little consensus about the weight of each vulnerability. Therefore, as mentioned earlier, we can not classify any particular vulnerability to be more severe than the other as it solely depends on the context in the code it is present in. Furthermore, given the pattern in the mistakes made by SC developers, it has been realised that additional tooling or alternative SCLs need to be discovered to minimise the exposure of SC code to the earlier-mentioned vulnerabilities.

[1] https://code4rena.com

[2] https://solodit.xyz

Code	**Title**	**Summary**
SCV1	Timestamp manipulation	Timestamp used in control-flow, randomness and storage, can open an exploit due to an ability for validator to manipulate the timestamp
SCV2	Pseudo-randomness	Using block number, block hash, block timestamp are not truly random generated parameters, and can be manipulated by the adversary validator
SCV3	Invalidly-coded states	When coding business logic, control-flow checks can be incorrectly coded resulting the SC entering into invalid state
SCV4	Access Control exploits	This is a more broad categorisation of vulnerabilities. It occurs when an adversary calls a restricted function. This is specifically present in *upgradeability* and *deleteability* of SCs
SCV5	Arithmetic operations	SCs are suspected to the same arithmetic bugs as classic programs. Therefore, unchecked operations can result in underflow/overflow or deletion by zero
SCV6	Unchecked externall calls	Unchecked re-entrant, forward, delegate calls can result in the contract entering into unexpected state

Table 2.1: classification of SC Vulnerabilities

3. Current Solutions

3.1 Overview

As mentioned earlier, given the increased use of SCs and the consistency in the presence of vulnerabilities and programmer mistakes different solutions have been presented to attempt to mitigate those. We can generally categorise them into 2 groups: safe SCLs which allow users to write safe and secure code, particularly described in Chapter 3.4, and formal verification tools which are used alongside traditional SCLs and are described in Chapter 3.3.

At the end of the chapter, we will have reviewed both categories of tools allowing us to evaluate their effectiveness in correlation to usability. Particularly, this chapter aims to provide a clear and concise framework to analyze and work with the SC tools dedicated to producing error-proof DApps.

3.2 Solidity: the good and the bad

Before we begin the analysis of existing solutions, it is essential to introduce the first and most popular SCL, Solidity[4][14]. Solidity has paved the way for SC development and become a de facto standard in the blockchain space. We will later discover a lot of similarities with Solidity in other SCLs. Understanding the way Solidity works will enable us to understand its limitations, and how other SCLs and toolings try to address them.

Solidity is a domain-specific, object-oriented[1], statically-typed, compiled programming language[14].

Looking at the Listing 1, we can identify some similarities with the OOP languages and C-like syntax. In particular, Solidity shares visibility keywords on lines *6* and *20*, variable declarations, methods. Noting the differences, Solidity has blockchain-specific data types such as *address* which intuitively represents an address on the system. Importantly, functions have to be declared *public* in order to be callable externally by the user or other contracts. We can also see the event declaration on

[1] It is also often referred to as *contract-oriented*. We will use these terms interchangeably.

```
1  // SPDX-License-Identifier: GPL-3.0
2  pragma solidity >=0.7.0 <0.9.0;
3
4  contract Owner {
5
6      address private owner;
7      event OwnerSet(address indexed oldOwner, address indexed newOwner);
8
9      modifier isOwner() {
10         require(msg.sender == owner, "Caller is not owner");
11         _;
12     }
13
14     constructor() {
15         console.log("Owner contract deployed by:", msg.sender);
16         owner = msg.sender;
17         emit OwnerSet(address(0), owner);
18     }
19
20     function changeOwner(address newOwner) public isOwner {
21         emit OwnerSet(owner, newOwner);
22         owner = newOwner;
23     }
24
25     function getOwner() external view returns (address) {
26         return owner;
27     }
28 }
```

Listing 1: Simple Solidity contract

the line *7*, which allows EVM to log an action on the system to be accessed by the client. Solidity also offers specific functions, called modifiers that allow for a change of behaviour of the function it is used on[14]. Finally, *view* keyword allows that the function does not modify the state and only reads from it. Other specification details are omitted as they touch upon more EVM-specific functionality.

It is evident from the listing the similarity of Solidity with other C-inspired systems languages that reduces the learning curve and makes it more appealing for developers to pick up. The syntax simplicity also enables clarity in reading the code and auditing it. The imperative, and object-oriented style provides additional familiarity for developers allowing them to think of an individual contract as a class object.

On the other side, programming in Solidity comes with similar challenges as its parents. Developers need to be aware of its operations on memory and storage when designing complex storage applications, how the execution stack is handled at the VM level, and implications on the use of unchecked arithmetic operations *SCV:5*. Additionally, the vulnerabilities listed in Table 2.1 has been mainly composed of Solidity SC exploits. Moreover, the immutable nature of SCs restricts the iterative approach in development, posing an even greater responsibility on developers to carefully model and design programs. This is where formal verification tools have found strong applications.

3.3 Formal Verification Tools

Formal verification describes the assessment of the correctness of a system with respect to a formal specification[15]. The specification is usually described in terms of models that are verified using mathematical proofs. There are multiple ways to formally verify a program that are focused on the specific parts of it. *Model checking* utilises propositional logic to verify the mathematical abstractions of the system. *Theorem proving* involves the verification of relations between the model and the statements about the system. Finally, *symbolic execution* focuses on the execution of the program using symbolic values instead of concrete values[15]. Different tools utilise different combinations of methods mentioned above. We are going to look at a couple of them.

KEVM[2] is a tool that provides an executable semantics of EVM using $\mathbb{K}$ framework. It uses reachability logic to reason symbolically about the system[16]. KEVM is a powerful tool that operates at the EVM bytecode level. Specifically, SC developers are required to write a specification in a separate file that is checked against the compiled EVM bytecode of the SC. Whilst this provides more fine-grained assurance of the safety and correctness, it requires specialised knowledge of the $\mathbb{K}$ framework and EVM semantics, hence, significantly increasing the development time.

The other interesting tool is Dafny[3]. Dafny is a general-purpose tool that checks inputs in any language using Hoare-logic and high-level annotations. Although Dafny offers compilation to some system languages, Solidity is not yet a supported target. Notably, work in the field suggests that the Dafny can be an effective and easy-to-use tool to produce a formal specification[17]. The syntax resembles a traditional imperative style and is substantially easier to learn and understand than KEVM.

Going further, there are tools that can used alongside Solidity code such as Scribble[4]. Scribble enables developers to provide formal specification of functions inside docstrings seamlessly integrating with existing Solidity code. It offers VS Code extensions and is actively maintained by Consensys. The trade-off is the limited expressiveness in comparison with KEVM and Dafny.

[2] https://jellopaper.org/index.html

[3] https://dafny.org/latest/DafnyRef/DafnyRef

[4] https://docs.scribble.codes/

Finally, there have been experiments to formally verify SC without any additional annotations. In particular, VeriSmart specifically focuses on ensuring arithmetic safety and preciseness in SCs[18]. However, VeriSmart fails to detect other types of errors, although the effort to apply to verifier to more areas of SC has been made.

Formal verification is a multi-disciplinary field offering multiple ways of reason about the systems. One of the actively researched fields is bounded model verification[19]. Developers are required to reason about the programs as finite state machines (FSM). This reasoning approach is more apparent in SC development since the state transition is at the core of blockchain execution. Bounded model checking has been realised by only a few experimental projects such as Solidifier[20], and Microsoft[21]. Both words attempt to translate Solidity code to an intermediate modeling language, Boogie[22]. Boogie then leverages SMT solvers in order to find any assertion violations.

All in all, we can see that formal verification tools provide a robust way of ensuring the safety and security of SCs. While significant effort has been made in the field, it is evident that formal verification tools in SC development try to compensate for Solidity's implicit state transitions and lack of *implicit* safety. Furthermore, the current tools significantly complicate the development process and provide very little comfort for developers to opt-in for them.

3.4 Safe Smart Contract Languages

As mentioned above, Solidity imperative programming style compromises a safe development process. Therefore, multiple attempts have been made to address a flawed programming model[4]. Alternative SCLs aim at providing built-in safety features in a type system, modeling, and function declaration in order to minimise the need for external tooling.

Some languages, such as Vyper[5], strive for simplicity. By stripping off some low-level features, Vyper minimises the chances of the developer misusing the dangerous operations. It also provides overflow checking, signed integers, and other safe arithmetic operations. However, Vyper is still immature and the recent bug in the compiler caused a massive re-entrancy exploit in the *curve.fi* AMM protocol[23]. Furthermore, Vyper still suffers from the same implicit state transition problem as Solidity.

To address the problem, it has been realised that a functional programming style may suit better for SC development due to an explicit approach to reason about a state transition. Although some small effort has been made in adapting Haskell, neither of the projects received any long-term support. It is still worth looking at some of the languages that suggest novice approaches to development.

[5] https://docs.vyperlang.org/en/latest/index.html

Flint is an experiment language with protected calls and asset types[24]. Protected calls introduce a role-based access system where the SC developer can specify the permitted caller to a message function. Another unique feature is array-bounded loops that partially address the halting problem. Flint also addresses a state-transition problem by allowing developers to specify all possible states in the contract. The message functions then need to explicitly specify then the state transition occurs. The language provides a significant improvement in a modeling approach. However, it still lacks the modeling SC input data in terms of constraints and invariants, and explicit state transition is still an optional feature that the developer can miss in using.

Another promising SCL reasons about SC development through dependent and polymorphic types[25]. It extends Idris[6]. It makes the developer model the SC as part of a state transition function by adopting a functional programming style. Dependent types provide a more fine-grained controller over the input and output data that flow through the SC functions. In particular, similar to Haskell, the language offers *side-effects* functionality that resembles *IO* monads in Haskell. The downside of the approach is that the syntax has become too cumbersome for other developers to learn. Thus, it has been stated that the language does not strive for simplicity and sacrifices it for safety.

3.5 Problem statement

Analysing the solutions mentioned earlier, we can identify the positive trend in providing the safety and security of SCs. Modern formal verification methods offer support to SC developers in ensuring that their code satisfies the requirements of the system, while proposed SCL solutions offer runtime safety minimising the need for the former.

However, there has been no effort in attempting to combine the two approaches into a single development process. Formal verification tools focus on the validation of functional correctness and model consistency of a program at the compile time, whereas SCLs focus on data validation at the runtime. Recent work suggests that the improved optimisation of SMT solvers allows us to turn the formal model specification into the runtime assertions[26]. Furthermore, no effort has been made to minimise false negatives in SC formal modeling, even though the methods have been developed for traditional systems[27].

[6] https://www.idris-lang.org

4. Proposed Solution

5. Project Planning

A. Gannt Chart

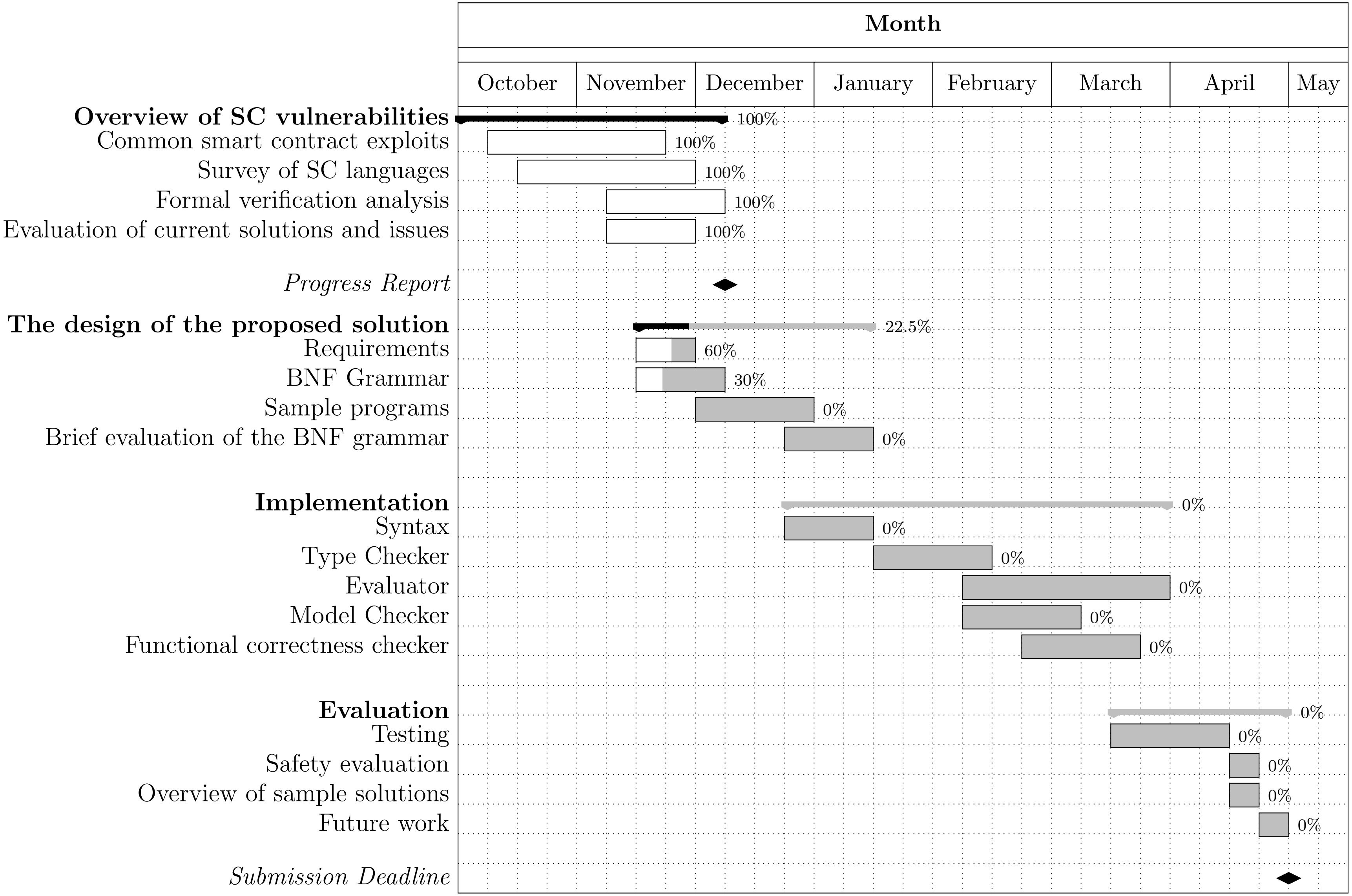

Figure A.1: Gantt Chart

References

[1] Nick Szabo. Smart contracts. In *Nick Szabo's Papers and Concise Tutorials*, 1994.

[2] Nick Szabo. Smart contracts: Building blocks for digital markets. In *Nick Szabo's Papers and Concise Tutorials*, 1996.

[3] Dr. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. https://gavwood.com/paper.pdf, 2014.

[4] Mohammad Hamdaqa Majd Soud, Gísli Hjálmtýsson. Dissecting smart contract languages: A survey. arXiv:2310.02799v2, 2023.

[5] Theodoros Dounas and Davide Lombardi. A cad-blockchain integration strategy for distributed validated digital design connecting the blockchain. 09 2018.

[6] Xiangfu Zhao, Zhongyu Chen, Xin Chen, Yanxia Wang, and Changbing Tang. The dao attack paradoxes in propositional logic. In *2017 4th International Conference on Systems and Informatics (ICSAI)*, pages 1743–1746, 11 2017.

[7] King of the Ether. Post-mortem investigation. https://www.kingoftheether.com/postmortem.htm, 2016. Accessed: 28/11/2023.

[8] Code4rena C4. Ondo finance. findings and analysis report. https://code4rena.com/reports/2023-09-ondo, 2023. Accesssed: 29/11/2023.

[9] Jinson Varghese Behanan. Owasp smart contract top 10. https://owasp.org/www-project-smart-contract-top-10/. Accessed: 28/11/2023.

[10] Stefano De Angelis, Federico Lombardi, Gilberto Zanfino, Leonardo Aniello, and Vladimiro Sassone. Security and dependability analysis of blockchain systems in partially synchronous networks with byzantine faults, 2023.

[11] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, page 164–186, Berlin, Heidelberg, 2017. Springer-Verlag.

[12] Code4rena C4. Arcade.xyz. findings and analysis report. https://code4rena.com/reports/2023-07-arcade, 2023. Accesssed: 29/11/2023.

[13] Code4rena C4. Centrifuge. findings and analysis report. https://code4rena.com/reports/2023-09-centrifuge, 2023. Accesssed: 29/11/2023.

[14] Ethereum Foundation. Solidity. https://docs.soliditylang.org/en/v0.8.23, 2023. Accesssed: 01/12/2023.

[15] Ethereum Foundation. Formal verification of smart contracts. https://ethereum.org/en/developers/docs/smart-contracts/formal-verification/, 2023. Accesssed: 05/12/2023.

[16] Hildenbrandt, Everett, Saxena, Manasvi, Rodriguesa, Nishant, Zhu, Xiaoran, Daian, Philip, Guth, Dwight, Moore, Brandon, Park, Daejun, Zhang, Yi, Stefanescu, Andrei, Rosu, and Grigore. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018.

[17] Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive verification of smart contracts with dafny, 2022.

[18] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts, 2019.

[19] Bernhard Beckert, Thorsten Bormer, Florian Merz, and Carsten Sinz. Integration of bounded model checking and deductive verification. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software*, pages 86–104, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[20] Pedro Antonino and A. W. Roscoe. Solidifier: Bounded model checking solidity using lazy contract deployment and precise memory modelling. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1788–1797, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Yuepeng Wang, Shuvendu K. Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, and Immad Naseer. Formal specification and verification of smart contracts for azure blockchain, 2019.

[22] K. Rustan M. Leino. This is boogie 2. June 2008.

[23] Chainsecurity. Curve lp oracle manipulation: Post mortem. https://chainsecurity.com/curve-lp-oracle-manipulation-post-mortem/, 2023. Accessses: 05/12/2023.

[24] Franklin Schrans, Daniel Hails, Alexander Harkness, Sophia Drossopoulou, and Susan Eisenbach. Flint for safer smart contracts, 2019.

[25] Jack Pettersson and Robert Edström. Safer smart contracts through type-driven development, 2016.

[26] Fonenantsoa Maurica, David R. Cok, and Julien Signoles. Runtime assertion checking and static verification: Collaborative partners. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 75–91, Cham, 2018. Springer International Publishing.

[27] Thai Son Hoang. *An Introduction to the Event-B Modelling Method*, pages 211–236. 07 2013.