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

German Nikolishin

December 22, 2023

Folidity - Safe Functional Smart Contract Language

Project Supervisor: Prof. Vladimiro Sassone
Second Examiner: TBD

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

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES
ELECTRONICS AND COMPUTER SCIENCE

A project report submitted for the award of BSc Computer Science

by German Nikolishin

This paper addresses the long-lasting problem involving the exploits of Smart
Contract vulnerabilities. There are tools, such as in the formal verification field
and alternative Smart Contract languages, that attempt to address these issues.
However, neither approach has managed to combine the static formal verification
and the generation of runtime assertions. Furthermore, this work believes that
implicit hidden state transition is the root cause of security compromises. In light
of the above, we introduce Folidity, a safe functional Smart Contract language with
a unique approach to reasoning about the modelling and development of Smart
Contract systems. Folidity features explicit state transition checks, a model-first
approach, and built-in formal verification tooling.

Statement of Originality

- I have read and understood the ECS Academic Integrity information and the University's Academic Integrity Guidance for Students.

- I am aware that failure to act in accordance with the Regulations Governing Academic Integrity may lead to the imposition of penalties which, for the most serious cases, may include termination of programme.

- I consent to the University copying and distributing any or all of my work in any form and using third parties (who may be based outside the EU/EEA) to verify whether my work contains plagiarised material, and for quality assurance purposes.

You must change the statements in the boxes if you do not agree with them.

We expect you to acknowledge all sources of information (e.g. ideas, algorithms, data) using citations. You must also put quotation marks around any sections of text that you have copied without paraphrasing. If any figures or tables have been taken or modified from another source, you must explain this in the caption and cite the original source.

> **I have acknowledged all sources, and identified any content taken from elsewhere.**

If you have used any code (e.g. open-source code), reference designs, or similar resources that have been produced by anyone else, you must list them in the box below. In the report, you must explain what was used and how it relates to the work you have done.

> **I have not used any resources produced by anyone else.**

You can consult with module teaching staff/demonstrators, but you should not show anyone else your work (this includes uploading your work to publicly-accessible repositories e.g. Github, unless expressly permitted by the module leader), or help them to do theirs. For individual assignments, we expect you to work on your own. For group assignments, we expect that you work only with your allocated group. You must get permission in writing from the module teaching staff before you seek outside assistance, e.g. a proofreading service, and declare it here.

> **I did all the work myself, or with my allocated group, and have not helped anyone else.**

We expect that you have not fabricated, modified or distorted any data, evidence, references, experimental results, or other material used or presented in the report. You must clearly describe your experiments and how the results were obtained, and include

all data, source code and/or designs (either in the report, or submitted as a separate file) so that your results could be reproduced.

The material in the report is genuine, and I have included all my data/code/designs.

We expect that you have not previously submitted any part of this work for another assessment. You must get permission in writing from the module teaching staff before re-using any of your previously submitted work for this assessment.

I have not submitted any part of this work for another assessment.

If your work involved research/studies (including surveys) on human participants, their cells or data, or on animals, you must have been granted ethical approval before the work was carried out, and any experiments must have followed these requirements. You must give details of this in the report, and list the ethical approval reference number(s) in the box below.

My work did not involve human participants, their cells or data, or animals.

Contents

1. Introduction

The concept 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, smart contracts were envisioned to inherit the natural properties of traditional "paper-based" contracts.

In 2014 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[3].

Ethereum Virtual Machine (EVM) iterated over the idea of Bitcoin Scripting[4], allowing developers to deploy general-purpose, Turing-Complete programs that can have their own storage, hence the state, written in Solidity[5]. This enabled sophisticated applications that grew beyond the simple funds transfers among users.

Overall, SC can be summarised as an *immutable*, *permissionless*, *deterministic* computer program that is executed as part of state transition in the blockchain system[6][3].

After a relatively short time, SCs have come a long way and allowed users to access different online services, also known as Decentralised Applications (DApps), in a completely trustless and decentralised way. The applications have spanned financial, health, construction[7], and other sectors.

2. Security and Safety of Smart Contracts

2.1 Overview

With the increased adoption of DApps and total value locked in them, there have been numerous attacks focused on extracting funds from SCs. Due to the permissionless nature of SCs, the most common attack vector exploits 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 interact with the publicly accessible parameters and interfaces to force the SC into an unexpected state, essentially gaining partial control of it.

A notorious example 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 [8].

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 [9]. The "King of the Ether Throne" contract could not 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*. *Functional safety* refers to the guarantees that the system behaves according to the specification irrespective of the input data[10], whereas *type safety* refers to the guarantees that the language provides a sound type system[11]. The two are often used interchangeably with the *security* of code as compromising the former affects the latter. When talking about *liveness*, we describe the business logic of a DApp, particularly whether it transitions into the expected new state[12]. This is particularly important for the execution of mission-critical software in a distributed context.

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 in an unexpected state preventing users from interacting with it[13].

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[14][15][16]. Some of the work has been recycled by bug bounty platforms, growing the community of auditors and encouraging peer-review of SCs through the websites such as *Code4rena*[1], *Solodit*[2], and many others.

Analysing the work mentioned above, SC vulnerabilities can be categorised into the six general groups outlined in Table 2.1. The six categories have been defined based on the analysis of the most common vulnerabilities, and how they affect the SC execution. Each category represents the general scope for a specific set of vulnerabilities that should be addressed in the SC development.

2.3 Setting the scene

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[17][13][18].

In particular, in [18], 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 exploits of different severity levels. Looking at the mentioned reports, there is little consensus about the weight of each vulnerability. Therefore, we can not classify any particular vulnerability as more severe than the other as it solely depends on the context in the code it is present. Furthermore, 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

Different solutions have been presented to mitigate the consistency in the presence of vulnerabilities and programmer mistakes. We can generally categorise them into two groups: safe SCLs, which allow users to write safe and secure code, particularly described in Chapter 3.3, and formal verification tools used alongside traditional SCLs presented in Chapter 3.2.

This chapter reviews both categories of tools, allowing us to evaluate their effectiveness in correlation to usability, aiming to provide a concise framework to analyse and work with the SC tools dedicated to producing error-proof DApps.

3.2 Formal Verification Tools

Formal verification describes the assessment of the correctness of a system concerning a formal specification[19]. The specification is usually described in terms of verifiable models using mathematical proofs. There are multiple ways to verify a program formally focused on specific parts. *Model checking* utilises propositional logic to verify the mathematical abstractions of the system[20]. *Theorem proving* involves verifying relations between the model and the statements about the system[21]. Finally, *symbolic execution* focuses on the execution of the program using symbolic values instead of concrete values[19]. Different tools utilise different combinations of methods mentioned above.

KEVM[1] is a tool that provides an executable semantics of EVM using $\mathbb{K}$ framework. It uses reachability logic to reason symbolically about the system[22]. 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.

---

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

The other interesting tool is Dafny[2]. 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[23]. The syntax resembles a traditional imperative style and is substantially easier to learn and understand than KEVM.

Some tools can be used alongside Solidity code, such as Scribble[3]. Scribble enables developers to provide formal specifications of functions inside docstrings seamlessly integrating with existing Solidity code. It offers VS Code extensions and is actively maintained by Consensys[4]. The trade-off is the limited expressiveness in comparison with KEVM and Dafny.

Finally, experiments have been conducted to verify SC without any formal annotations. In particular, VeriSmart focuses explicitly on ensuring arithmetic safety and preciseness in SCs[24]. However, VeriSmart fails to detect other types of errors, although an effort has been made to apply the verifier to more areas of SC[25].

Formal verification is a multi-disciplinary field offering multiple ways of reason about the systems. One of the actively researched topics is bounded model verification[26]. Developers are required to reason about the programs as finite state machines (FSM)[27]. 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[28] and Microsoft[25]. Both projects attempt to translate Solidity code to an intermediate modelling language, Boogie[29]. Boogie then leverages SMT solvers to find any assertion violations.

Overall, we can see that formal verification tools provide a robust way of ensuring the safety of SCs. While significant effort has been made in the field, it is evident that formal verification tools in SC development attempt to compensate for Solidity's implicit state transitions and lack of *implicit* safety.

3.3 Safe Smart Contract Languages

Multiple attempts have been made to address a flawed programming model of Solidity[30]. Alternative SCLs aim to provide built-in safety features in a type system, modelling, and function declaration 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 developer's chances of misusing the dangerous

---

[2]https://dafny.org/latest/DafnyRef/DafnyRef
[3]https://docs.scribble.codes/
[4]https://consensys.io/diligence/scribble
[5]https://docs.vyperlang.org/en/latest/index.html

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[31]. Furthermore, Vyper still suffers from the same implicit state transition problem as Solidity.

Flint is an experiment language with protected calls and asset types[32]. 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 need to specify the state transition, which occurs explicitly. The language provides a significant improvement in a modelling approach. However, it still lacks the modelling 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[33]. It extends Idris[6] and 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 control 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.

3.4   Problem statement

We can identify the positive trend in providing the safety 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 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[34]. Furthermore, no effort has been made to minimise false negatives in SC formal modelling, even though the methods have been developed for traditional systems, such as Event-B[35].

---

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

4. Proposed Solution

4.1 Outline

In light of the above, we believe there is a need for a solution that combines two approaches to allow SC developers to reason about their program in terms of FSM models that can be verified at the compile time for functional correctness and model consistency, and enable an automatic generation of invariants and constraints to validate the data at runtime.

We propose *Folidity*, a safe smart contract language. Folidity will offer the model-first approach to the development process while featuring the functional-first programming style. The language intends to offer a safe and secure-by-design approach to the programming, ensuring the developer is aware of any state transitions during execution.

The list of feature requirements has been comprised based on the vulnerabilities described in Table 2.1.

1. **Provide abstraction over timestamp** in response to *SCV:1*. We are interested in the limited use of timestamps in SCs in favour of block number or another safe primitive.

2. **Provide a safe interface for randomness** in response to *SCV:2*. Folidity should also provide source of randomness through a standardised interface.

3. **Enable model-first approach in development** in response to *SCV:3*. Developers should reason about the storage in terms of models and how they are updated by events. This approach is inspired by the Event-B[35] work, which can also be applied to SC development.

4. **Explicit state checks at runtime** in response to *SCV:3* and *SCV:6*. Similar to *Requirements:3*, SC developers should be aware of any state transitions that update the state of the model. State transitions must happen explicitly and be validated at the runtime to guarantee *liveness*.

5. **Static typing** in response to *SCV:3* and *SCV:5*.

6. **Polymorphic-dependent types** in response to *SCV:3*. Polymorphic-dependent types should be part of a runtime assertion check during state transition and model mutation[1].

7. **Role-based access** in response to *SCV:4*. All message functions that mutate the model should be annotated with the role-access header specifying which set of accounts is allowed to call it.

8. **Checked arithmetic operations** in response to *SCV:5*. All arithmetic operations should be checked by default, and the developer is responsible for explicitly specifying the behaviour during over/underflow, similar to Rust.

9. **Enforced checked recursion or bounded loops** in response to *SCV:3*. Infinite loops should not be permitted, and any loops should generally be discouraged in favour of recursion. The recursion base case should be specified explicitly with appropriate invariants. Bounded loops may be used but should be limited to list or mapping iterations.

As part of the language design, the SC building workflow is illustrated in Figure 4.1
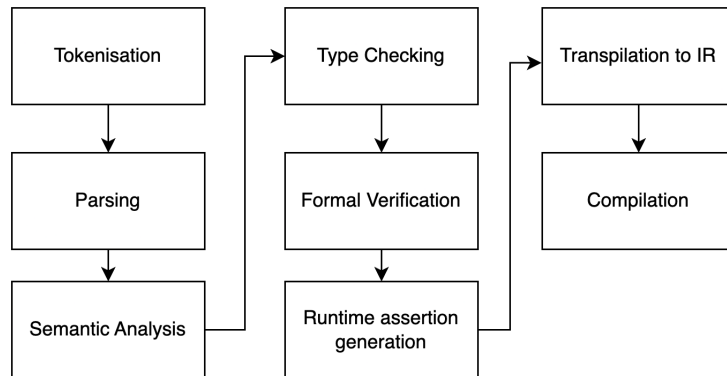

FIGURE 4.1: Build workflow

As one of the core features of Folidity, formal verification is part of the build process. Having verified the model consistency, invariants, and constraints, the program is considered safe to generate runtime assertions.

Another core feature is a pure computation context of the SC in Folidity. As illustrated in Figure 4.2(a), state mutations to the contract storage and the global state (e.g. account balances) happen independently of each other. Folidity proposes a new execution model when a portion of a global state is *embedded* into the local state of the SC as shown in Figure 4.2(b). *Global state* refers to the overall state of the blockchain system (e.g. account balances), whereas *local state* describes the storage of an individual SC.

---

[1] *Model mutation* and *state transition* refer to the same process. They are used interchangeably
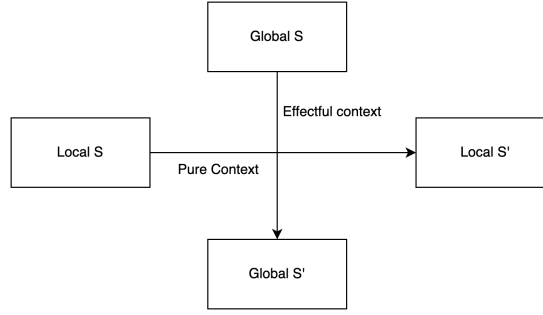
(a) Traditional execution context
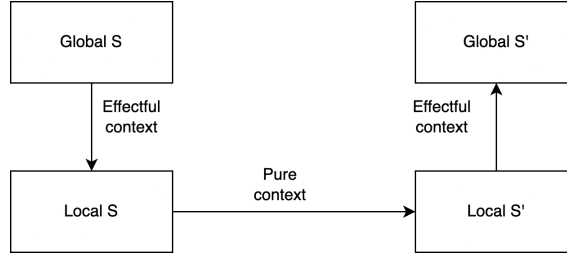
(b) Transfomed execution context

FIGURE 4.2: Changes to the execution context

4.1.1 Model consistency: Simple example

As an example of the theory behind model consistency in SCs, let's look at role-based access. Suppose:

- $* = \{\text{All addresses}\}$

- $M = \{\text{Moderators of the system}\}$

- $A = \{\text{Admins of the system}\}$

Then, we can model a role-based access hierarchy as

$$A \subseteq M \subseteq *$$

Subsequently, given some event for the system *add_mod(a: Address)*, we can define following invariants for the system:

$$i_0 := card(A) = 1$$
$$i_1 := card(M) = 5$$

And the invariant for the event:

$$i_2 := c \in A$$

Where

- c - caller's address

- i_n - enumerated invariant with some boolean statement

- $card(...)$ - cardinality of a set

For the denoted event, suppose we mutate the model by adding an address to a set of admins:

$$A := A \cup \{a\}$$

Then, we can verify the model consistency for some state transition from an initial state S to a new state S', $S \to S'$, using propositional logic.

$$\frac{(i_0 \wedge i_1 \wedge i_2) \to A \cup \{a\}, a \in *, c \in A}{A \cup \{a\}}$$

However, as it can be seen, one of the premises violates the invariant, in particular:

$$\frac{card(A) = 1 \to A \cup \{a\}, a \in *}{A \cup \{a\}}$$

.

In practice, the following error can be picked at the compile time by using symbolic execution of the code. The other invariant, i_2, can be picked at the runtime by generating an appropriate assertion.

4.2   Implementation

The language will be implemented using Rust[2] due to its memory-safety guarantees and efficiency. Different parser-combinators alongside custom lexers are going to be used for the development of the parser. Folidity also requires SMT-solver for formal verification and generation of runtime assertions. In order to facilitate this functionality, Z3[3] will be used since it also provides Rust bindings. It was debated to use Boogie, since it provides a higher-level abstraction, but it was quickly discarded due to lack of documentation and increased development time.

As a target blockchain for the language, Algorand[4] has been selected. Algorand is a decentralized blockchain platform designed for high-performance and low-cost transactions, utilising a unique consensus algorithm called Pure Proof-of-Stake to

---

[2]https://www.rust-lang.org
[3]https://microsoft.github.io/z3guide
[4]https://developer.algorand.org

achieve scalability, security, and decentralisation[36]. One of the potential drawbacks of Folidity is a computational overhead due to complex abstractions and additional assertions. EVM-based blockchains have varying costs for the execution, i.e. fees, that depend on the complexity of a SC. On the contrary, although Algorand has a limited execution stack, it offers fixed, low transaction fees. Additionally, Algorand execution context explicitly operates in terms of state transition, which perfectly suits the paradigm of Folidity. Finally, Algorand offers opt-in functionality and local wallet storage, allowing users to explicitly opt-in to use the SC. This provides additional support in the role-based access control in Folidity.

As a target compilation language, Tealish[5] has been chosen. Although, Algorand offers Teal[6] – a low-level, stack-based programming language. Due to time limitations of the project, it is more realistic to use Tealish. It operates on the same level as Teal while offering useful abstractions speeding up the development of Folidity.

4.3   Scope

As part of the development process, it has been decided to limit the scope to supporting only a single SC execution. Cross-contract calls require extra consideration in design and development. Therefore, *SCV:6* may not be fully addressed in the final report. Additionally, optimisation of the execution is not considered relevant at this stage in favour of safety and simplicity. Finally, Algorand offers smart signatures, a program that is delegated a signing authority[7]. As they operate in a different way from SCs, they are also outside the scope of this project.

5. Project Planning

A significant groundwork in research of current solutions and their limitations has been done as illustrated by Gannt chart in Appendix A.1. Since the requirements have been collected, some progress has been made in the design of BNF grammar that will later pave the way for the development of the parser. It is still possible to research more formal verification methods during the grammar design.

From the beginning of January, the first iteration of grammar should be completed, and the active development of the type checker and formal verifier should begin.

---

[5]https://tealish.tinyman.org
[6]https://developer.algorand.org/docs/get-details/dapps/avm/teal/
[7]https://developer.algorand.org/docs/get-details/dapps/smart-contracts/smartsigs

A. Gannt Chart

	Month						
October	November	December	January	February	March	April	May

Overview of SC vulnerabilities ━━━━━ 100%
Common smart contract exploits 100%
Survey of SC languages 100%
Formal verification analysis 100%
Evaluation of current solutions and issues 100%

Progress Report ◆

The design of the proposed solution ━━━ 22.5%
Requirements 60%
BNF Grammar 30%
Sample programs 0%
Brief evaluation of the BNF grammar 0%

Implementation 0%
Syntax 0%
Type Checker 0%
Evaluator 0%
Model Checker 0%
Functional correctness checker 0%

Evaluation 0%
Testing 0%
Safety evaluation 0%
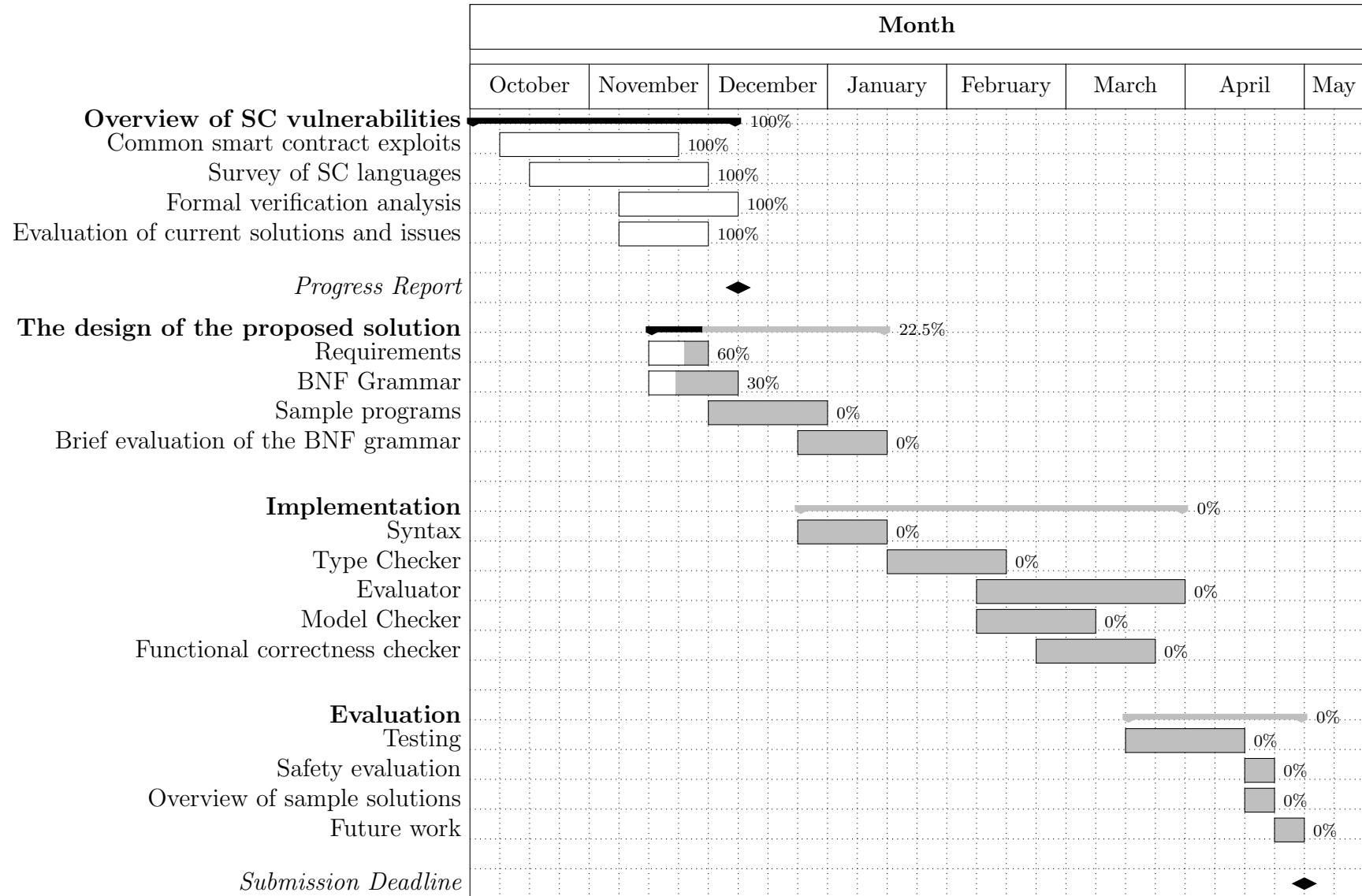Overview of sample solutions 0%
Future work 0%

Submission Deadline ◆

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] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

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

[6] Stuart D. Levi, Alex B. Lipton, Skadden, Arps, Slate, and Meagher & Flom LLP. An introduction to smart contracts and their potential and inherent limitations. https://corpgov.law.harvard.edu/2018/05/26/an-introduction-to-smart-contracts-and-their-potential-and-inherent-limitations, 2018.

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

[8] 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.

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

[10] Heinz Gall. Functional safety iec 61508 / iec 61511 the impact to certification and the user. pages 1027 – 1031, 05 2008.

[11] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[12] Rob van Glabbeek. Ensuring liveness properties of distributed systems: Open problems. *Journal of Logical and Algebraic Methods in Programming*, 109:100480, 2019.

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

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

[15] 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.

[16] 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.

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

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

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

[20] Alireza Souri, Amir Masoud Rahmani, Nima Jafari Navimipour, and Reza Rezaei. A symbolic model checking approach in formal verification of distributed systems. *Human-centric Computing and Information Sciences*, 9(1):4, 2019.

[21] John Harrison, Josef Urban, and Freek Wiedijk. History of interactive theorem proving. In *Computational Logic*, 2014.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] M.D. Nguyen, D. Stoffel, M. Wedler, and W. Kunz. Transition-by-transition fsm traversal for reachability analysis in bounded model checking. In *ICCAD-2005. IEEE/ACM International Conference on Computer-Aided Design, 2005.*, pages 1068–1075, 2005.

[28] 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.

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

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

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

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

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

[34] 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.

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

[36] Jing Chen and Silvio Micali. Algorand, 2017.