Modeling, Specification and Verification of Smart Contract Applications

Zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

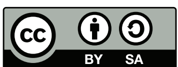
DISSERTATION

von

Jonas Schiffl

Tag der mündlichen Prüfung: 21. Oktober 2024

Referent: Prof. Dr. rer. nat. Bernhard Beckert,
Karlsruher Institut für Technologie
Koreferent: Prof. Dr. rer. nat. Wolfgang Ahrendt,
Chalmers University of Technology



This document is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License (CC BY-SA 4.0):
https://creativecommons.org/licenses/by-sa/4.0/deed.en

Acknowledgments

There are many people who supported me in many different ways during the time in which this doctoral thesis was created. I would like to express my gratitude here.

First, I thank my supervisor Prof. Dr. Bernhard Beckert, who gave me the opportunity to pursue my research within his group. His support in everything was invaluable.

I am also deeply grateful to Prof. Dr. Wolfgang Ahrendt for agreeing to be the second reviewer of my doctoral thesis, and for all the fruitful and inspiring discussions we have had.

Prof. Dr. Thomas Bläsius, Prof. Dr. Hannes Hartenstein, Prof. Dr. Anne Koziolek, Prof. Dr. André Platzer, and Prof. Dr. Ralf Reussner have all given me valuable feedback on my dissertation project.

I am also very grateful for the opportunity to work in an exellent group with great colleagues. I thank you all, roughly in order of appearance: Simon Greiner, Sarah Grebing, Mihai Herda, Markus Iser, Michael Kirsten, Prof. Shmuel Tyszberowicz, Jonas Klamroth, Lionel Blatter, Annika Vielsack, Wolfram Pfeifer, Florian Lanzinger, Philipp Kern, Joshua Bachmeier, Samuel Teuber, Romain Pascual, Debasmita Lohar, and Henriette Färber. I am also grateful to Simone Meinhart and Ralf Koelmel for always having our back in terms of administration and IT support.

Special thanks go to Mattias Ulbrich, whose excellence and enthusiasm got me hooked to the formal verification group, and Alexander Weigl, who has gone out of his way to help and support me.

I also want to thank my colleagues from other groups who have been great in discussions and collaborations: Frederik Reiche, Matthias Grundmann, Marc Leinweber, Oliver Stengele, Sebastian Friebe, Alisa Jung, Sophie Corallo, Sebastian Hahner, Asmae Heydari Tabar. Cheers!

I always got important input and feedback from the students I supervised. For this, I thank Alexander Linder, Engelhard Heß, Michele Massetti, Jan-Philipp Töberg, Jonas Wille, Jonas Kaiser, and Tim Holzenkamp.

I would like to thank the Karlsruhe Institute of Technology and the KASTEL project for enabling my research.

When doing research in the area of smart contracts and cryptocurrency, it is fundamentally important to cultivate a healthy skepticism. In this regard, I am grateful to Molly White, David Gerard, and Amy Castor, whose independent journalistic work I greatly admire.

Lastly, I want to thank all those who have become part of my life in other ways.

This includes the excellent people who play Ultimate in Karlsruhe. As representatives of all the cool kids in the MTV, I want to mention Kraut, Clara, and Tank for coaching the Mixed II team over the last years.

Furthermore, I am grateful for being a part of KIT Konzertchor. I thank our director, Nikolaus Indlekofer, and all the singers I have gotten to know and like over the years.

Thanks to all the friends who have made my life enjoyable! Jonas and Johanna; Micha; Moritz; Nori and Sebastian; my pandemic flatmates Stephi and Emma-Lilo; Sophie, Lori, Xenia, Lisa, Serkan, Ioana, Veit, Hölli, Adam, Julia, Marius; fellow singers Ana, Annegret, David, Jakob, Jan, Klaus, Moritz, Pia; fellow disc throwers Alu, Dollar, Jannik, Niklas; Geneviève, Kurt, David, Sarah, Marco.

My family has given me love and support for my entire life. Mama, Papa, Bruderherz: Thank you for everything!

Finally, thank you, Fanny. You make my life better in every way.

Summary

Smart contracts are programs which run in a blockchain network. They manage access to resources that are stored on the blockchain, such as cryptocurrencies or tokens representing real-world assets. Smart contracts are unique in that they allow anyone to run a program on a different computer, while still being certain about the execution semantics, and about what source code is being executed. This comes at the price of immutability: Once deployed, the source code smart contracts generally cannot be changed. Furthermore, the source code, including possible programming errors, is usually publicly available. This means that any vulnerability has a large probability of being exploited. Since smart contracts cannot be patched, it is very important that smart contracts are correct and secure upon deployment.

Indeed, formal analysis of smart contracts has been a very active research area. Many tools for static analysis and formal verification of different classes of properties have been developed. However, a long and ongoing history of vulnerabilities and exploits of smart contract applications shows that security in this field is very much a pressing issue. In the domain of smart contract applications, correctness properties are highly application specific, which places them out of reach of static analysis tools. Existing tools for formal verification, where developers can provide a specification, still lack the mechanisms to express many typical correctness and security properties.

In my thesis, I contribute to the field of formal methods for smart contracts by creating the Scar approach for model-driven development of correct and secure smart contract applications. Before implementing an application, smart contract developers first describe it in terms of an intuitive, platform-agnostic metamodel. Within this model, they can also specify high-level security and behavioral correctness properties, and check whether the model contains any inconsistencies. Finally, a combination of code generation, static analyses, and formal verification of automatically generated formal annotations leads to an implementation that is correct and secure w.r.t. the initial model.

Main Contributions

My main contribution consists of a metamodel of smart contract application with formal specification of behavioral correctness. On this foundation, I develop a capability-based approach for specifying and verifying security properties. Furthermore, I propose a practical notion of liveness properties in smart contracts, and a way of specifying liveness properties that is intuitive and results in practically feasible proofs. Apart from the abstract metamodel, I also develop platform-specific methods that are required to translate a model into a correct and secure implementation.

An Overview of Smart Contracts and Formal Verification Before introducing the Scar approach, I give an overview of the history and the technological foundations of smart contracts. Furthermore, I present a characterization of smart contracts which forms the base for the formalization developed in the later chapters.

Furthermore, I give an introduction to the research area of formal methods for smart contracts. I give an overview of the main approaches, possible classifications, and of individual tools that are of interest.

The Scar Approach As the base for the model-driven Scar approach, I develop a metamodel for smart contract applications. Structurally, an applications consists of a set of contracts, which in turn have state variables and functions. State variables are typed according to a simple type system inspired by the Solidity language, which I envision to be the main practical use case of the approach.

In addition to the structural description, Scar also provides a way to describe the behavior of an application: The ScarML functional specification language lets developers specify the behavior of functions in terms of pre- and postconditions. Correctness properties of the individual contracts of an application can be specified in the form of invariants.

Furthermore, Scar provides a process to derive a source code implementation from a given model: A source code skeleton is automatically generated, with the functions annotated with formal specification. When the functions are implemented such that they satisfy the generated specification, the developer can be sure that their implementation matches the intended behavior encoded in the Scar model.

The Scar metamodel is an abstraction in several ways: First, it abstracts from the smart contract platform and the programming language that is going to be used. Importantly, this makes the model independent from the details of

the underlying platform, *e.g.*, the Ethereum Virtual Machine and programming language, *e.g.*, Solidity. The abstraction also reduces the necessary effort in case of changes in a platform or programming language. In the relatively new field of smart contracts, these changes happen frequently, and can invalidate tools which operate directly on a specific version of a language. In a similar way, the metamodel is also an abstraction from tool- or version-specific specification languages.

Overall, the SCAR approach provides a simple way of describing a smart contract applications and application-specific correctness properties. Developers can work on an intuitive abstraction and verify that their implementation conforms to the properties specified on the model.

Model-driven Security Smart contract applications manage resources in the form of cryptocurrency or tokens representing other assets. Therefore, security is eminently important in the smart contract domain, and functional correctness problems often overlap with security issues. Although a plethora of tools especially for static analysis have been developed, security continues to be a concern. In part, this is because security is often an application-specific property.

As a contribution to smart contract security, the SCAR approach is extended: Developers are required to specify security properties upfront on the abstract model of the application. They are then provided with a process resulting in an implementation that is correct with respect to these security properties.

I identify three security-relevant capabilities: Modifying the state of an application, calling functions, and transferring currency. Furthermore, I define the actors to which these capabilities are attached, namely accounts and smart contract functions. In the metamodel, actors can be summarized in roles. I develop a set-based semantics for the capabilities, and define notions of consistency and least privilege on the model level. Furthermore, I develop a process for developing an implementation that is correct w.r.t. a given model. This process consists of code and annotation generation, static analysis and formal verification. The process is instantiated for the Solidity programming language.

Liveness Properties An important part of behavioral correctness of smart contracts is described by liveness properties, *e.g.*, in the form of guarantees that an action will lead to some desirable result in the future. This kind of property can be hard to specify and verify, in particular because application-specific fairness assumptions w.r.t. function invocation and the behavior of other parties are usually necessary for any liveness proof to succeed.

In my work, I develop an approach for modeling and verifying liveness properties. First, I analyze smart contract liveness properties discussed in the literature. I find that the smart contract paradigm of decentralization and trustlessness induces a certain, commonly occurring kind of liveness property which is tied to the ability of an agent to induce a state change, *e.g.*, a transfer of resources. I introduce the ScarTL specification language, which contains formal specification concepts which capture this notion of liveness, and extend the Scar metamodel accordingly. Finally, I develop an approach for verifying liveness properties on the model level by deriving them from the function contracts and invariants that are already part of the model.

Specification Languages for Frame Condition In order to obtain an implementation which is correct and secure w.r.t. a given model, my approach requires some domain-specific methods. These include automatic generation of code and formal specification. In particular, my approach generates frame conditions, *i.e.*, a specification of what a smart contract function cannot or will not do. These conditions are useful to modularize functional correctness proofs, but they are also necessary for the capability-based security approach. I develop specification languages for frame conditions in two specific platforms, namely Ethereum and Hyperledger Fabric. The languages are based on the theory of dynamic frames.

Outline

Before presenting my contributions around the Scar approach, a contextualization is appropriate. Therefore, Chapter 1, gives an overview of the history of smart contracts, and of their characteristics. Since the term “smart contract” is not clearly defined in the literature, the characterization is also intended to be a definition of what constitutes a smart contract for the purposes of this work.

Chapter 2 motivates the necessity of formal analysis for smart contracts, and give a comprehensive overview of the existing tools and methodologies.

In Chapter 3, the Scar approach is introduced, a model-driven development process based on formal verification. Scar enables developers to formalize the structure and intended behavior of an application without having to focus on specifics of a platform or programming language.

The Scar formal model serves as the basis for specification and verification of application-level correctness and security properties. In Chapter 4 the Scar metamodel is extended to include roles and capabilities. This allows developers to specify and verify application-level access control policies. Chapter 5 analyzes the prevalence of liveness properties in smart contract applications. As

a consequence, Scar is extended to include common temporal properties, and approaches for their verification.

In Chapter 6, two specification languages for frame conditions are presented, one for the Ethereum platform, and one for Hyperledger Fabric contracts written in Java. Chapter 7 concludes.

New and Previously Published Material

Some of the material in this thesis has been previously published. Some other parts are not yet submitted or have been written solely for this thesis. This section is dedicated to attribution.

The first two chapters have been written exclusively for this thesis. The third chapter, which describes the Scar metamodel and overall approach, is scheduled for submission as a journal paper.

Chapter 4 is largely based on [SWB23], with the main addition being the evaluation of the capability-based approach on the Palinodia case study. Chapter 5 is based on [SB24], with some additional work in the sketched verification approaches. Chapter 6 is based on [BS20].

Contents

Main Contributions vi
Outline viii
New and Previously Published Material ix

Contents **x**

1 An Introduction to Smart Contracts **1**
1.1 Historical Overview 1
1.2 Characterization of Smart Contracts 8
1.3 Smart Contract Platforms 12
1.4 Applications 18
1.5 Hype and Criticism 25
1.6 Conclusion 27

2 Formal Verification of Smart Contracts **29**
2.1 The Need for Formal Methods 29
2.2 Common Errors and Vulnerabilities 31
2.3 Classification of Verification Approaches 33
2.4 Tools 36

3 The Scar Approach **47**
3.1 Model-driven, Verification-based Development 48
3.2 Scar Overview 50
3.3 Scar Application Metamodel 54
3.4 Scar Semantics 58
3.5 Scala Implementation of the Scar approach 67
3.6 Consistency between Model and Code 68
3.7 Applications of Scar 74
3.8 Evaluation 76
3.9 Conclusion 83

4 Capability-based Security for Smart Contract Applications **85**

4.1 Modeling Capabilities for Smart Contract Applications 86
4.2 Semantics of Capabilities 91
4.3 Analyzing Model Consistency 93
4.4 A Secure Solidity Implementation 95
4.5 Evaluation 100
4.6 Conclusion and Future Work 104

5 A Practical Notion of Liveness in Smart Contract Applications 107
5.1 Liveness Properties in Smart Contracts 108
5.2 ScarTL 111
5.3 Verification 115
5.4 Evaluation 119
5.5 Conclusion and Future Work 123

6 Specification of Frame Conditions for Solidity and Hyperledger Fabric 125
6.1 Introduction 125
6.2 Motivation 126
6.3 Frame Conditions for Solidity 128
6.4 Frame Conditions for Hyperledger Fabric 136
6.5 Towards Analysis and Verification of Frame Conditions 140
6.6 Related Work 142
6.7 Conclusion 143

7 Conclusion 145
7.1 Summary 145
7.2 Future Work 147

List of Figures 149

List of Tables 150

Bibliography 151

Chapter 1

An Introduction to Smart Contracts

Smart contract applications are a relatively new phenomenon. This chapter will first give an overview of the history of the concept, mainly in terms of technology, but also in terms of the societal perspectives from which smart contracts originate. Then, we give an introduction to the characteristics of smart contracts. Furthermore, we describe existing platforms, and introduce some use cases and examples.

1.1 Historical Overview

This section presents an overview of the technological precursors and preliminaries of smart contracts.

Blockchain and Bitcoin

The concept of smart contracts as viewed in this work is tightly bound to the concept of a *blockchain*. A blockchain is an append-only data structure which is replicated across a decentralized network. New data is added according to a consensus protocol. The security of the system is guaranteed by cryptographic primitives.

Public-key Cryptography Public-key cryptography, also known as asymmetric cryptography, is a fundamental technology in modern information security, underpinning a wide array of protocols and applications that ensure confidentiality, integrity, and authenticity of digital communications. This crypto-

graphic paradigm was first conceptualized in the groundbreaking work of Diffie and Hellman in 1976 [DH76], who introduced the notion of key pairs to solve the key distribution problem inherent in symmetric cryptography.

In public-key cryptography, each participant possesses a pair of related keys: a public key and a private key. The public key, as its name suggests, is distributed publicly and used by others to encrypt messages intended for the key pair owner. Conversely, the private key is kept secret and is used to decrypt these messages. The security of this system relies on the computational infeasibility of deriving the private key from the public key, even though they are mathematically linked. This asymmetry eliminates the need for a secure channel to exchange keys, a significant limitation in symmetric key cryptography.

The most widely used public-key algorithms include RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography). RSA, based on the practical difficulty of factoring large composite numbers, was one of the first public-key cryptosystems and remains widely implemented. ECC, which is based on the properties of elliptic curves over finite fields, offers similar security levels to RSA but with significantly smaller key sizes, leading to efficiency gains in both computational overhead and memory usage.

Two fundamental applications of public-key cryptography are encryption and digital signatures. Messages to some agent are encrypted with this agent's public key, and can only be decrypted with the associated private key. Conversely, when a private key is used to create a digital signature of a message, the sender's public key can be used to validate that the message originates from the sender (or at least someone who knows the sender's private key).

Hash Functions Hash functions originate in checksums, *i.e.*, calculations on some data with the purpose of detecting errors that may have been introduced.

A hash function is a deterministic algorithm that maps an input of arbitrary length to a fixed-size value, typically referred to as the hash value or digest. This transformation is designed to be fast and to produce outputs that appear random, despite being derived from the inputs in a deterministic manner.

Cryptographic hash functions have the following additional properties:

- Pre-image resistance: given a hash value, it is computationally infeasible to find the input
- Second pre-image resistance: given an input, it is difficult to find a different input that produces the same hash value
- Collision resistance: it is hard to find any two distinct inputs that hash to the same output

Cryptographic hash functions, such as SHA-256 (Secure Hash Algorithm 256-bit [NIS02]) and MD5 (Message Digest Algorithm 5, [Riv92]), are specifically designed to meet these security properties. SHA-256, part of the SHA-2 family developed by the National Security Agency (NSA), produces a 256-bit hash value and is widely used in various security protocols, including SSL/TLS for secure internet communications. Despite its historical popularity, MD5 is no longer considered secure against attackers due to vulnerabilities that allow for the construction of collisions [XLF13].

In the context of digital signatures, hash functions contribute to the process by which a document is signed. The document is first hashed, and then the resulting hash value is encrypted with a private key to produce the digital signature. This process ensures that the signature is both compact and sensitive to any changes in the document, since even a minor modification will result in a drastically different hash value due to the avalanche effect.

Hashcash Hashcash [Bac02] is a proof-of-work (PoW) algorithm originally proposed by Adam Back in 1997 as a mechanism to reduce email spam and denial-of-service attacks. The core concept involves requiring a computational effort to be performed by the sender of a message, which acts as a deterrent against the mass sending of unsolicited messages. This computational task involves solving a cryptographic puzzle that is simple to verify but computationally intensive to find: In Hashcash, the sender of an email must include a header containing a hash value that meets certain criteria. Specifically, the hash value must have a specified number of leading zeros, a requirement that necessitates iterating through numerous inputs to find a suitable hash. The original implementation was based on the SHA-1 hash function, but modern implementations use more secure alternatives like SHA-256 due to vulnerabilities in SHA-1.

An important property of Hashcash is its ability to scale the difficulty of the proof-of-work, making it adaptable to varying levels of computational power and desired security. The computational effort required to solve the puzzle can be adjusted by altering the number of leading zeros required in the hash output, allowing the system to remain effective as processing power evolves.

Bitcoin The Bitcoin protocol builds on the above technologies to provide a pseudonymous, decentralized, peer-to-peer transaction platform. It was introduced in 2008 by an anonymous entity known as Satoshi Nakamoto in a whitepaper titled "Bitcoin: A Peer-to-Peer Electronic Cash System" [Nak08]. Bitcoin is the first decentralized cryptocurrency, and the first implementation of a blockchain. Bitcoin addresses several fundamental challenges in digital cur-

rency, such as the prevention of double-spending without relying on a central authority.

Bitcoin was conceived as an electronic payment system based on cryptographic proof instead of trust or regulation. The whitepaper outlines the design principles and mechanisms that underpin Bitcoin. Central to Bitcoin's concept is the idea of a decentralized network where transactions are validated and recorded by a distributed network of participants rather than a central entity. Bitcoin's decentralized nature aims to make it resilient against censorship and centralized control, which are perceived to be prevalent issues in traditional financial systems.

Bitcoin operates on a blockchain, a public ledger that records all transactions in a chronological order. Each block mainly consists of a set of transactions, but also contains a reference to the previous block, forming a chain of blocks. The blockchain is maintained by a network of nodes, each containing a copy of the ledger. This distributed ledger technology ensures transparency and immutability of transaction records.

Each user of the Bitcoin blockchain has a pair of cryptographic keys—a public key, which serves as the address to receive bitcoins, and a private key, which is used to sign transactions. A transaction includes inputs (references to previous transactions) and outputs (addresses and amounts).

Each Bitcoin transaction is digitally signed using the sender's private key and broadcast to the network. Transactions are grouped into blocks in the form of Merkle trees [Mer88]. The root hash of the tree is part of the block header. Figure 1.1 illustrates the contents of blocks.

New blocks are appended to the blockchain by so-called miner nodes. Mining, *i.e.*, the process of creating a new block, happens in the context of the Proof-of-Work (PoW) consensus algorithm, inspired by Hashcash. To create a valid block, mining nodes must find a random value (a *nonce*) such that the hash of the entire block starts with a number of zeroes. This number determines the mining difficulty and is scaled such that a new block is created every ten minutes, on average. Once a block is successfully mined, it is broadcast to the network and added to the other nodes' local copies of the blockchain. The miner is rewarded with newly created bitcoins and transaction fees. In case of competing blocks, the protocol stipulates that nodes should accept a longer incoming chain.

The PoW mechanism serves multiple purposes. The difficulty of PoW makes it computationally infeasible to alter the blockchain, as an attacker would need to re-mine all subsequent blocks. Furthermore, PoW allows any participant with sufficient computational power to participate in mining, promoting a decentralized network. Lastly, miners are incentivized through block rewards and transaction fees, ensuring continuous network availability.

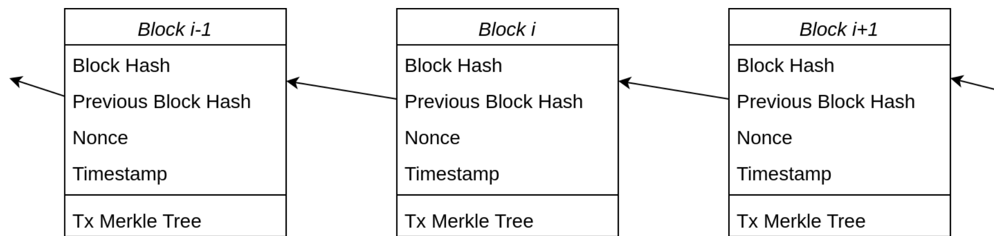


Figure 1.1: Illustration of the blockchain data structure

Bitcoin Script Bitcoin Script [Bra19] is a programming language integrated into the Bitcoin protocol. It is intended to automate the validation and execution of transactions on the Bitcoin network. Bitcoin Script is an assembly-like stack-based language designed specifically for the purpose of defining transaction conditions.

Importantly, Bitcoin Script is Turing incomplete, meaning it lacks the capability to perform arbitrary computations. This limitation is a deliberate design choice to mitigate risks such as infinite loops and other vulnerabilities that could arise from more complex programming constructs.

The language comprises a set of predefined operations (opcodes) that include basic arithmetic, logical operations, and cryptographic functions. These opcodes are used to define the conditions under which a Bitcoin transaction can be considered valid. For instance, common operations include "CHECKSIG" for verifying digital signatures and "HASH160" for executing hash functions. The combination of these operations allows the construction of various transaction types, from simple payments to more complex multi-signature transactions.

Other consensus mechanisms Bitcoin's proof of work consensus mechanism is extremely wasteful in terms of power consumption, especially when the price of Bitcoin tokens is high and miners are incentivized to work on adding new blocks even at high difficulties. To avoid this, several other consensus mechanisms have been proposed and implemented.

Proof of Stake In proof of stake consensus, the creator of the next block (often called "validator" in this context) is determined by network participants according to the share of tokens they are willing to "stake", *i.e.*, commit as collateral. In the system used, *e.g.*, by Ethereum, a verifiable random function weighed with the stake sizes chooses the next validator. In "delegated" proof of stake, token holders vote for the next validator.

This introduces some degree of centralization, since participants with more financial resources have a higher power of decision-making.

Permissioned Mechanisms Other consensus mechanisms that were proposed for blockchain networks have an even larger degree of centralization built into them, as the set of validators is limited, or validators have to be known so that misbehavior can be punished. This includes the Byzantine Fault Tolerance algorithms used in Hyperledger Fabric, as well as the "Proof of Authority" system promoted by some other platforms.

Stored Procedures

When viewing blockchains as databases, the closest historical analogy to smart contracts are *stored procedures*. Stored procedures are precompiled user programs, written in a query language, which run inside the database server[Ris09]. Their earliest occurrence was in IBM System R, the first implementation of the SQL database query language [Cha+81].

Stored procedures typically do not only execute a query for the application program, but they manipulate other data in the database. They encapsulate, and thereby enable the re-use of, common business logic. At the same time, they may increase performance on the side of the calling application. They may also be advantageous if the communication between the application program and the database is slow or unreliable, since calling a stored procedure requires just a single message, as opposed to a string of messages and back-and-forth communication. Regarding security, stored procedures can be access controlled directly, like other objects in the database.

However, stored procedures have also been regarded as bad practice in software engineering. One important point of criticism is that the separation of business logic and data is violated by procedures which are stored as data, and that this leads to difficulties in version controlling, testing and maintaining the code. They also lead to a tight coupling between application and database, which makes changes to either of the two more challenging. Lastly, even though stored procedures can be access controlled, it is easy to introduce security vulnerabilities in them, *e.g.*, when the effects of a procedure are not fully understood by a developer.

When a blockchain is viewed as a database, smart contracts are largely equivalent to stored procedures. They are compiled programs, stored within the database they manipulate. The differences are due to the different environments in which they run, particularly by the emphasis on decentralization in smart contract platforms. In contrast to stored procedures, calling a smart contract's functions is typically the only way the database can be changed. Furthermore, smart contracts typically cannot be changed at all, or only at the cost of some decentralization. While access control to stored procedures can be

fine-grained, smart contract functions are, in principle, open for every network agent to call them – they have to do their own access control.

Szabo's Smart Contracts

The term "Smart Contract" originates in a 1997 paper by Nick Szabo [Sza97]. In it, Szabo envisions contracts which run on computers. They combine user interfaces and protocols (especially cryptographic protocols) to formalize and secure relationships in networks.

According to Szabo, there are three main objectives of smart contract design: First, observability, *i.e.*, "the ability of the principals to observe each others' performance of the contract"; second, verifiability, *i.e.*, "the ability of a principal to prove to an adjudicator that a contract has been performed or breached"; and third, privity, *i.e.*, "the principle that knowledge and control over the contents and performance of a contract should be distributed among parties only as much as is necessary for the performance of that contract".

Szabo discusses various cryptographic techniques, such as digital signatures, mixing, and "post-unforgeable transaction logs". The latter he proposes to realize via one-way hash functions, reminiscent of the way hashing was later to be used in blockchain networks.

In his paper, Szabo considers a wide range of possible applications for digital smart contracts, including credit, digital rights management, and payment systems. He also envisions autonomous agents who can assume the role of a contract party, or of an adjudicator to a contract.

Szabo's ideas and the newly-coined term "Smart Contract" gained prominence only in 2016, when the developers of the Ethereum blockchain (cf. Section 1.3) adopted the term.

Ethereum

The current concept of smart contracts as computer programs in decentralized networks was heavily influenced or, one might say, introduced with the description and implementation of the Ethereum platform and the Ethereum virtual machine in 2016. The Ethereum platform is described in Section 1.3. In the following, we characterize the type of smart contracts, platforms and applications that are analyzed in this thesis.

1.2 Characterization of Smart Contracts

There is no established consensus on the precise definition of what constitutes a smart contract. In the wake of Ethereum's success, many different smart contract platforms were created, with very different goals and perspectives.

These perspectives range from simply regarding smart contracts as "programs stored on blockchains" to "programs running in conjunction with distributed ledgers" [And+18]. Others still view smart contracts as programs that are meant to automatically enforce legal contracts, or highlight the fact that smart contracts are programs that can be run on someone else's computer without relinquishing control over the execution.

To give a clearer picture of the domain this work is concerned with, this section will give an introduction of the characteristics of smart contracts, as well as highlight some consequences of these characteristics.

Defining Characteristics

We present an overview of the main characteristics of smart contracts, smart contract applications, and smart contract platforms for the purposes of this work.

Turing-complete Programs First of all, for the purposes of this work, smart contracts are computer programs written in turing-complete programming languages.

Many non-turing-complete smart contract languages have been developed (cf. Section 1.3), starting with Bitcoin Script (see Section 1.1). Some of these have been designed with the explicit goal of simplifying security analysis and formal verification – which becomes much easier with more restricted languages.

Here, however, only smart contracts in turing-complete languages will be considered. This is due to two main reasons: First of all, the vast majority of smart contract applications have been written in turing-complete languages. While this does not prove the necessity of turing-complete smart contract languages (and may in fact just be an artifact of the prominence of the Ethereum platform), common usage is still a reason to focus on these platforms. Secondly, this work is aimed at (potentially large-scale) *applications* in a broad sense. The restrictions of non-turing-complete languages do not fit this vision.

Decentralized Execution Environment Smart contracts are characterized by running in a decentralized environment. A system is "decentralized", for our purposes, if the authority over the system does not rest with a single entity, but is spread among different groups or entities. This contrasts with a system being

distributed, in that a system may be distributed over many different machines in different locations, but still be controlled by one entity. Decentralization always has an aspect of sharing decision-making power; in that sense, it is political.

In computer networks, the decentralization of a platform also determines to what degree a participant or user of the platform can be certain of the execution environment. In a centralized setting, by definition, it is possible for a single entity to make significant changes to the operation of a platform. In a decentralized setting, such changes require a lot of coordination and are thus harder to make. This gives participants a degree of certainty about the environment they find themselves in.

Open World Related to decentralization, smart contracts are furthermore characterized by existing in a public setting in which there are no general limitations to participation. In a blockchain platform, this is exemplified by the fact that anyone can run a node to create and validate new blocks, and that anyone can submit transactions and expect them to be carried out eventually. Furthermore, the code of smart contracts must be publicly available in some form (in blockchain platforms, the executable code is stored directly on the ledger, like a stored procedure in a database).

There have been a number of so-called *private* blockchain platforms. Hyperledger Fabric [And+18] (see Section 1.3) differentiates in two dimensions: "permissioned / permission-less" describes whether the set of participants who are allowed to run a node is limited, and "public / private" refers to whether any network participant is allowed to make transactions.

In this work, we focus on systems that are, by this definition, public and permission-less: Everyone can run a node, and everyone can make transactions. This includes untrusted and adversarial actors. Applications operating in such systems need to take this into account, in particular when considering questions of security.

Nevertheless, the methods developed in this thesis may still be useful for platforms which do not fit this description, if adapted accordingly.

State Smart contract platforms, despite being decentralized in nature, create a centralized view of the overall state (often called the *ledger state* or *blockchain state*, depending on the underlying data structure). This state is publicly visible to everyone on the platform. It can be viewed abstractly as a set of *storage locations*.

The central defining characteristic of a smart contract is that it defines a namespace within this state, *i.e.*, a set of storage locations that can only be changed by interacting with the smart contract.

For this, smart contracts expose a set of *functions*, which can be called by all participants of the platform, including other contracts. Importantly, calling a function of a smart contract is the only way to change the state in this contract's namespace.

Conversely, a smart contract's functions are also the only point where access to the contract's state can be restricted – there is no platform-level access control mechanism. If some functionality is supposed to be limited to a subset of actors, the corresponding checks can only be carried out in the function itself.

Transactionality, Determinism, Termination In smart contracts, function calls are transactional in nature: A call either succeeds and results in some change to the overall state, or it reverts completely, without any effect. This includes nested function calls: If any call in a chain fails, no changes take effect.

Furthermore, to ensure reproducible behavior across the decentralized network, smart contract functions must be deterministic. Therefore, any potential sources of nondeterminism, like parallel computation or floating-point arithmetic, must be ruled out in some form.

For similar reasons, smart contract functions must terminate – otherwise, they interrupt the network. Some platforms ensure termination by restricting the programming language accordingly, but the turing-complete platforms considered in this work have to enforce termination in other ways.

Accounts Participants in smart contract platforms can be real-world persons or organizations, who need to be identifiable in some way and to have a way to authenticate themselves (usually by a cryptographic key pair). Besides these external participants, contracts themselves have also been viewed as actors, due to their ability to call other contracts, but also due to their (relative) immutability. This gives rise to the notion of *accounts*, which may either represent a person or an organization in the real world (*external account*), or a smart contract (*contract account*).

Cryptocurrency Lastly, in this thesis, we focus on smart contract platforms that have a built-in *cryptocurrency*, *i.e.*, fungible digital tokens that are ascribed some monetary value and that can be transferred between accounts. Cryptocurrencies have been a defining feature of the vast majority of smart contract platforms, and arguably also played an important role in the hype around them. Although some platforms de-emphasize cryptocurrencies (*e.g.*, Hyperledger Fabric, cf. Section 1.3), the most prominent platforms have them built in to a degree that cannot be ignored in any analysis of smart contract cor-

rectness and security. This includes special data types for currency, built-in account balances recorded in the native currency, as well as special primitives for transferring currency between accounts.

Implications

In summary, smart contracts are turing-complete programs which expose some functionality to the participants in an open, decentralized network. They operate on a shared central data structure (the ledger), on which they define a namespace. Within this namespace, the state of the ledger can only be changed by interacting with the smart contract's functions. These functions are deterministic and transactional: Unless there is an error, a function call terminates and its effects are committed to the ledger atomically. Participants in the platform can call these functions knowing what code is going to be executed (and being able to validate the execution afterwards), despite the execution happening on a computer not under their control.

In the following, we highlight some implications of this characterization, as well as some further definitions and perspectives.

Smart Contract Applications The terms "smart contract" and "smart contract application" are not clearly delimited in the literature. For the purposes of this work, there is not a huge difference: A smart contract is one program which exhibits the above characteristics, and a smart contract application is a set of smart contracts deployed to fulfill a particular purpose.

The term "Decentralized Application" is often used to describe applications of which smart contracts are a part, but which also contain other parts, like client software or user interfaces. These parts are not considered to belong to a smart contract application here.

Immutability A corollary of the decentralized paradigm and the guaranteed execution environment is that smart contracts must be, and usually are, very hard to change. If participants should be able to rely on what code is executed when they call a smart contract function, then this function cannot be subject to change.

This introduces its own set of problems. Changing a program is necessary to fix errors in the program, so there is a direct trade-off between the guarantees of the decentralized execution environment and the ability of developers to release patches to their programs. While there are solutions that increase patchability (*e.g.*, proxies, cf. [WZ18]), these solutions invariably lead to a loss

in decentralization, since the power to decide whether a patch is applied has to be handed to some person or group.

Smart Contracts and Law Nick Szabo's idea of smart contracts (see Section 1.1) was directed at taking contracts out of the domain of law as much as possible by translating them into digital objects. This attitude is echoed in the "Code is Law" perspective often applied to Ethereum and other smart contract platforms. This view holds that a smart contract, once published, defines its own law: All possible uses are allowed, whether intended by the developers or not.

However, in the real world, this view has not prevailed, and attackers who used smart contracts in ways not intended by their developers have routinely been prosecuted by those developers or the platform managers. The domain of law has so far concurred and often found the perpetrators of such attacks guilty of fraud or manipulation (as, for example, in the case of Avraham Eisenberg [Off24]).

Others have taken a compromise view, arguing that *specification* is law (cf. [Ant+22]), *i.e.*, there should be an abstract description of the intended behavior of a smart contract. If an update conforms to this description, it is admissible. Of course, this leaves the question of how the intended behavior is described, and how to test whether an implementation conforms to it. The approach proposed in this work attempts to answer this question.

Security As a consequence of their unique characteristics, smart contracts are exceptionally attractive targets for attackers: They usually manage money (in the form of cryptocurrency) or other digital tokens which represent valuable assets. Therefore, any error in the program can, with a high probability, lead to a security vulnerability. This criticality is exacerbated by the fact that the source code (or byte code) of smart contracts is public and cannot be fixed easily, so that any existing error is likely to be found and exploited by attackers.

1.3 Smart Contract Platforms

This section gives an overview of the most important smart contract platforms. A heavy focus is on the Ethereum platform and its Solidity programming language, which dominates the space both in public attention and in the value of the managed tokens.

Ethereum

Ethereum was the first major smart contract platform, and arguably introduced the very definition of what is currently understood to be a smart contract (which this work also mostly adheres to). Although the size of a decentral platform can be hard to measure, there seems to be agreement that Ethereum remains by far the largest platform by the value of the tokens traded on it, and by the value of the contracts in the network[Coi24b] [Coi24a].

The Ethereum whitepaper ([But13]) was published in 2013, with the technical specification following in 2014 [Woo14]. Ethereum founder Vitalik Buterin adopted the term "smart contract" to refer to computer programs stored on a public blockchain. This interpretation proved to be very influential, mostly due to the predominance of Ethereum in the volume of trading and news coverage. To this day, the term "smart contract" is often used to refer to a program on the Ethereum blockchain, usually one written in the Solidity programming language. This usage remains pervasive even in academic writing. For example, a recent overview paper on smart contract security ("Smart Contract and DeFi Security: Insights from Tool Evaluations and Practitioner Surveys") [Cha+23] is solely concerned with Ethereum smart contracts.

There are two types of accounts on the Ethereum blockchain: External accounts, identified by a public key, represent a real-world entity (like a person or company). Contract accounts represent the smart contracts. Every account has a balance in the built-in Ether cryptocurrency.

The central data structure of the Ethereum network is the Ethereum blockchain, which is continually updated by nodes running an Ethereum client, using a proof of stake consensus mechanism. A client stores the current state of the network, the "world state", in what is called the *state database*. It also contains an implementation of the Ethereum Virtual Machine (EVM). The EVM takes programs written in EVM bytecode, and computes their effect based on the current state.

In Ethereum, state changes are effected by calling a public function of a contract. A successful function call which results in such a state change is called a transaction. When an account wishes to make a transaction, it submits the name of the function and the function parameters along with a digital signature to the *mempool*, the set of pending transactions. From there, mining nodes can choose transactions to be included in the next block that is appended to the blockchain. For each transaction, the mining node computes its effects, updates the state accordingly, and broadcasts the block to the network.

Transactions require a payment in cryptocurrency, called *gas*. The size of the payment depends on the complexity and the storage requirements of the called function – each bytecode instruction consumes a pre-defined amount of

gas, and if the gas sent along with a transaction is not sufficient, the transaction is reverted and no state change happens (apart from the gas fees, which remain with the mining node). The gas mechanism is Ethereum's way of forcing termination: Since each function call can only consume a finite amount of gas, execution is inherently limited. Non-terminating functions abort when all their gas is consumed, and participants are disincentivized to call such functions, since their money will not be refunded.

The term *transaction* also hints at the transactionality of Ethereum function calls: A function call either terminates successfully, or it reverts without changing the state. The only exception to this is the gas mechanism, where account balances do change if a function call is reverted.

Smart contracts for the Ethereum platform are usually written in high-level programming languages and then compiled to EVM bytecode. The dominant language is Solidity. The only other higher-level language currently under development is Vyper [Tea24].

Solidity Solidity [Tea] is a "contract-oriented" language developed specifically for smart contract applications on the Ethereum platform. It has been the de-facto standard high-level programming language on Ethereum since its release in 2015.

Solidity is an imperative, strongly typed language. The type system consists of value types, which are always passed by value, and reference types, which can be manipulated through multiple names. Value types include literals, booleans, strings, addresses (which represent an account), enums, and integers. In order to enable memory-efficient programming, Solidity provides individual types for integers of all bit lengths divisible by 8, up to 256. The type of functions is also a value type in Solidity.

Reference types include arrays, C-like structs, and key-value mappings that map primitive-typed keys to values. When declaring variables or parameters of reference types, the location of the data has to be stated explicitly: The *calldata* location refers to the non-modifiable location of function arguments. Variables of reference types can exist in the *memory* location, which exists only for the time of the external function call, but can be modified (unlike *calldata*). The third possible location is *storage*, which refers to the persistent storage of the Ethereum blockchain.

Functions in Solidity can be visible internally (*i.e.*, from within the contract where they reside), externally (*i.e.*, they are part of the contract's interface and can be called by anyone on the platform), or both. The respective visibility modifiers are `private` (only visible to the contract where the function is defined) and `internal` (includes inherited contracts), `external`, and `public`. Functions

take a list of parameters and can return zero or more named values. Their behavior can be restricted by modifiers. Built-in modifiers are `view` for functions that do not modify the state and `pure` for functions that neither read from nor write to the state. Solidity also offers user-defined function modifiers, which store pieces of code for re-use.

The call environment of a function consists of the calling account, the amount of Ether transferred with the call, the time of the call, and the current block number (among others). These environment variables can be accessed from within a function body with the `msg.sender`, `msg.value`, `block.timestamp`, and `block.number` keywords.

Solidity has two built-in convenience functions for checking boolean conditions, `require` and `assert`. They differ in the kind of error they are meant to catch: `require` is meant to be used for expected errors, *e.g.*, input validation or access control. When the condition in a `require` statement evaluates to false, an error is created, and the function reverts (*i.e.*, no state change happens). `assert` is supposed to be used for catching internal errors. As per the Solidity documentation, a correct contract should never violate an assertion. If an `assert` fails, it causes a Panic error. The effect is similar in that the execution of the function is halted and no changes to the state are applied.

Since `require` and `assert` statements represent assumptions and assertions, respectively, formal verification tools often treat them as specification. For example, the SOLC-VERIFY formal verification tool treats them as pre- and postconditions. Several assertion checking tools also build on them (see in-depth description of verification approaches in Section 2.4).

Figure 1.2 shows a basic Solidity contract with 2 state variables, `owner` of type `address`, and `numbers`, which is a mapping from the address type to integers. Next, there is a user-defined function modifier `onlyOwner`, which checks whether in the current context the address of the caller is equal to the `owner` address. Here, the underscore symbol "`_`" is the placeholder for the function body. A function with the `onlyOwner` modifier will first check the requirement and then proceed to the function body. The only function in the contract, `write`, is modified in this way. It takes two parameters. After the access control check due to the modifier, it checks whether the `numbers` mapping already contains data at the given address and returns this information as a boolean variable named `res`. If no data was stored yet, the new data is inserted.

Solana

Solana [Yak18] is a blockchain-based smart contract platform. It was introduced in 2018 and claims better transaction throughput and less transaction costs than Ethereum. Currently, it is one of the largest platforms according to the value of

```
contract SimpleStorage {
    address owner;
    mapping(address => int) numbers;

    modifier onlyOwner() {
        require(msg.sender == owner,
                "Only owner may call this.");
        _;
    }

    function write(address a, int i) public onlyOwner
       returns (bool res) {
        if (numbers[a] != 0) return false;
        numbers[a] = i;
        return true;
    }
}
```

Figure 1.2: A basic Solidity smart contract

tokens it manages[Coi24b]. Its consensus mechanism is a variant of delegated proof of stake (cf. Section 1.1).

Solana smart contracts are compiled to a variant of Berkeley Packet Filter (BPF) bytecode. Therefore, they can, in principle, be written in any language supported by the LLVM framework. The most commonly used programming language is Rust. In order to avoid non-determinism and to adapt to the single-threaded environment that smart contracts are run in, some Rust features cannot be used in Solana smart contracts. These include the randomness and time libraries, as well as libraries for concurrent programming[Sol24a].

Unlike Ethereum, Solana strictly distinguishes data and (stateless) program code. Accounts can be of either type, and programs are passed pointers to data locations. Data accounts have owner accounts in Solana, which can be any program account, a wallet account representing a natural person, or a multisig wallet representing a set of actors. In order to decrease the cryptocurrency balance of a data account, or to change the stored data, a signature of the owner account must be included in a transaction. Furthermore, Solana smart contracts can be marked as upgradeable, allowing the owner account to change the code.

Other platforms

In this section, other, smaller smart contract platforms will be mentioned. In particular, platforms are highlighted that emphasize formal verification of their

smart contracts.

Concordium Concordium [Con24] is a public, permissionless blockchain platform designed with a focus on identity and privacy but at the same time on regulatory compliance. Before participating in the network, actors have to undergo identity verification.

Concordium smart contracts are compiled to Web Assembly, so they can be written in a number of different higher-level languages. Currently, Like Solana smart contracts, they are mostly written in Rust.

Concordium has also been the focus of research efforts in the areas of cryptography and smart contract verification (see Section 2.4).

Aptos On the Aptos blockchain [Fou24a], developers can deploy smart contracts written in the Move programming language [Zho+20].

Move is *resource-oriented*, *i.e.*, its type system makes it possible to create data that behaves like a resource in that it cannot be copied or deleted, but only transferred. The Move prover is a formal verification tool for smart contracts written in the language.

Hyperledger Fabric Hyperledger Fabric emerged as one of the projects from the Linux Foundation's Hyperledger umbrella project. It aims to offer an "operating system for permissioned blockchains" [And+18]. Fabric networks are not open in the sense discussed in Section 1.2; they consist of agents who know each other's identity, and access to transactions is limited to this set of known agents at most. The degree of political decentralization that can be ascribed to a Fabric network is mostly determined by the applied consensus mechanism. Furthermore, there is no built-in cryptocurrency.

Fabric smart contracts are programs written in one of a number of languages, *e.g.*, Go, Java, or Javascript. Function calls are regulated by access control and submitted to a number of nodes. If these nodes compute the same results, the resulting state changes are submitted to an ordering service in the form of a read/write set (*i.e.*, a set of locations and values that are read or written, along with versions in order to detect read/write conflicts), and finally broadcast to all participating nodes.

Although not smart contracts in the strict sense of the above, Hyperledger Fabric contracts share some of the other characteristics: They are written in turing-complete languages, which allows the creation of arbitrary applications, and each smart contract defines its own state space within the network.

```
contract SimpleBank {

    mapping(address=>uint) balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint256 amount) public {
        require(balances[msg.sender] > amount);
        balances[msg.sender] -= amount;
        msg.sender.transfer(amount);
    }
}
```

Figure 1.3: Solidity simple bank application

1.4 Applications

We present some smart contract applications, ranging from simple examples to more complex real applications.

Bank One of the simplest examples often cited in introductions is a simple bank application, where a smart contract accepts transfers from participants, logs their balances, and allows them to withdraw their funds. It consists of one state variable `balances`, which maps accounts to unsigned integer balance values. Furthermore, it exposes two functions `deposit` and `withdraw`, which can be called to pay currency to the smart contract and get the funds back, respectively.

Figure 1.3 shows a Solidity version of the simple bank application. While it does not serve any practical purpose, it already must fulfill some fundamental correctness and security properties: The bookkeeping must be done correctly, and it must be guaranteed that only the account that deposited some funds has access to them.

Auction Another example (found, *e.g.*, in the documentation of the Solidity programming language [Doc]) is a smart contract implementing an auction. It is relatively simple to implement, but it has complex temporal correctness properties.

The implementation details vary. For demonstration purposes, consider an application as in Figure 1.4 consisting of a single contract which has six state variables: an owner account, a variable of type *UInt* storing the end time of

```
contract Auction {
    enum State {OPEN, CLOSED, FINALIZED};
    address payable owner;
    uint auctionEnd;
    mapping(address => uint) bids;
    uint highestBid = 0;
    address highestBidder = address(0);
    State state = State.OPEN;

    constructor(uint _duration) {
        auctionEnd = block.timestamp + _duration;
        owner = msg.sender;
    }

    function bid() public payable {
        require(state == State.OPEN, "Auction is already
           closed");
        require(msg.value + bids[msg.sender] > highestBid,
           "Bid must be higher than current highest bid");
        bids[msg.sender] += msg.value;
        highestBidder = msg.sender;
        highestBid = bids[msg.sender];
    }

    function withdraw() public {
        require(msg.sender != highestBidder, "The highest
           bidder cannot withdraw their bid");
        if(bids[msg.sender] != 0) {
            uint amt = bids[msg.sender];
            bids[msg.sender] = 0;
            payable(msg.sender).transfer(amt);
        }
    }

    function close() public {
        require(state == State.OPEN, "Auction is already
           closed");
        require(block.timestamp >= auctionEnd, "Auction
           cannot be closed yet");
        state = State.CLOSED;
        owner.transfer(highestBid);
    }

    function claim() public {
        require(state == State.CLOSED, "Auction must be
           closed");
        state = State.FINALIZED;
    }
}
```

Figure 1.4: Solidity auction application

the auction, a mapping `bids` which records the bids made by each actor resp. account, and `highestBidder` and `highestBid`, which store the current leader of the auction and their bid, respectively. Finally, an enum variable `state` records the state of the auction (open, closed, or finalized).

Furthermore, the contract has four functions: `bid()` transfers some amount of currency (specified by the caller) from the caller to the auction contract. If the amount is higher than the current leading bid, the `highestBidder` variable is overwritten with the caller's address, and their bid is recorded in `bids`. The function `close()` sets the `state` variable to `closed` and then assigns ownership of the auction item to the current highest bidder. The `withdraw()` function can be called by all losing bidders. It transfers the corresponding amount (as recorded in `bids`) to the caller. Finally, the `claim()` function can be called by the winner of the auction after the auction is closed, to transfer ownership of the auctioned item. The exact effects of this function are dependent on the concrete use case. Therefore, they are left unspecified in the example.

Escrow When smart contracts are used, *e.g.*, to buy and pay for online purchases, a variation of the so-called *oracle problem* can arise. If buyer and seller do not fully trust each other, then the purchase is difficult to carry out: Should the money be paid first, and the bought item shipped after the money has arrived? Or should the money only be paid after shipment? The payment could be automatically triggered by the successful delivery, but who gets to confirm the delivery?

One solution to this problem is an escrow contract as in Figure 1.5. Buyer and seller create a smart contract where the buyer first pays twice the required payment, but is refunded upon confirmation of delivery. This creates an incentive for both parties to act according to the protocol. Of course, depending on the goals of both participants, it is still possible for them to harm the other, *e.g.*, by not delivering the item at all. More involved versions of the escrow try to take this into account by also forcing the seller to deposit currency.

Casino Gambling has historically comprised a large part of smart contracts deployed on the Ethereum blockchain. One example has already been the subject of an academic effort at formal verification: The *VerifyThis* long-term challenge in 2021 was concerned with a Casino application [AWH21]. The Solidity contract given as an example contained two accounts, the operator and the player. A game works as follows: The operator can add or remove cryptocurrency from a pot. At some point, they create a game by choosing a secret number, hashing it and putting the hash value on the publicly visible blockchain. A player can then accept the game by guessing the parity of the original num-

```
contract Escrow {
  enum State { AWAIT_PAY, AWAIT_DELIVERY, COMPLETE }

  State public currState;
  address public buyer;
  address payable public seller;
  uint price;

  constructor(address _buyer, address payable _seller)
     public {
    buyer = _buyer;
    seller = _seller;
  }

  function deposit() external payable {
    require(msg.sender == buyer, "Only buyer!");
    require(currState == State.AWAIT_PAY, "Already paid");
    require(msg.value >= 2 * price, "Must pay deposit")
    currState = State.AWAIT_DELIVERY;
  }

  function confirmDelivery() external {
    require(msg.sender == buyer, "Only buyer!");
    require(currState == State.AWAIT_DELIVERY, "Cannot
       confirm delivery");
    seller.transfer(price);
    buyer.transfer(address(this).balance)
    currState = State.COMPLETE;
  }
}
```

Figure 1.5: A Solidity escrow smart contract, taken from [Zyn]

ber. Once the player has made their guess, the game is determined when the operator reveals their secret number.

The Casino has many interesting properties for verification. The main focus is on the fact that care must be taken to avoid an implementation in which the operator can block the execution of the application in case they lose, *i.e.*, by not revealing their original secret. This property can be described in various ways and various levels and makes the Casino an interesting case for evaluation of the temporal specification language developed in this work.

Tokens and Decentralized Exchanges One pervasive real-world use case for smart contracts is creating, buying and selling digital tokens. The block-

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (
    uint256 balance)
function transfer(address _to, uint256 _value) public
    returns (bool success)
function transferFrom(address _from, address _to, uint256
    _value) public returns (bool success)
function approve(address _spender, uint256 _value) public
    returns (bool success)
function allowance(address _owner, address _spender) public
    view returns (uint256 remaining)
```

Figure 1.6: The methods of the Solidity ERC 20 token standard, from [Fou24b]. This interface is implemented by creators of new cryptocurrencies on the Ethereum blockchain.

chain and smart contract hype led to the creation of a multitude of tokens, often associated to some underlying assets, or a proposed business model. Creating a custom token with a smart contract has become a standardized process. Two important standards are ERC-20 ([Fou24b], cf. Figure 1.6), which defines a fungible token, and ERC-721 [Fou23], which defines a non-fungible token.

Fungible tokens, called "coins", are often traded against each other on exchanges. While centralized exchanges are simpler to use for non-technical users, they also go against the industry's claim of decentralization. Therefore, decentralized exchanges have been developed. These consist of smart contracts where traders can buy and sell tokens according to pre-defined rules.

Augur Augur [Pet+18] is a prediction market platform built on the Ethereum blockchain. It allows users to create and participate in markets where they can predict the outcome of future events. The goal of the platform is to leverage the collective knowledge and insights of its users to forecast events across various domains, including finance, politics, sports, and more.

Participants in Augur can create markets by defining an event and its possible outcomes. Other users can then buy shares in these outcomes, essentially betting on what they believe will happen. The prices of shares fluctuate based on supply and demand, reflecting the collective probability assigned to each outcome by the market participants.

One particularly interesting aspect of smart contract-based prediction markets is that they have to solve the oracle problem, *i.e.*, the problem of how to cor-

rectly transfer real-world knowledge to the blockchain state. In the Ethereum setting, this problem has to be solved in the absence of a trusted party, and in the presence of strong financial incentives, *e.g.*, for incorrectly resolving a prediction market. To solve this, Augur employs a "decentralized oracle system" to verify the actual outcomes of events. This system consists of a network of nodes who stake their tokens, known as REP (Reputation), to report on event outcomes. Honest reporting is incentivized through rewards, while dishonest reporting can result in penalties.

Palinodia Palinodia [Ste+19; SDH20] is an application that allows users to assess the integrity of software binaries they download from the internet. It works by establishing unique identities for software and enforcing access control over these identities, including the publication and revocation of integrity protecting information for individual binaries.

Palinodia is comprised of three kinds of smart contracts as depicted in Figure 1.7: A *Software* contract establishes a root identity for a software product and is controlled by a Software Developer via an Identity Management contract. It can store references to several distinct Binary Hash Storage (BHS) contracts (representing different intermediary identities of the software), each managed by a different Maintainer. A *Binary Hash Storage* contract represents an intermediary identity of a software product and is managed by a Maintainer. Hashes of binaries (stored elsewhere) can be published, representing an endorsement by the Maintainer. They can also be revoked later. Each BHS contract is associated to one Software contract and one Identity Management contract. *Identity Management* contracts are used by Software Developer and Maintainer alike to control who has access to the functions of the software and BHS contracts respectively. In particular, individual Ethereum public keys can be added to and removed from Identity Management contracts to authorize or de-authorize them.

Users of the Palinodia application can interact with these smart contracts through a user client, which obtains contracts and their current state from the Ethereum network in order to verify both the integrity and endorsement of binaries they wish to use. To facilitate this, binaries include a metadata manifest with the address of their respective BHS contract. After obtaining the current state of the BHS contract, which includes the published hash of the binary for comparison, the user client proceeds to check whether the BHS contract is endorsed by the Software contract it expects. The first time a user obtains a binary, the address of its corresponding Software contract is stored as a trust anchor. Any binary that links back to such a stored Software contract through a BHS contract is trusted by the user client.

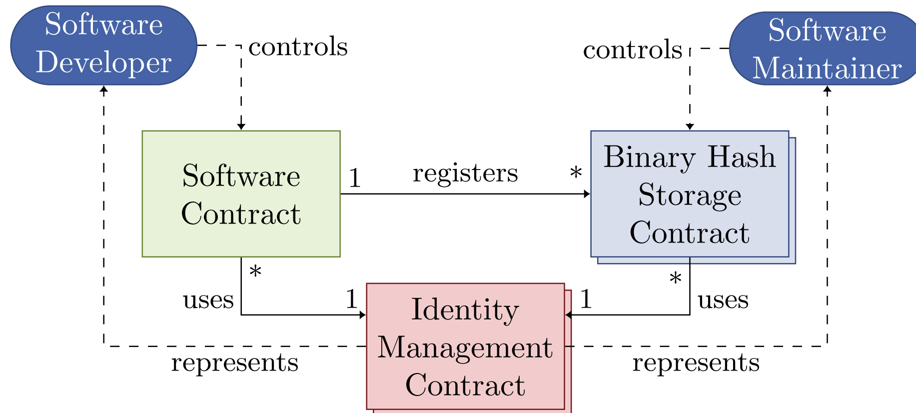


Figure 1.7: Overview of Palinodia, consisting of two roles (blue pills) and three kinds of smart contracts (colored rectangles) and their mutual relations, taken from [Ste+19]

Palinodia has been the subject of some academic research itself; a case study on the verification of access control requirements was conducted on it [Sch+21]. Furthermore, in a comparison of different methods of coupling smart contracts, Palinodia was the main subject [Fri+21].

In the context of this work, Palinodia is useful for evaluating the Scar approach. First, the structure and the requirements of Palinodia are described in detail in the paper. This makes it possible to model it in Scar and evaluate whether the Scar metamodel is suitable for the task (see Section 3.8). Furthermore, the generated source code can be compared to the existing source code to highlight advantages and shortcomings of Scar.

Secondly, since access control is an integral concern of the application, the question will be asked if Palinodia, as an application, can be specified in Scar in a simple and complete manner (see Section 4.5).

Decentralized Identity One more use case of Ethereum and other smart contract platforms has been as a provider of identity management. Proving one's identity, or selectively revealing personal information like age or academic credentials, is highly sensitive. This has lead to a number of approaches advocating for decentralized identity management building on blockchain and smart contract technology.

A very thorough approach is presented in DecentID [Fri+21]. In it, smart contracts on the Ethereum platform serve as a decentralized trust anchor, making use of the central guarantee of smart contract applications: that anyone can call their functions and be certain of the execution environment and the exe-

cuted code, despite the execution happening on somebody else's computer. In DecentID, users can create identities and authenticate themselves to different services. These services can only access attributes disclosed to them; all other attributes are kept secret.

The Ethereum DID Registry project [eth24b] has similar goals; their library has been deployed to the Ethereum blockchain and is still used occasionally [eth24a].

1.5 Hype and Criticism

This thesis is concerned with smart contracts as a technology, and with the correctness and security of concrete applications of this technology. As such, smart contracts are neither good nor bad. However, seeing the hype generated by the media as well as by people who promote cryptocurrencies as a financial investment, it is important to summarize the criticism as well.

Claims on Decentralization Decentralization claims in the blockchain and smart contract domain often should be taken with a grain of salt for two main reasons. Firstly, even if an application runs within a decentral system, this does not mean the application itself is decentralized. Access to functionality can, and often should, still be controlled (in fact, Chapter 4 of this thesis is entirely about how to do this correctly).

Secondly, blockchain platforms themselves are often not as decentralized as claimed. In the Bitcoin network, three to four mining pools make up more than half of the network, and have consistently done so for years [Gen+18]. In theory, they have considerable power over the development of the network, and over which transactions are executed. Furthermore, the implementation of the Bitcoin Core software which is running on many nodes in the network is curated by a small team of maintainers. While the software is open source and node operators can choose to reject changes, the core team holds a lot of power in practice.

Ethereum is not entirely different in this regard. Since the switch to a proof of stake consensus model, economics of scale put less pressure on miners to merge with others (although it has been argued that proof of stake also creates bad incentives [Wan+20]). As for the power of the core team, in 2016, a small number of people unilaterally decided to "roll back" the blockchain after the now-famous DAO hack (see Section 2.2). Although a central decision like this may be much harder to push through now that the Ethereum network has grown much larger, it is still a reminder that especially in unforeseeable, critical circumstances, decentralization might not be a strong as it is claimed to be.

There are other, less obvious sources of centralization: Sultanik et al. report that of all Bitcoin traffic, 60% traverses just three ISPs, and that in Ethereum, smart contract reuse may pose a high risk, since a high share of contracts is so similar to other, already deployed contracts that one can reasonably assume that large parts are just copied and pasted [Sul+22]. This introduces single points of failure that are hard to spot, despite the ostentatiously decentral network.

Technical Criticism Decentralization of a world-wide network has drawbacks in performance and scalability. One example is transaction throughput, which is limited to at most seven transactions per second [Cro+16] in the Bitcoin network and around 15 transactions per second on the Ethereum platform. This is several orders of magnitude less than payment processors like Visa, who process tens of thousands of transactions per second in peak times [SC21]. In times of high use, the Ethereum and Bitcoin networks can become congested, and participants have to pay high fees for their transactions to go through at all.

One proposed solution are so-called "Layer 2" networks, which aggregate transactions and commit them to the underlying blockchain in a batched fashion. However, these solutions invariably lead to a more centralized network. Furthermore, routing in such networks is believed to be an NP-hard problem [Di +18], so that their real-world applicability is still unclear.

One recurring point of criticism is the energy consumption of proof of work consensus; the Bitcoin network consumes more energy than a medium-sized country [OM14].

Usability of blockchains and smart contracts has also been criticized. In order to participate in the Bitcoin network, users have to deal with complex cryptographic systems. This presents a high entry barrier and can result in an irrevocable loss of funds. On the other hand, it incentivizes users to offload these tasks to centralized services like cryptocurrency exchanges, which negates the purported positive effects of decentralization.

Economic and Social Criticism Nassim Taleb criticizes the idea of cryptocurrencies as an investment, or as a hedge against inflation [Nic21]. Numerous other writers have written about the multiple scams and other nefarious practices, such as money laundering and fraud, that are regularly occurring in the blockchain and smart contract ecosystems [Whi24] [Ger24].

A recurring theme is what is dubbed "Solutionism", where blockchain technology is proposed without any actual need for either the blockchain data structure or the decentralized network, and where centralized alternatives would be preferable in all aspects.

Another recurring theme in the space is extreme concentration of wealth in the possession of very few actors. One particularly gregarious example is when new tokens are created with the sole purpose of creating a price on cryptocurrency exchanges and selling them before discontinuing the project, a surprisingly common practice which has been referred to as “rug-pulling”. However, uneven distribution of wealth is also echoed in Ethereum’s proof of stake consensus, which rewards those who already possess more than others.

1.6 Conclusion

Smart contracts are programs which give unique guarantees about the execution environment, and about what code is executed, even in decentralized platforms. While not as ubiquitous and pervasive as some commentators suggest, smart contracts can be used as building blocks for decentralized systems.

In this chapter, we surveyed the unique characteristics of smart contracts. We also pointed out that the guarantees smart contracts deliver come at the cost of some drawbacks. Most importantly, despite being tasked with the management of valuable resources, smart contracts often cannot be patched if bugs are discovered. This means that developers of smart contracts need to take precautions to ensure that smart contracts are correct upon deployment. In Chapter 2, we give an overview of the existing methods to achieve this.

Chapter 2

Formal Verification of Smart Contracts

Smart contracts have not only spurred hypes around cryptocurrencies and decentralized platforms, they have also inspired an impressive amount of research in the field of formal methods. In this chapter, we first give a short overview of the reasons for this development, and describe the errors and vulnerabilities that have had the highest impact. In the following, the existing approaches are reviewed and categorized. The chapter concludes with a more in-depth introduction to the works most related to the approach presented in this thesis.

2.1 The Need for Formal Methods

Beginning with the deployment of the Ethereum blockchain in 2015, formal analysis of smart contracts has quickly become a very active field of research. The number of tools especially for static analysis, but also for specifying and verifying more involved properties, is large enough that overviews of smart contract formal verification have become an established genre.

As already hinted at in Section 1.2, smart contracts require formal verification more than many other classes of programs. This is due to a number of reasons.

First, smart contracts typically manage assets, usually in the form of digital tokens representing either cryptocurrencies or other assets. The combined value of cryptocurrencies is in the billions of dollars (even if one concedes that the notion of "market capitalization" for cryptocurrencies is misleading, cf. [Whi22]). This creates an enormous incentive for attackers to find and exploit programming errors. Making this worse is the fact that in smart contracts,

correctness and security overlap much more than in other domains: Since the data smart contracts store is often literally an asset, or at least directly related to one, any minor error in the business logic of a smart contract has a high probability of also being a security vulnerability.

Furthermore, due to the public nature of smart contract platforms, the source code, or at least the executable byte code, of smart contract is available to everyone, including would-be attackers.

On top of this, smart contracts cannot easily be patched – at least not without sacrificing decentralization. If there is an authority who can unilaterally decide to update some functionality, then an application might be better off running on that authority's server instead of on a public blockchain. So, once an application is deployed, it is on its own, and any error found afterwards is both unlikely to be fixed and likely to be exploited.

All these characteristics lead to the conclusion that smart contracts should always be correct and secure upon deployment – in a domain where correctness does not just mean adhering to the specification, and security is not only the absence of a set of known vulnerabilities.

This has lead to a plethora of research and tool development in the field of smart contract security. Multiple papers have been written on the typical errors that smart contract developers make and the vulnerabilities arising from them, with a prominent focus on reentrancy. In response, a number of static analysis tools were developed to discover the most common and serious vulnerabilities. Somewhat later, vulnerabilities were categorized in the spirit of CVE repositories for classifying and reporting security issues. This categorization also enabled work on benchmarking static analysis tools.

At the same time, the first tools for formal specification and verification of user-defined correctness properties of smart contracts were proposed. These approaches usually target programs at the source code level and enable developers to describe the intended behavior of smart contracts, *e.g.*, in terms of invariants or function contracts. More recently, there have also been approaches to compare such formal verification tools, *e.g.*, using benchmark sets [BP17]. However, this is a harder task than doing the same for static analysis tools detecting known vulnerabilities , because the more involved formal verification tools differ more in terms of the specification language, and the properties they address.

While there exist some approaches targeting other platforms and programming languages, the vast majority especially of static analysis tools targets the Ethereum platform, either at the EVM bytecode level or on the Solidity source code level.

2.2 Common Errors and Vulnerabilities

Public awareness concerning the pitfalls of Ethereum, and possibly the scale of the damages caused by programming errors, was first raised by an attack on "The DAO" (Decentralized Autonomous Organization), a smart contract which was intended to act as an autonomous venture capital platform [Meh+19]. Participants were supposed to be able to fund projects through the platform, and reap financial rewards in case of successful projects. The smart contract attracted many investors; at the time of the attack, it contained a sizable share of all Ether in circulation.

An attacker managed to exploit errors in the protocol as well as a reentrancy vulnerability, stealing Ether valued at over 50 Million USD at the time. In reaction, the Ethereum blockchain was rolled back: A majority of nodes reset their local states to a point before the attack. Some disagreeing nodes went on to become "Ethereum Classic", a rival blockchain where the attack was never rolled back.

The attack highlighted the general attractiveness of smart contracts for attackers due to their financial nature, but also a concrete type of vulnerability, namely reentrancy. Reentrancy is a type of recursion between at least two functions; it occurs when a contract calls another contract's function containing a callback to the original caller. In Ethereum, sending money is a function call, and can result in a callback. Many programmers who adopted the then-new Solidity programming language failed to understand the possible consequences, and reentrancy vulnerabilities became the single most critical type of vulnerability on the Ethereum platform [Cha+23].

Other common types of vulnerabilities were pointed out, in an early paper about analyzing Ethereum smart contracts, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale" [Nik+18]. Later, the numerous types of security issues with Ethereum even gave rise to the Smart Contract Weakness Classification ([SWC23]), an overview site inspired by the CVE system [Mit24]. The site lists 36 different types of weaknesses. In recent years, it has been surpassed by other sites with similar goals, *e.g.*, the Smart Contract Security Verification Standard [SCS24] or OpenSCV [VIL23].

Some vulnerability types exist specifically in the domain of smart contracts, and are likely to escape programmers whose experiences are in other domains. These domain-specific vulnerabilities include **weak randomness**. Generating random numbers in a blockchain network is not trivial. Commonly used sources of randomness such as timestamps cannot be used in security-critical contexts. This is because the miner node who executes a transaction also determines its timestamp, and can therefore anticipate the random number and possibly discard undesired results. Similar problems can arise with all other

properties of the blockchain network, like the block hash or the block number.

Another specific vulnerability is **transaction-ordering dependence**: In some cases, the correctness of a smart contract may depend on the order in which transactions happen. The power over this ordering rests with the miner nodes, and therefore cannot be relied upon for correctness. A related problem is the possibility of *frontrunning*: Since transactions are submitted to a publicly visible pool before being executed, it is possible for network participants to analyze pending transactions and taking advantage of them. This has led to the phenomenon of *miner extractable value*, which describes the financial gain which a miner node can expect by frontrunning a given transaction. This is especially relevant on decentralized cryptocurrency exchanges.

Another domain-specific problem are **failed calls**. In Ethereum, currency transfers are function calls, and as such give control over program flow to the recipient. This can enable reentrancy attacks, but also gives the recipient the possibility to fail receiving the transfer on purpose. This can be especially harmful when several transfers are carried out at once, *e.g.*, in a loop which transfers currency to a set of recipients. If failed calls are not handled correctly, a single failed call reverts the transaction. This has led to the practice of letting others withdraw funds, instead of actively transferring them.

Another important class of vulnerability can be summarized as **Improper Access Control**. While access control is a very general area of information security, it also plays a very important role in smart contract applications; two entries in the SWC registry fall in this category, highlighting unprotected Ether withdrawal, and unprotected *selfdestruct* instructions, which remove a contract from the Ethereum world state.

While all these issues are important and can go wrong in many ways, it is important to note that in the smart contract domain, security is highly application specific. As [Zha+23] notes, security issues in software traditionally come from vulnerabilities such as buffer overflows, privilege escalation, information leaks. This is different from functional bugs, which cause unexpected behavior, but not necessarily a security issue. Analyzing security can be done abstractly, in the same manner, independent of the application. For smart contracts, however, security and functional correctness overlap much more, which means that security is a much more application-dependent question. This makes security much less automatically checkable.

Therefore, while also introducing some static analysis approaches in the following, the main focus of this chapter is on tools and techniques where users can define custom, application-specific properties.

2.3 Classification of Verification Approaches

There are many dimensions in which smart contract analysis and verification tools can be categorized. We explore these dimensions to put the work presented in later chapters into context.

Overviews

Due to the large number of works in the field of smart contract security, several overviews have been published, with the goals of categorizing, analyzing, and benchmarking different analysis and verification tools.

Munir and Taha aim to give a comprehensive overview of verification methodologies, tools, and properties of interest in the domain of smart contracts in their paper *Pre-Deployment Analysis of Smart Contracts – A Survey* [MT23]. They also include a list of survey and overview papers.

Chaliasos et al. give a security-focused overview of smart contract analysis tools in *Smart Contract and DeFi Security*[Cha+23]. Their main finding is that static analysis tools are not, in practice, sufficient to detect security problems. They attribute this to the fact that security in smart contracts is very application-specific, and therefore needs user-defined specification.

Durieux et al. compare nine static analysis tools on one data set which contains known vulnerabilities, and on a large benchmark set consisting of all deployed smart contracts for which the Solidity source code is available [Dur+20]. They find that more than half of all vulnerabilities are not found by any of the tools. The Mythril tools find most, but still fails to find 76% of the known issues.

Deductive Verification and Model Checking

The term "formal verification" denotes rigorous processes to ensure that a system behaves according to its specifications. Such processes rely on methods based on mathematical logic.

The two main approaches to formal verification are *model checking* and *deductive verification*. Deductive verification encompasses all methods which generate *proof obligations* in some formal logic from a system and its specification. A proof of these obligations corresponds to a proof of correctness of the system against its specification. Proof can either be carried out in an interactive theorem prover (such as Coq or Isabelle/HOL, see Section 2.4), or automatically, after a translation to automated tools such as Satisfiability Modulo Theory (SMT) provers (cf. Section 2.4).

Model checking is the systematic exploration of the possible state space of a system. For finite systems, this exploration can be made exhaustively, al-

though state space explosion can make this approach computationally difficult or infeasible. Infinite state systems can be explored exhaustively only if suitable abstractions can be found to reduce the state space. The properties to be verified are usually defined in temporal logics, such as Linear-time Temporal Logic (LTL) or Computation Tree Logic (CTL).

In the SCAR approach that forms the central contribution of this work, automated deductive verification of smart contract source code serves as the justification for an abstract model of the application, which can then be analyzed by model checking. The higher-level properties proven on the model, such as temporal and security properties, are also fulfilled by the source code.

Verification on Different Layers

Like all modern software, smart contracts form a technological stack that is several layers deep. Tools and methods have been devised to analyze correctness properties on all layers. The focus of this work is verification of properties at the source code level as well as on abstract models which abstract from the source. However, relevant properties also exist at other levels – platform, or virtual machine – verification approaches exist for each of these levels.

As for the platform level, smart contract platforms are defined by complex network and communication protocols, and by the execution environment which the applications run in. Examples include the proof of stake consensus algorithm of Ethereum, and the Ethereum Virtual Machine. An example of formal verification on this level is the Verification of Ethereum's "Beacon" blockchain by Cassez, Fuller, and Asgaonkar [CFA22].

For Ethereum specifically, smart contracts are always publicly available as EVM bytecode. Therefore, static analysis tools may choose this level of abstraction to work on. Compared to source-level verification, this eliminates the abstraction introduced by compilation. However, Ethereum bytecode contains very little static information, making analysis difficult [Sch+20].

The deployed application is considered in runtime verification. The system is monitored while it runs, and it is observed whether the current execution violates or fulfills a pre-defined set of properties. Typically, a *monitor* is generated from a formula describing a desired property of the system (*e.g.*, a temporal logic formula). The system is then instrumented to notify the monitor of its behavior. The monitor then judges whether the current behavior fulfills the property.

Runtime verification is less complex than static formal verification methods like model checking or theorem proving because it only observes a finite set of execution traces. Conversely, it is also less powerful (*e.g.*, relational properties, which range over more than one execution of a program, are non-monitorable).

In the context of smart contracts, runtime verification can be used to check whether the system is in an undesirable state, and take action accordingly (see ContractLarva [AEP18]).

Specification Type

Existing formal methods for smart contracts can be subdivided by how the analyzed properties are specified – either there is a pre-defined set of properties, or properties can be specified by the user.

Static analysis can be conducted to check whether a program contains a given **syntactic pattern**. This kind of analysis is fast and lightweight, but also imprecise.

As an example in Solidity, a useful check might be that every function which changes the state of the contract, *i.e.*, contains an assignment to state variables, also contains a statement of the form `require(msg.sender == ...)`, which controls access to the function. This check is implemented, *e.g.*, in the Securify tool [Tsa+18]. Some forms of reentrancy can also be detected syntactically, *e.g.*, by checking whether a contract changes its state after an external call.

More relevant patterns can be expressed on the level of an application's **control-flow graph** (CFG). The nodes of the CFG represent basic blocks, *i.e.*, parts of the code without jumps or branches. An edge between two basic block represents a jump in the control flow.

Many vulnerabilities in smart contract applications can be encoded as patterns on an application's CFG. This is especially relevant in Ethereum, where external function calls yield control from the calling contract to the callee. Examples include reentrancy and reachability of critical functionality, *e.g.*, a contract's "suicide". More patterns are explored, *e.g.*, in the 2018 paper "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale" [Nik+18].

However, many approaches have been developed where developers can specify application-specific correctness properties. This can happen in the form of assumptions and assertions in the code (cf. the approaches in Section 2.4). Other approaches allow specification on some formal model of an application (cf. Section 2.4). Still others have developed their own source-level specification languages, usually based on first order logic or temporal logics (cf. Section 2.4). Bartoletti et al. make an attempt to define benchmarks for this category of formal verification tools [Bar+24].

Methods and Techniques

Formal methods build on various techniques. Static analysis tools which work on the control flow graph commonly rely on **reachability analysis**, *i.e.*, the

question of whether a given state in the program flow is possible.

Other approaches rely on **symbolic execution**, where a program is executed with symbolic values as an input. From this, a set of traces is obtained, each of which has a path condition. Then, traces with unsatisfiable path conditions can be pruned, and the remaining traces can be checked for conformance with, or violation of, some property. Symbolic execution is generally limited by the bounds on the search (which may lead to false negatives) and by the way the desired properties are specified, which may lead to false positives.

2.4 Tools

This section presents notable tools in the area of smart contract verification and analysis, loosely ordered by the categories described above. It also serves as an overview of the previous work most closely related to the SCAR project. Fo example, we introduce approaches which focus on the resource character of smart contracts, and is therefore closely related to the capability-based security approach presented in Chapter 4 and the concept of ownership discussed in Chapter 5. The approaches listed in the final parts of this section are related to the overall SCAR approach presented in Chapter 3.

Static Vulnerability Analysis

The tools for smart contract analysis collected here take an approach where a number of vulnerabilities is collected, and the tool subsequently tries to establish that a smart contract does not contain any of these vulnerabilities. The approach often involves symbolic execution to explore all possible traces from a given entry point. Vulnerabilities are characterized as patterns on these traces.

These approaches generally produce false positives as well as false negatives. False positives occur when a pattern is not specific enough (*e.g.*, an execution trace is compliant with a pattern characterizing reentrancy, but does not actually allow reentrant behavior). False negatives may be produced on two different levels: Either the verification is bounded and does not find a possible violation, or the pattern (on traces) which is supposed to define a property is not precise.

The earliest academic tool for static analysis of Ethereum smart contracts was OYENTE [Luu+16]. It transforms Ethereum bytecode into a control flow graph, on which it then conducts symbolic execution. The resulting traces are checked for transaction-ordering dependence, timestamp dependence, mishandled exceptions, and reentrancy. While the tool is no longer actively maintained, it is still widely used in benchmarking comparisons. The MAIAN tool

[Nik+18] is an extension of the Oyente approach which also considers attacks that require multiple transactions.

SLITHER [FGG19] is a static analysis tool for Ethereum smart contracts. It takes as input the abstract syntax tree generated by the Solidity compiler from a set of Solidity source code files. The input is first converted to a control flow graph with additional information about the contract, such as inheritance and a list of expressions. This is then further converted to SlithIR, an intermediate language using Single Static Assignment form. On this representation, different analyses can be conducted, including the detection of predefined vulnerabilities, but also opportunities for optimization. Furthermore, SLITHER enables the configuration of custom "printers" to output information about the program, such as information flow, in human readable form. This can also serve as the basis for custom analyses.

The ECHIDNA tool [Gri+20] is an approach for fuzzing Solidity smart contracts, *i.e.*, simulating the behavior of a system with randomized input. Echidna analyzes the behavior for possible violations of pre-defined security vulnerabilities, but also of user-defined correctness properties.

MYTHRIL [Mue18] is an open static analysis tool for Ethereum byte code. Analysis is based on symbolic execution. The tool continues to be used and has been positively evaluated in benchmarks in comparison to other static analyzers [Dur+20].

ETHOR [Sch+20] is a tool for finding vulnerabilities in Solidity smart contracts by conducting reachability analysis. Unlike other static analysis tools, ETHOR claims soundness, *i.e.*, the absence of false negatives. While the authors of previous tools have already claimed to achieve soundness in their evaluation (cf. [Kal+18]), Schneidewind et al. give a formal proof that their methodology is sound against the formal semantics of the EVM as defined by [Gri+20].

Static analysis has also been developed for Solana smart contracts written in Rust [Tav22] and [Cui+22].

SMARTBUGS [Fer+20] is a tool which combines several existing static analysis tools in one executable container, making it very easy to execute those.

Comparisons Chaliasos et al. analyze the performance of automated vulnerability checking tools at the hand of 127 high-impact real-world attacks [Cha+23]. Their most striking result is that the vast majority of the attacks they analyze could not have been avoided using static analysis tools, since these tools are unable to detect issues that stem from errors implementing the business logic. All of the attacks that are detected by static analyzers are based on reentrancy vulnerabilities. Of the analyzed tools, SLITHER performs best at detecting reentrancy, but also yields a high rate of false positive results.

Assertion Checking

All common smart contract languages have built-in concepts of assumptions and assertions, which check conditions specified in the programming language at runtime and throw an error and revert in case the condition is not met. For example, Solidity has the `require` keyword to encode assumptions, *e.g.*, about the caller of a function or the function arguments, and the `assert` keyword to encode knowledge about necessary conditions at some point during execution.

For the purposes of formal methods, assumptions and assertions can be used as a specification mechanism. This can be beneficial, since it means that the specification language is the same as the programming language, and developers do not need to be trained in a new language. However, this also means that assertions lack the abstractions and quality-of-life features that specification languages can provide, *e.g.*, quantification over unbounded data types. A number of tools have been developed which can statically check source code assertions for possible violations:

SolCMC [Mar+20] is an approach for verifying that assertions in the Solidity source code cannot be violated. It has since been integrated into the Solidity compiler, where it can be activated with a command-line flag [Oto+23]. Verification is based on a translation to constrained horn clauses.

The VeriSol tool [Wan+19] is a static assertion checker for Solidity built on a translation to the Boogie intermediate language [Bar+06]. Apart from verifying the absence of assertion violations, the tool also provides a mechanism to specify conformance to access control policies. Similarly, the Solidifier [AR21] tool also translates Solidity code with assumptions and assertions to Boogie. The resulting proof obligations are discharged with the Corral model checker [LQL12]. Both of the above approaches conduct only bounded verification.

Model Checking

Model checking is a broad category, and a large, diverse set of tools use this formal verification approach. There are differences in the representation of the model, and in the verification tool set.

Nehaï, Piriou, and Daumas model Ethereum smart contracts [NPD18] in the language of the nuXmv model checker [Cav+14]. The approach also provides a model of the blockchain environment. However, the provided data types are limited to integers and booleans.

Lahbib et al. explore modeling smart contracts in Event-B [Lah+20] and proving invariants and safety properties using the tools provided by the framework [Abr10]. As per the paper, the translation is still manual.

Yet another existing formalism is built upon by Garfatta et al. [Gar+22]: Smart contracts are translated to colored petri nets. Safety properties are then specified and model-checked in the petri net formalism. The work of Pinna and Tonelli [PT22] goes in the other direction: smart contracts are first modeled as petri nets, and safety properties are verified on them. Then, Solidity smart contracts are generated from the petri nets.

SmartAce [Wes+22] extends the Solidity compiler to support bounded and parameterized smart contract verification. It is based on the SEAHorn formal verification framework [Gur+15].

Outside of Ethereum, there is also an approach for bounded model checking of Solana smart contracts written in Rust [Ott24].

Automatic Deductive Verification

The approaches in this section all feature specification languages which are more expressive than the programming language that they target. All of them feature universal and existential quantification. This requires the use of more powerful verification techniques.

solc-verify solc-verify [HJ20] is a formal verification tool for programs written in Solidity. It takes as input Solidity smart contracts which are annotated in solc-verify's formal specification language. Annotations can be contract invariants, which specify conditions that always have to be maintained after any public function call, and function contracts. Function contracts consist of pre- and postconditions, conditions about events, and frame conditions, which specify which part of the state a function may modify.

Specification expressions are in Solidity with some first-order logic constructs, including quantification over array and mapping elements as well as a bounded sum operator. Furthermore, in the postcondition, the state before the function call can be accessed. Function calls can be used in the specification if the concerned function is marked as internal (in which case it is inlined for verification), or if it has a function contract so that it can be abstracted.

For verification, options can be specified, *e.g.*, for the treatment of arithmetic operations and integer data types. solc-verify translates its input to Boogie and then discharges the resulting proof obligations either with the Z3 [dB08] or the CVC4 [Bar+11] SMT solver.

solc-verify has been used in other works. Antonino et al. developed an approach for safe upgrading of deployed smart contracts, where before an upgrade, it has to be proven that the new version of the contract still fulfills some required correctness property [Ant+22; Ant+24].

Celestial CELESTIAL [Dha+21] is a functional verification tool for Solidity smart contracts. It extends Solidity with annotations for invariants, postconditions, conditions for which a transaction will revert, and function frame conditions clauses.

Annotated CELESTIAL contracts can then be translated to F*, an ML-like language which includes a tool chain for formal verification. Verification is based on a translation of annotated programs as an SMT proof obligation.

CELESTIAL also includes an environment model of the Ethereum network. The annotated contract is verified in the context of this model by SMT solvers.

Dafny DAFNY [Lei10] is an approach for functional verification consisting of the Dafny programming language and a verifier. The Dafny language contains constructs for function contracts and loop invariants. The verifier builds on BOOGIE and Z3.

DAFNY has been used as a platform to write verified smart contracts which can then be translated to Solidity and deployed [CFQ22]. Furthermore, the Ethereum Virtual Machine has been specified in DAFNY in terms of executable semantics [Cas+23].

VerX VERX [Per+20] is a tool for verification of safety properties of Solidity smart contracts. Its specification language includes past temporal operators (`always`, `once`, `previously`. It is also possible to refer to a function call and its arguments as a predicate in the specification.

An application is viewed as an implicit loop over all function calls, where the function, the arguments and the caller are chosen non-deterministically. Verification of the specified properties is conducted by symbolic execution of this implicit loop, and verifying that the constraints collected in the course of the symbolic execution satisfy the specified properties.

Theorem Proving

Isabelle Modeling Hirai presented a semantic definition of the EVM in Isabelle/HOL [NPW02] in 2017 [Hir17].

Building on this definition, Amani et al. create an approach for formal verification of smart contracts in EVM bytecode [Ama+18]. While the approach allows for very expressive specification, verification in Isabelle is mostly interactive, which makes the approach less suitable for non-experts in theorem proving.

Marmsoler and Brucker present an embedding of Solidity into Isabelle/HOL, which enables specification and verification of source-level properties [MB22].

KEVM The K framework [RŞ10] allows the definition of programming languages, calculi, and type systems. From the definition, a number of programming language tools, such as compilers, but also formal analysis tools, *e.g.*, symbolic execution tools, can be automatically generated.

There have been works which use the K framework in the domain of smart contracts. While a definition of Solidity within the K framework has not seen much use [Jia+20], the definition of the EVM semantics in the K framework [Hil+18] has been endorsed by the EVM developer team and built upon by other formal approaches (*e.g.*, [Sch+20]).

Functional Smart Contracts in Coq In the context of the ConCert framework [ANS20], Annenkov et al. have created an embedding of a functional smart contract programming language in Coq [Ann+21]. This makes it possible to test and verify smart contracts with the Coq theorem prover, and extract smart contracts which exhibit the desired correctness properties.

Languages

Scribble So far, there have been relatively few attempts to create specification languages for smart contracts, unless in the context of specific tools. One notable exception is Scribble, which has been used as a specification language for different approaches ranging from runtime verification to assertion checking [Mis+24].

Scribble [SCr23] is an annotation language for Solidity smart contracts. The main goal is to provide tooling for runtime verification. Annotations (in Solidity) can be written for functions ("`if_succeeds`") to encode pre- and postconditions. State variables can also be annotated to require that some property holds on an update of the variable. This can be used, *e.g.*, to express a concept of ownership. Furthermore, contract invariants can be specified. Clauses encoding frame conditions, such as in Celestial or solc-verify, are not possible.

Scilla Scilla [Ser+19] is an intermediate-level functional smart contract programming language which aims to avoid common vulnerabilities observed in Solidity smart contracts, namely reentrancy. Scilla is not turing-complete, but the authors argue that all common use cases can still be implemented, and that trading off some expressiveness is worth the gain in security and ease of verification. In the paper, the authors sketch how verification of Scilla smart contracts can be conducted through an embedding in Coq [Pau11].

Obsidian OBSIDIAN [Cob17; Cob+20] is a platform-independent programming language for smart contracts which aims to avoid common errors made in Solidity and other languages. It is a typed language based on linear types. Linear types ensure that objects can be used only once, and are therefore a good fit for modeling non-fungible resources in smart contract applications. OBSIDIAN also includes a concept of ownership.

The workflow with OBSIDIAN is to write an application in the language and rely on the built-in type checker to verify that no unintended behavior can occur. Afterwards, the application can be translated to a specific platform. As of 2020, only Hyperledger Fabric was supported, and the documentation [Obs] does not contain any updates afterwards.

Resource-based Specification and Verification Approaches

One characteristic of smart contract applications is that they usually manage resources of some kind, *e.g.*, cryptocurrencies or tokens representing some real-world asset. Some of these resources are built into the platform, like the Ether cryptocurrency. Others are created within smart contracts, using the data types the platforms provide.

Some formal verification approaches take the resource properties of such data types into account, and provide constructs for reasoning about ownership or resource transfers.

Move Language and Prover MOVE is a smart contract programming language developed for Facebook's Diem blockchain. Although Diem has been discontinued, the MOVE language is still being developed [Mov24], particularly because of its strong emphasis on formal verification. MOVE's type system and resource model are designed to prevent common vulnerabilities, such as reentrancy and unauthorized asset duplication, by ensuring that assets are treated as linear resources that cannot be accidentally duplicated or destroyed. The language is also modular, allowing for the creation of reusable and composable components, which can enhance security and efficiency in smart contract development.

The MOVE Prover is a formal verification tool integrated with the MOVE language to ensure the correctness of smart contracts. It checks for safety properties such as invariants, preconditions, and postconditions. By automatically generating and checking proofs, the MOVE Prover helps developers identify and mitigate potential security issues early in the development process.

2Vyper 2Vyper [Brä+21] is a verification tool for Ethereum smart contracts written in the Vyper language. The specification language has resources as first class specification elements, and these automatically have all the properties expected of them, *i.e.*, they cannot be copied, duplicated, or deleted. Furthermore, the specification language includes a notion of ownership, where it can be specified that some resource can only be moved or changed by a certain account.

Resource Specifications for Rust-based Smart Contracts Grannan and Summers observe that classic specification languages with first-order expressions over program state often lead to cumbersome specification when it comes to programs that deal with resources that cannot be copied or duplicated [GS23]. While, *e.g.*, a developer may take it for granted that money in a bank cannot simply be deleted or copied, properties like this have to be specified using frame conditions, creating a mismatch between the mental model of the developer and the specification.

Based on these observations, a resource-based specification language for smart contracts written in Rust is developed. Verification is implemented as an extension of the Prusti verifier [Ast+22] (which is, in turn, an extension of the separation logic-based Viper approach [MSS16]).

Formal Model

This section presents verification approaches which target application-level properties and work on some abstract model. Therefore, the approaches mentioned here are the closest to the Scar approach introduced in Chapter 3.

Quartz Quartz [Kol+20] is an approach in which smart contracts can be concisely specified, statically analyzed on the model level, and then turned into Solidity implementations which fulfill the analyzed properties.

The specification language is a mix of the TLA+ systems specification language [Lam02] and Solidity. In TLA+, systems are specified in terms of state variables and *actions*, which define state transitions. In Quartz, actions can be written in a limited subset of Solidity, which does not contain loops and branching. However, Quartz offers some constructs to reason about authorization and time, *e.g.*, it is possible to define actions which are only possible from a point in time onward, or in a given interval. With the actions translated to TLA+, an application can then be model-checked with the TLC model checker [YML99].

SmartPulse SmartPulse [Ste+21] is a tool for checking safety and also, especially, liveness properties of Solidity smart contracts. Properties are specified in SmartLTL, which contains primitives for functions being called, functions finishing execution, reverting, and sending ether. Fairness specifications can be specified if necessary to prove liveness properties.

In addition to source code and temporal specification, an environment can be specified, consisting of an attacker model (the set of functions the attacker can call, and a bound on the number of calls) and a blockchain model (gas costs, especially of the `transfer` function). Useful defaults are provided for the environment models.

SmartPulse builds on VeriSol's translation to Boogie. For the verification of temporal properties, specification and programs are translated to Büchi automata and verified by the Ultimate Automizer tool [Hei+13].

FSolidM and VeriSolid The FSolidM tool [ML18] by Mavridou and Laszka and its successor VeriSolid [Mav+19; Nel+20] have been developed to enable *correct-by-design* Solidity smart contracts. In both approaches, developers model smart contracts as transition systems, consisting of pre-defined states and actions. Action define the transitions between states. They can contain Solidity source code.

The authors define structural operational semantics for transitions written in a supported Solidity subset. This allows verification of some properties, such as the existence of an initial state, or function termination.

While FSolidM is for single smart contracts, VeriSolid extends it by a mechanism called *deployment diagrams*, which allows specifying an application consisting of several contracts and their interactions. This specification can then be translated to a model in the Behavior, Interaction, Priority language [Bli+15]. Then, existing tools can be used to check this model for safety and liveness properties. Since the transitions in the model are defined in Solidity by the developer, generating an implementation from the model is straightforward. One caveat of the overall approach is that the developer has to define an explicit finite set of states that an application can be in.

Model-driven Development

In Chapter 3, the Scar approach for model-driven development based on formal verification is introduced. While model-driven development is not a formal method as such, this section lists some approaches which enable development of smart contract applications in this way.

Skotnica and Pergl [SP20] develop DasContract, an UML-like domain-specific language for modeling smart contracts. Their metamodel includes el-

ements for contracts, data types, transactions, and actor. The approach allows the creation of a number of domain-specific models and diagrams, including and action models which can specify conditions on transactions, as well as structure diagrams and process diagrams, which are similar to UML class diagrams and sequence diagrams, respectively.

The iContractML language [HMQ20] focuses more on the deployment of Solidity smart contracts. A metamodel of smart contracts is developed, consisting of the *contract*, *transaction*, *asset*, and *participant* elements. From the model, code can be generated for the Ethereum and Hyperledger Fabric platforms.

Other approaches in this area include architectural modeling of smart contracts [Jur+23] and model-driven development of smart contracts for cyber-physical systems [Gar+18].

Chapter 3

The SCAR Approach

Chapter 2 gives an overview of the multitude of existing approaches which apply formal methods to smart contracts. Despite the availability of these tools and methods, security remains a challenging issue in the domain of smart contracts.

In this chapter, we show that to achieve more secure smart contract applications, an approach is needed which provides a more suitable level of abstraction for describing security and correctness properties. At the same time, such an approach must still result in a concrete implementation which satisfies the specified properties.

For this, we propose the SCAR ("Smart Contract Abstract Representation") approach. At its core, the approach consists of a *metamodel* of smart contract applications. Developers can create instances of this metamodel to describe an application in terms of its state and the functionality it exposes.

Furthermore, the SCAR approach provides a process to get from a model to an implementation that is consistent with the model. This process is based on the automatic generation of formal specification, and the use of existing tools for formal verification.

The model serves as ground truth for application-level specification and analysis of security properties like access control policies and temporal properties. While some analyses have been developed and implemented, the SCAR approach is also intended as an extendable platform, where new analyses can be easily added, and other tools can be integrated.

In the following, we give some evidence for the necessity of this verification-based, model-driven approach in the domain of smart contracts. Then, we present an overview of the goals, the core components, and the processes of the SCAR approach in Section 3.2. In Section 3.3, the SCAR application metamodel is described in terms of syntax and semantics. Afterwards, we describe how

consistency between a SCAR model of an application and a source code implementation of the same application can be achieved. The SCAR approach has been implemented in a publicly available project. Next, we discuss examples of SCAR models of the use cases described in Section 1.4. Finally, in Section 3.7, we sketch some further possible use cases of the SCAR approach that go beyond the process described in Section 3.2.

3.1 Model-driven, Verification-based Development

As detailed in Chapter 2, static analysis and formal verification of smart contracts have been very active fields of research. Numerous approaches for detecting bugs and common weaknesses as well as specifying and verifying user-defined correctness properties have been proposed; Munir and Taha list 194 works in their paper on *Pre-Deployment Analysis of Smart Contracts – A Survey* [MT23].

However, smart contract security is still a problem. There are entire websites dedicated to listing all the instances of smart contract applications being hacked (for example, [Whi24] and [Ger24]). Many of the exploited applications were even audited, prior to the attacks, by companies like CertiK [Cer24], who perform audits consisting of both manual code review and automated static analysis. This indicates that the methods used in the audits are not sufficient to guarantee security.

Although static analysis tools may help to avoid common vulnerabilities, current research shows that they are not sufficient to ensure smart contract application security. For example, Zhang et al. come to the conclusion that the majority of bugs in smart contracts is “hard to find” and “not machine-auditable” [Zha+23]. Chaliasos et al. state, after analyzing a set of high-impact attacks, that only 8% of the vulnerabilities which led to these attacks could have been detected with *any* state-of-the-art static analysis tool [Cha+23]. Moreover, He et al. argue that most current vulnerability detection tools “can only detect vulnerabilities in a single and old version of smart contracts” [He+23]. This also hints at the wide-spread phenomenon of software quickly becoming unusable in the wake of advancements; for example, new versions of the Solidity programming language or the Ethereum virtual machine can render analysis tools obsolete, unless they are actively developed. Ivanov et al. come to a similar conclusion in their survey of security defense mechanisms, concluding that “[...] many vulnerabilities are not covered by any solutions” [Iva+23]. They also point out another way in which static analysis tools can be insufficient,

namely that "Apart from strict false positives, static analysis tools also report vulnerabilities that are not exploitable (*e.g.*, for economic reasons)".

In the presentation of the HORSTIFY tool, Holler, Biewer, and Schneidewind also touch upon the problem of false positives and, especially, false negatives, claiming that "most analysis tools aiming at provable soundness guarantees fall short" [HBS23]. On this, the authors base their argument that better static analysis tools for vulnerability detection are needed, especially tools that give verifiable soundness guarantees.

Overall, however, we can conclude that static analysis tools alone do not and cannot suffice to ensure security in smart contract applications. In this domain, security is just too application-specific.

Formal verification tools go a different route by allowing their users to define the desired properties, and attempting a proof. Unlike with vulnerability detection tools, there is not always a yes or no answer; a proof attempt can always time out. If a proof is found, the program is deemed to fulfill its specification. On one hand, this greatly increases the complexity of formal analysis, since the developer has to write formal specification. On the other hand, a positive result, *i.e.*, a proof of correctness of a desired property, is a much more powerful result than the absence of some predefined vulnerabilities.

Different approaches allow specifying different kinds of properties (cf. Section 2.4). The two most common kinds are function contracts and invariants, *e.g.*, SOLC-VERIFY [HJ20] or CELESTIAL [Dha+21]. Other tools go beyond this and also allow specifying temporal properties, like VERX [Per+20].

One difficulty that developers face when using source code-level formal verification tools of this kind is a lack of abstraction. The specification is on the source code level. Therefore, a developer will always have to think about the application in terms of a specific platform and programming language, even when specifying abstract behavior. This is especially difficult with relatively new programming languages, like Solidity, where developers often lack familiarity with the peculiarities, and where best practices have not yet been established.

Furthermore, important correctness properties in the smart contract domain are often not properties of single functions, but of the entire application. This makes them inherently hard to specify on the source code level. A good example of this is access control: A developer may want to specify that a certain functionality must only be available to a subset of users, or that a part of the application state may only be modified by the owner of the application. While access to a single function can be expressed as a precondition to that function, there is no immediate way to specify a restriction of access to state.

A more general perspective on this is offered by Grannan and Summers: Developers might also face difficulties in specifying the behavior of smart contract applications due to the huge semantic gap between intuitive concepts like

resource ownership and access control on one hand, and their representation in a programming language on the other hand (cf. [GS23]). It is easy to state that an account "owns" the amount stored in a variable, or that only a subset of callers may access a state variable, and the intent behind these statements is clear. However, the formal specification needed to describe these intentions in terms of function contracts and invariants, is much more complex. This indicates a need of an additional layer of abstraction, which allows developers to use "easy" concepts in a way that still builds on the rigorous guarantees of formal methods.

The above observations lead to the Scar model-driven development approach for smart contract applications that presented in this work.

3.2 Scar Overview

In this chapter, we present Scar, a modeling and analysis approach for smart contract applications. The Scar approach consists of three main parts:

- A **metamodel of smart contract applications**, in which these applications can be defined in terms of their structure, *i.e.*, the contracts, state, and functions they consist of. A type system is also part of the metamodel.
- The **ScarML specification language**, which serves to describe the behavior of individual functions in terms of pre- and postconditions, and properties of individual contracts in terms of invariants.
- A **model-driven process** based on formal verification, which leads from the creation of a Scar model to a source code implementation that can be deployed on a specific platform. Formal methods ensure that the implementation is consistent with the model.

In the following, we use the term *basic metamodel* to refer to the structural part of the metamodel. This part consists of contracts, *i.e.*, objects which encapsulate their own state (in the form of state variables) and functionality (in the form of functions). The basic metamodel does not contain ScarML, the behavioral specification part of the metamodel. An instance of the basic metamodel is a basic model.

The ScarML specification can be attached to functions or individual contracts. In contrast, we use the term *application-level property* to describe properties that cannot be readily expressed in ScarML, because they to not pertain to just one function or contract. In Chapter 4 and Chapter 5, we extend the Scar metamodel with specification languages for such application-level properties, and develop specific model-level analysis techniques.

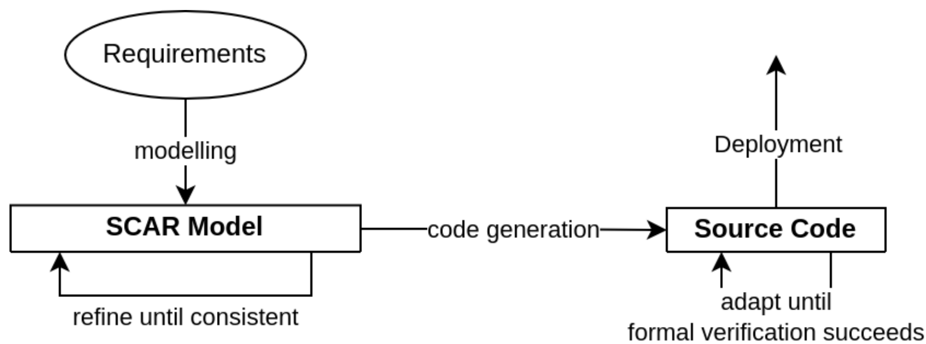


Figure 3.1: The SCAR process of smart contract application development

Model-driven Development Process Overview

An overview of the intended workflow with the SCAR platform is given in Figure 3.1. After collecting the requirements for an application, developers create a basic SCAR model. This basic model defines an application in terms of its smart contracts, their state variables and functions, and the behavior of these functions. The developer then proceeds in two directions: First, they specify application-level correctness and security properties (cf. Chapter 4 and Chapter 5) on the model. They conduct the analyses provided by SCAR (or by the integration of SCAR with other tools) to verify that these properties are consistent with the basic model. Secondly, they use the code generation capabilities provided by SCAR to automatically create a source code skeleton in the desired target programming language, annotated with specification in the language of a formal verification tool suitable for that language. They proceed to implement the functions such that the verification tool is able to prove that the implementation is correct w.r.t. the generated specification. This process is inspired by the Design by Contract paradigm [Mey92].

Goals of the SCAR Approach

The main goal of the SCAR approach proposed in this work is to give developers the possibility to succinctly specify application-level security and correctness properties of smart contract applications, and to equip them with a process that yields an application which is guaranteed to fulfill these properties.

SCAR also serves more abstract purposes. It ties in with a general drive to "shift left security", an approach in software development which aims to factor in security as early in the software development cycle as possible. Furthermore, it aims to reduce the semantic gap between complex application-level properties and their instantiation on the source code level.

Scar is intended to enable different formal analysis tools. The separation of model and implementation is beneficial especially when something changes: When a function implementation is adapted, does this affect any application-level correctness properties? The separation between model and code reduces this question to whether the changed function still fulfills its contract. Similarly, Scar may reduce the necessary effort in case of a change in requirements.

Specific design decisions

Scope of an Application As described in Section 1.2, a smart contract can be viewed as an object that defines a namespace and encapsulates some functionality that may operate on that namespace. A smart contract application, then, could be defined as the set of contracts which is concerned with performing a certain task. Since smart contracts can call other contracts, and have contract-typed state variables, a smart contract application can also be viewed as the transitive closure of an initial set of contracts: Every contract that is called by the application or in the state of an application becomes part of the application. This is the view we take in Scar.

Note that, in practice, this may mean that developers have to specify contracts that are already deployed, *e.g.*, existing libraries, if these contracts are to be called from the planned application.

Function Abstraction In Scar, functions are abstracted by first-order logic function contracts, instead of being defined in terms of a program. This has benefits: First, it allows introducing specification constructs that are not existing in some programming languages, while still including all necessary domain-specific elements. This is especially relevant for quantifiers and sum operators, which are needed to reason about unbounded data types.

Most importantly, though, this abstraction can be viewed as a change of perspective of the developer: A Scar model describes *what* a function does, as opposed to *how* it does it.

Under-specification of Functions In Scar, developers can choose to leave functions under-specified. Function contracts are not necessarily deterministic and allow different concrete implementations of the same function.

This brings flexibility: If an under-specification still suffices to prove some application-level property, then it has fulfilled its purpose, and the implementation details do not matter as long as the implementation is consistent with the model.

Precondition Semantics In specification languages which allow the specification of function or method contracts, the precondition has to be met by the caller of the function (*e.g.*, in JML [LC06] or ACSL [Bau+08]), and if the precondition is not met, the behavior of the function is not defined. In the adversarial environment of smart contracts, where functions are called by untrusted and possibly malicious actors, this does not make sense. Therefore, we adapt slightly different semantics where if the precondition is not fulfilled, no transaction happens and the state remains unchanged. If the precondition is fulfilled, then the function must terminate in a state that fulfills the postcondition (as with other comparable specification formalisms). This is also the semantics that formal verification tools for the Ethereum platform have adopted.

Contracts as Data Types Should an application model be static, in the sense that every individual contract is described by it, and the set of contracts remains the same? Or can contracts be created dynamically over the execution of an application?

There exists a trade-off between expressive power and complexity here. In many real-world applications, contracts are in fact created dynamically, and to describe these applications, the modeling language needs to allow this. On the other hand, this increases the complexity of the specification, and makes it harder for developers to intuitively grasp the meaning of a model.

Scar tries to find a good compromise in this regard. Contracts are data types, and can be dynamically created during execution. However, simple, static applications can still be described easily.

Time In smart contract networks, there are two different notions of time. The first is concerned with the time at which a transaction was executed, or at which a block was mined, respectively. The second notion is about the order of blocks and transactions. Intuitively, the first notion is about timestamps, while the second is about block (or transaction) numbers.

Functions in smart contracts can make reference to system time. Therefore, specifying correctness and security properties requires reasoning about the system time. One example is the auction application presented in Section 1.4, in which the `close` function can only be called after a predefined period of time has elapsed.

Block numbers are always increasing by 1, but can be unreliable for time estimation due to changes in block difficulty and other normal fluctuation.

Block timestamps are different between platforms. In Ethereum, they are set by the miner of a block. By the Ethereum specification, the timestamp must be larger than that of the previous block, but not by more than 900 seconds.

However, most Ethereum clients (such as Geth [Gol18]) reject blocks if their timestamp is larger than that of the client attempting to validate it. In practice, this limits the miners' flexibility with the timestamp to a few seconds, and makes timestamps a much more reliable base for estimating elapsed time than block numbers. This is also true for the other platforms, e.g, Solana, where timestamps are calculated on request, as an average over timestamps on recent blocks [Sol24b].

Since both notions of time can reasonably be useful in specification, both timestamp (`\systime`) and block number (`\blocknum`) are specification elements in SCAR.

3.3 SCAR Application Metamodel

This section presents the core of the SCAR approach, namely, its smart contract application metamodel. First, we present the basic model elements and SCAR's type system. Then, we define the specification language used within the model to describe the behavior of functions in terms of functions contracts.

Basic Model Elements and Type System

Figure 3.2 gives an overview of the basic metamodel. The topmost element of the metamodel is the *application*. It consists of *contract* elements. Contracts, in turn consist of *stateVariable* and *function* elements. State variables have a name and a type. Functions have a name, a list of named and typed parameters, an optional return type, and a function contract written in the SCAR functional specification language (see Section 3.3 below).

In order to describe the initialization of an application, an initial condition can be specified for each contract in the same specification language. To closely reflect the constructors in smart contract programming languages, the initial condition can be parameterized.

The SCAR type system is visualized in Figure 3.3. In it, there are four primitive types: Booleans (*Bool*), signed and unsigned integers (*Int* and *UInt*, respectively), and strings (*String*). Accounts (*Account*) can be either external accounts, representing a real-world entity, or contracts. Both have a non-negative balance, and every account has a unique ID. Each contract in an application defines its own type, which consists of a name, a list of typed and named state variables, and a list of the functions of that contract. Furthermore, there are two composite types: Arrays and Mappings, which map keys of a primitive type to values (whose type is not restricted). In addition, there are two user-defined

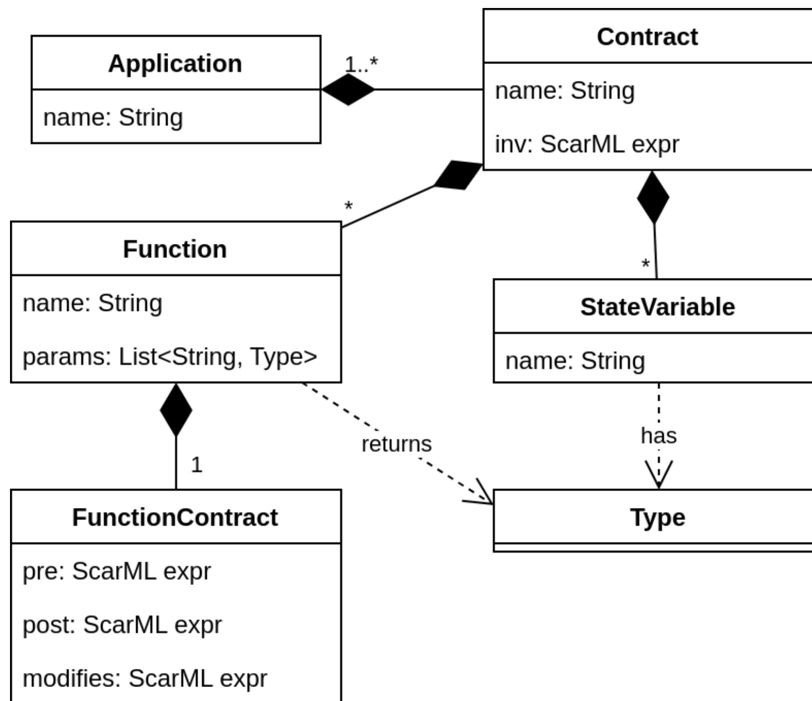


Figure 3.2: The core metamodel. The Type hierarchy is presented in Figure 3.3. Function contracts and invariants are expressions in the ScarML functional specification language described in Figure 3.4.

types: Enums have a String list of constant names. Structs are records with a list of named and typed fields.

In order to avoid problems with infinite recursion in the type system, we restrict user-defined types to be non-recursive. This means that no struct or contracts type may contain a field or state variable of its own type, or a collection thereof.

ScarML

The ability to specify the behavior of a smart contract application in a modular way is a core element of the Scar approach. In this section, we introduce ScarML, the Scar modeling language. ScarML is a specification language for contract invariants (for specifying properties of smart contract instances) and function contracts (for specifying the behavior of single functions). Function contracts consist of preconditions, postconditions, and frame conditions. Invariants, preconditions, and postconditions are defined as Boolean *specification expressions*. Frame conditions are specified as *frame expressions*.

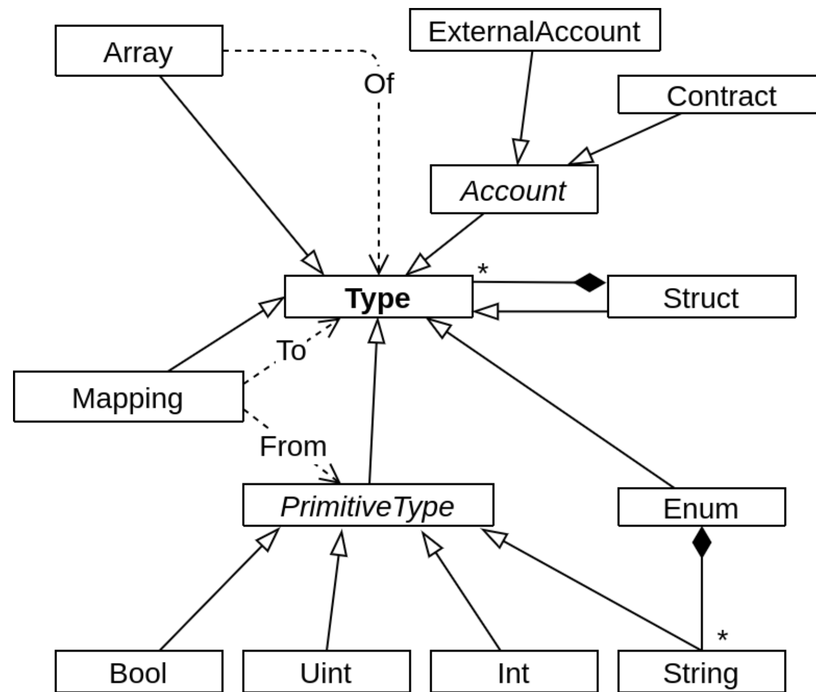


Figure 3.3: A graphical description of the type system

The grammar of ScarML specification expressions is defined in Figure 3.4. Expressions are typed.

Each state variable in an application model is a term of the variable's type. Elements of composite data types are also terms. Terms of number types can be combined with arithmetic and comparison operators. Terms of Boolean type can be negated and combined with the standard Boolean connectors. Existentially and universally quantified expressions are also allowed. There are two equality operators, reflecting value equality and referential equality (see Section 3.4).

There is a special operator for evaluating a term in the pre-state of a transaction (`\old`). Furthermore, there are some constructs that represent the context of a transaction, *i.e.*, the caller, the amount transferred with the call, the system time of the call, and the number of the block in which the transaction is executed. Another domain-specific term is the `\hash` operator. Furthermore, the specification can contain terms representing a (pure) function call. All Boolean, integer, and string literals are also terms. Terms of Boolean type can be used as top-level expressions in invariants, preconditions and postconditions.

We provide some set-typed terms for reasoning and quantifying over primitive data types and elements of unbounded data types: `\keys()` and `\values()`

$$
\begin{array}{rcl}
\phi & ::= & v \mid v\texttt{[}t\texttt{]} \mid v.id \\
& \mid & \texttt{[1-9][0-9]+} \\
& \mid & \texttt{true} \mid \texttt{false} \\
& \mid & \phi\ \texttt{[== | !=]}\ \phi \\
& \mid & \phi\ \texttt{[=== | !==]}\ \phi \\
& \mid & \texttt{!}\phi_1 \mid \phi_1\ \texttt{[\&\& | || | =>]}\ \phi_2 \\
& \mid & \texttt{-}\phi_3 \\
& \mid & \phi_3\ \texttt{[+ | - | * | / | \%]}\ \phi_4 \\
& \mid & \phi_3\ \texttt{[< | <= | >= | >]}\ \phi_4 \\
& \mid & [\forall \mid \exists]\ \texttt{(}qVar\texttt{)}\ \texttt{:}\ \phi \\
& \mid & \texttt{\textbackslash result} \mid \texttt{\textbackslash old(}t\texttt{)} \\
& \mid & \texttt{\textbackslash caller} \mid \texttt{\textbackslash amt} \mid \texttt{\textbackslash systime} \mid \texttt{\textbackslash blocknum} \\
& \mid & \texttt{\textbackslash send(}\phi_3\texttt{,}\phi_5\texttt{,}\phi_6\texttt{)} \\
& \mid & \texttt{\textbackslash hash(}\phi\texttt{)} \\
& \mid & \texttt{\textbackslash sum(}\phi_7\texttt{)} \\
& \mid & StringLiteral \\
& \mid & funName\texttt{(}args\texttt{)} \\
& \mid & \texttt{\textbackslash values(}\phi\texttt{)} \mid \texttt{\textbackslash keys(}\phi\texttt{)} \mid \texttt{\textbackslash size(}\phi\texttt{)} \\
& \mid & \texttt{\textbackslash in} \\
& \mid & \texttt{\textbackslash creates}\ type\texttt{:}id\texttt{(}params\texttt{)}
\end{array}
$$

Figure 3.4: The grammar of ScarML. ϕ_1 and ϕ_2 are of type bool, ϕ_3 and ϕ_4 are number-typed, ϕ_5 and ϕ_6 are of type account, and ϕ_7 is a set of number-typed values

refer to the domain and the set of elements of an array or mapping, respectively. For these, the `\in` keyword signifies set inclusion. The `\size()` keyword refers to the number of elements of an array or mapping variable.

A bounded sum operator `\sum()` is provided, which takes a number-typed set expression, like an integer array or the values of a mapping to unsigned integers.

Within a quantified expression, the quantification variable is written *v: r* where *v* is an identifier and *range* is either the name of a primitive type (*i.e.*, int, uint, string, or account), or a set-typed expression, like `\keys()` or `\values()`.

The `\creates()` keyword expresses that a new contract of a given type with given parameters is created as part of a function execution.

Term Types Terms representing variables, elements, contract members are of the type of the element they represent. A function call term is of the type that the function returns. Number literal terms are of type uint. The unary

frameExpr ::= `modifies :` *id* | *id.e* | *id.** | *id*`[`*e*`]` | *id* `[`*e1*`..`*e2*`]` | *id*`[*]`

Figure 3.5: The grammar for ScarML frame condition expressions

minus yields a term of type `int`.

The arithmetic functions are polymorphic. Both subterms can be either of type `int` or `uint`. With the exception of subtraction, the resulting term is of type `uint` iff both subterms are also of type `uint`. Otherwise, it is of type `int`. Subtraction always yields a term of type `int`.

The `\old()` operator yields a term of the same type as the term it contains. The `\result` term can only be used within a function postcondition and is of the type this function returns.

The domain-specific special terms `\amt`, `\systime`, `\blocknum`, and `\hash()` are of type `uint`. The `\caller` term is of type `account`. The `\send()` term is of Boolean type.

A function call term can only refer to a *pure* function. We define a function to be pure if it is side-effect free, *i.e.*, it does not modify the state in any way.

For reasoning and quantifying over primitive data types and elements of unbounded data types, we introduce some terms that are set-typed: `\keys()` and `\values()` yield finite sets of the type of the array or mapping they refer to. `\size()` is of type `uint`. Set-typed expressions can also occur within range expressions of quantifiers.

Frame conditions In function contracts, ScarML allows the specification of frames, *i.e.*, the parts of the state that a function may modify. The frame condition syntax in ScarML is given in Figure 3.5.

3.4 Scar Semantics

We develop the semantics of the Scar metamodel. First, we define the domains of the different types, and the locations that arise from a given application. With this, we can define the state of an application.

Going forward, we define the semantics of a Scar model in terms of execution traces consisting of environment and application steps. In the last part of this chapter, we define the semantics of ScarML.

Types and Locations in the Basic Metamodel

In order to define the state of an application, we first require some preliminary definitions. First, we say that $\mathcal{A}$ is the abstract set of all accounts. In an appli-

Type t of variable a	*ValsOf*(t)
Int	$\mathbb{Z}$
UInt	$\mathbb{N}_0$
Bool	$\{\top, \bot\}$
String	*String*
Account	$\mathcal{A}$
Enum$(c_0, ..., c_i)$	$\{c_0, ..., c_i\}$
mapping$(t_1 \Rightarrow t_2)$	$\{f : t_1 \rightarrow t_2\}$
mapping$(t_1 \Rightarrow t_3)$	$\{f : t_1 \rightarrow t_3\}$
$t[]$	$\{f : UInt \rightarrow t\}$
Struct	$\{f : a.fields \rightarrow AllValues\}$
Contract	$\{f : a.vars \rightarrow AllValues\}$

Table 3.1: The set of values a variable can assume. (with t_2 a primitive type and t_3 a non-primitive type).

cation a, at each point in time, there are two sets of accounts *in the application*: $\mathcal{C}_a$, the set of contracts in the app, and $\mathcal{E}_a$, the set of external accounts in the app. These two sets are first defined in the initial step (see below) and change over time. Together, they form $\mathcal{A}_a = \mathcal{C}_a \cup \mathcal{E}_a$, the combined set of all "known" accounts.

Each Scar type defines a set of possible values that a variable of this type can assume. We define a function *ValsOf* which maps Scar types to the set of their possible values:

Definition 1 *(The ValsOf function) The function ValsOfmaps Scar types to subsets of AllValues as defined in Table 3.1.*

We describe the state of an application in terms of *Locations*. Each state variable *v* defines a set of locations according to its type. Variables of primitive type define exactly one location, which is described simply by the name of the variable. Composite-type variables define sets of locations, with each location described as a tuple. The size of the tuple is defined by the nesting depth of the type. For example, an *Int* array a defines a set of locations, each of which is described by a tuple (a, x), where x is an integer. In analogy to the *ValsOf* function, we define the *LocsOf* function in Table 3.2:

Definition 2 *(The LocsOf function) The function LocsOfmaps Scar types to sets of locations as defined in Table 3.2.*

With this, the set of locations of an application *a* is $Locs_a = \bigcup_{c \in \mathcal{C}_a} LocsOf(c)$, and the state of an application $\mathcal{S}_a : Locs_a \rightarrow AllValues$ is a function that maps

Type t of variable a	$LocsOf(a)$
Int, *UInt*, *Bool*, *String*	$\{a\}$
Account	$\{a, a.balance\}$
Enum$(c_0, ..., c_i)$	$\{a\}$
mapping$(t_1 \Rightarrow t_2)$	$\{a\} \times ValsOf(t_1)$
mapping$(t_1 \Rightarrow t_3)$	$\{a\} \times ValsOf(t_1) \times LocsOf(t_3)$
$t[t_2]$	$\{a\} \times \mathbb{N}_0$
$t[t_3]$	$\{a\} \times \mathbb{N}_0 \times LocsOf(t_3)$
Struct	$\{a\} \times LocsOf(fields(a))$
Contract	$\{a\} \times LocsOf(stateVars(a))$

Table 3.2: The set of and the set of locations defined by a variable of each type (with t_2 a primitive type and t_3 a non-primitive type)

locations to values. Where it is clear (or irrelevant) which application is referred to, we omit the subscript.

State variables are never uninitialized, but arrays and mappings can have elements that have not been set. In accordance with Solidity's state defaults, the state function is defined as a total function over all locations, even if those locations have not been initialized. In that case, the location is defined to be in the default state. We define the default values for the Scar types as follows:

Definition 3 *(Default values) The function defValOf maps Scar types to values as defined in Table 3.3.*

The *defaultAccount* is a special value of type *Account*. For composite types, the default value is a variable with all elements set to the default value of the element type, *e.g.*, the default value for a variable of type `mapping(account=>int)` is the function mapping all accounts to 0.

Semantics of ScarML

We define the semantics of ScarML in terms of an evaluation function over the state.

Evaluation Function Invariants and pre- and postconditions are evaluated in a *step* τ_i of the execution, which consists of the call context ctx_i and a state s_i (cf. the description of the trace semantics below).

Definition 4 *(Evaluation Function) The* ***partial evaluation function*** $[\![\cdot]\!]$ *is defined in Figure 3.6. It maps terms of the ScarML specification language to the set AllValues of all possible values. The evaluation function is parameterized with a call context.*

Type t of variable a	$defValOf(a)$
Int	0
UInt	0
Bool	$\bot$
String	“”
Account	*defaultAccount*
Enum($c_0, ..., c_i$)	c_0
Composite type	f(x) = *defValOf*(x)

Table 3.3: The default value for each type. The default value for the String type is the empty string. For the account type, a special value *defaultAccount* is introduced. For the composite data types, the possible values are functions; For these types, the default value is the constant function which maps everything to the default value.

The evaluation function is not defined for terms containing out-of-bounds accesses or a division by zero.

Definition 5 *A condition c is **fulfilled** (or satisfied) in a step τ_i if $[\![c]\!]_{\tau_i} = \top$. A condition c is **satisfiable** if there exists a possible step (i.e., a combination of a state s and a call environment) τ such that c is fulfilled in τ.*

Set-typed Expressions To make quantification over composite data types more intuitive, ScarML contains some set-typed expressions. However, the language is limited to very basic expressions representing the elements of such composite types. We judge that introducing set operators increases the complexity of specification and verification while providing only very little benefit.

The `\keys` and `\values` expressions are set-typed. Each variable of the unbounded array and mapping data types defines a function. The domain of this function consists of all values for which the function is defined, which corresponds to the set of mapping keys or array indices that have been assigned a value. The codomain is the set of all these values. The `\keys` and `\values` expressions map a variable to its domain or codomain, respectively.

The exact semantics of what constitutes the domain of an array or a mapping is specific to the application platform (or even to a concrete verification tool). Therefore, the translation of these expressions to the source code annotations is crucial.

Function frames Frame conditions in ScarML define what part of an application’s state may be modified by the function. The grammar is given as a part

$$
\begin{aligned}
[\![\texttt{true}]\!] &:= \top \\
[\![\texttt{false}]\!] &:= \bot \\
[\![v]\!] &:= s_i(v) \\
[\![v[t]]\!] &:= s_i((v, [\![t]\!])) \\
[\![\texttt{-}\phi]\!] &:= -[\![\phi]\!] \\
[\![\phi_1 + \phi_2]\!] &:= [\![\phi_1]\!] + [\![\phi_2]\!] \\
[\![\phi_1 - \phi_2]\!] &:= [\![\phi_1]\!] - [\![\phi_2]\!] \\
[\![\phi_1 * \phi_2]\!] &:= [\![\phi_1]\!] * [\![\phi_2]\!] \\
[\![\texttt{\textbackslash result}]\!] &:= return_i \\
[\![\texttt{\textbackslash old(}t\texttt{)}]\!] &:= [\![t]\!]_{s_{i-1}} \\
[\![\texttt{\textbackslash caller}]\!] &:= caller_i \\
[\![\texttt{\textbackslash amt}]\!] &:= amt_i \\
[\![\texttt{\textbackslash systime}]\!] &:= systime_i \\
[\![\texttt{\textbackslash blocknum}]\!] &:= blocknum_i \\
[\![\texttt{\textbackslash send(}\phi_3\texttt{,}\phi_4\texttt{,}\phi_5\texttt{)}]\!] &:= send([\![\phi_3]\!], [\![\phi_4]\!], [\![\phi_5]\!]) \\
[\![\texttt{\textbackslash hash(}t\texttt{)}]\!] &:= hash([\![t]\!]) \\
[\![\phi_1 \texttt{ == } \phi_2]\!] &:= [\![\phi_1]\!] =_{val} [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ != } \phi_2]\!] &:= [\![\phi_1]\!] \neq_{val} [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ === } \phi_2]\!] &:= [\![\phi_1]\!] =_{ref} [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ !== } \phi_2]\!] &:= [\![\phi_1]\!] \neq_{ref} [\![\phi_2]\!] \\
[\![\texttt{!}\phi]\!] &:= \neg[\![\phi]\!] \\
[\![\phi_1 \texttt{ \&\& } \phi_2]\!] &:= [\![\phi_1]\!] \wedge [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ || } \phi_2]\!] &:= [\![\phi_1]\!] \vee [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ => } \phi_2]\!] &:= [\![\phi_1]\!] \Rightarrow [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ < } \phi_2]\!] &:= [\![\phi_1]\!] < [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ <= } \phi_2]\!] &:= [\![\phi_1]\!] \leq [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ >= } \phi_2]\!] &:= [\![\phi_1]\!] \geq [\![\phi_2]\!] \\
[\![\phi_1 \texttt{ > } \phi_2]\!] &:= [\![\phi_1]\!] > [\![\phi_2]\!] \\
[\![\forall\ qVar : \phi]\!] &:= \forall x \in RangeEval(qVar) : [\![\{qVar/x\}\phi]\!] \\
[\![\exists\ qVar : \phi]\!] &:= \exists x \in RangeEval(qVar) : [\![\{qVar/x\}\phi]\!] \\
[\![\texttt{\textbackslash sum(}a\texttt{)}]\!] &:= \textstyle\sum_{i \in domain(a)} [\![a]\!](i) \\
[\![\texttt{\textbackslash keys(}m\texttt{)}]\!] &:= domain(m) \\
[\![\texttt{\textbackslash values(}m\texttt{)}]\!] &:= codomain(m) \\
[\![\texttt{\textbackslash size(}m\texttt{)}]\!] &:= |codomain(m)| \\
[\![v(t)]\!] &:= s_i((v, [\![t]\!])) \text{ if } [\![t]\!] \in domain(v) \\
[\![\phi_1/\phi_2]\!] &:= [\![\phi_1]\!]/[\![\phi_2]\!] \text{ if } [\![\phi_2]\!] \neq 0 \\
[\![\phi_1\%\phi_2]\!] &:= [\![\phi_1]\!] mod [\![\phi_2]\!] \text{ if } [\![\phi_2]\!] \neq 0 \\
a\ mod\ n &:= a - n * (a/n)
\end{aligned}
$$

Figure 3.6: The ScarML evaluation function $[\![\cdot]\!]$. The function is parameterized with the step τ_i, which contains the environment values $caller_i$, amt_i, $systime_i$, $blocknum_i$, and the prestate s_{i-1} and poststate s_i

$Expr$	$[\![Expr]\!]_{loc}$
`\nothing`	$\emptyset$
id	$\{id\}$
id.e	$([\![id]\!]_{loc}, [\![e]\!]_{loc})$
*c.**	$[\![c]\!]_{loc} \times statevars(c)$
*s.**	$[\![c]\!]_{loc} \times fields(c)$
id[*e*]	$([\![id]\!]_{loc}, [\![e]\!]_{loc})$
m(*)	$[\![a]\!]_{loc} \times dom(m)$
m[*]	$[\![a]\!]_{loc} \times \mathit{ValsOf}(t_1)$
a[*e1*..*e2*]	$[\![a]\!]_{loc} \times \{n \in \mathbb{N} \mid [\![e1]\!] \leq n \wedge n < [\![e2]\!]\}$
a(*)	$[\![a]\!]_{loc} \times dom(a)$
a[*]	$[\![a]\!]_{loc} \times \mathbb{N}$

Figure 3.7: Definition of the function frame evaluation function $[\![\cdot]\!]_{loc}$, where *c* is a contract, *s* is a struct, *m* is a mapping from type t_1 to t_2 and *a* is an array

of the basic model grammar in Figure 3.9. The semantics of the frame condition is defined by the location evaluation function $[\![\cdot]\!]_{loc}$.

Definition 6 *(Frame condition evaluation) The* ***frame evaluation function*** $[\![\cdot]\!]_{loc}$ *is defined in Figure 3.7. It maps SCARML location expressions to locations in the model (cf. Section 3.4).*

Note that a function annotated with a `\nothing` frame condition may still create a new contract. However, the new contract cannot be referred to from any location within the application, since that would require the modification of the application state.

Unbounded Types and Default Values Mappings and arrays are finite, but unbounded dynamic data structures in SCAR. We treat them like functions: The domain of an array `a` is the set of all indices for which an element has been set. In SCARML, we provide two kinds of access to these data structures. The first one is *functional* access, expressed by parentheses (“ `a(i)` ”). Functional access is defined only if the index (or key) is in the domain, and remains undefined otherwise.

The second way of accessing elements is close to what a Solidity developer might expect. It is expressed with brackets (“ `a[i]` ”) and always defined. If no element has been created for a given key or index, the access will yield a default value (as specified in Table 3.1).

Equality ScarML offers two equality operators (namely, "`==`" and "`===`", as well as the corresponding inequalities), so that developers can distinguish between referential equality and value equality. For this, we define two equality predicates $=_{ref}$ and $=_{val}$. Two expressions are referentially equal if they refer to the exact same location.

To be of equal value, two expressions must have the same type. For primitive types, two expressions are considered equal iff their values are equal. Two account-typed expressions are equal iff they refer to accounts with the same ID. Expressions of composite types are value equal if all their elements are value-equal.

Note that $x =_{ref} y$ implies $x =_{val} y$.

Division by Zero Division by zero is treated in different ways by different programming languages and verification tools. In Solidity, a division by zero results in an error, which leads to a function reverting without state change. In some logic tools, *e.g.*, SMTLIB2 and the z3 SMT solver, division by zero is undefined. The solc-verify tool is based on these technologies. Therefore, we choose to adopt these semantics for the Scar specification language because it limits the risk of discrepancies between the model-level and implementation-level specification.

Semantics of a Scar Model

In this section, we define trace-based semantics of a full Scar model, *i.e.*, a basic model with function contracts and invariants.

Definition 7 *(Model plausibility) A Scar model m is* ***plausible*** *if*

- *all contract invariants are satisfiable, and*
- *all function pre- and postconditions are satisfiable, and*
- *the initial conditions of each contract type in m are satisfiable, and*
- *for each contract type in m, the initial conditions of m imply each contract invariant, i.e., every possible state which satisfies the initial conditions must also satisfy every contract invariant.*

For a given Scar model, an execution of the application is an infinite trace of steps $\tau_0, \tau_1, \tau_2, \ldots$ where each step τ_i is either an *environment step* or an *application step.*

Initial step The initial step τ_0 is defined in the `appInit` part of the application specification. In it, the initial set of contracts is declared, along with a list of parameters according to the initial parameter types that each contract requires. After the initial step, the set of known contracts $\mathcal{C}$ consists of all contracts declared in `appInit`, as well as all contract-typed locations within these (*i.e.*, state variables of contract type, as well as contract-typed elements of arrays, mappings, and structs).

Similarly, the set of known external accounts is initialized with all account-typed state variables and account-typed elements of arrays, mappings, and structs.

For τ_0 to be valid, the initial condition of each contract in $\mathcal{C}$ must be fulfilled.

Environment steps An environment step characterizes a step in which no function of the application is called, and in which the application state does not change, with the possible exception of account balances. Furthermore, block number and system time can increase.

Formally, an environment step tau_i consists of

- ctx_i, the context of the step, consisting of
  - $systime_i$, the system time of the step
  - $blocknum_i$, the block number of the step
- s_i, the state of the application after τ_i

Definition 8 *(Validity of an environment step) For an environment step τ_i to be **valid**, the following conditions must hold:*

- $blocknum_i = blocknum_{i-1} \vee blocknum_i = blocknum_{i-1} + 1$
- $systime_i = systime_{i-1}$ *if* $blocknum_i = blocknum_{i-1}$
- $systime_i > systime_{i-1}$ *if* $blocknum_i = blocknum_{i-1} + 1$
- $\forall\, l \in Locs : \begin{cases} s_i >= s_{i-1}, \textit{ l is balance} \\ s_i = s_{i-1}, \textit{ else} \end{cases}$

Application steps An application step is the result of a call to one of the application's functions. Formally, an application step τ_i consists of

- ctx_i, the call context, consisting of
  - $systime_i$, the system time of the step

    - $blocknum_i$, the block number of the step

- f_i, the function that was called to reach τ_i, consisting of
    - $caller_i$, the caller of the function
    - amt_i, the amount transferred with the function call
    - $params_i$, the call parameters
    - pre_i, the precondition of f_i
    - $post_i$, the postcondition of f_i
    - $return_i$, the return value of f_i
    - mod_i, the set of locations f_i may modify
- s_i, the state of the application after the execution of f_i

Definition 9 *(Validity of an application step) For an application step τ_i to be **valid**, the following conditions must hold:*

- $blocknum_i = blocknum_{i-1} \vee blocknum_i = blocknum_{i-1} + 1$
- $systime_i = systime_{i-1}$ *if* $blocknum_i = blocknum_{i-1}$
- $systime_i > systime_{i-1}$ *if* $blocknum_i = blocknum_{i-1} + 1$
- pre_i *must be fulfilled in* τ_{i-1}
- $post_i$ *must be fulfilled in* τ_i
- $\forall\, l \in Locs \setminus mod_i : \begin{cases} s_i >= s_{i-1}, l \text{ is balance} \\ s_i = s_{i-1}, \text{else} \end{cases}$

Trace Semantics With this, we now define the semantics $\mathcal{S}$ of a SCAR model as the set of all infinite traces of valid steps. Environment and application steps can occur in any order:

Definition 10 *(SCAR trace semantics) For a plausible SCAR model m, the semantics $\mathcal{S}(m)$ is the set of traces $\{\mathrm{T} := \tau_0, \tau_1, \tau_2, ...\}$ where τ_0 is a valid initial step for m, and each $\tau_{i \in \mathcal{N}_{>0}}$ is either a valid environment step or a valid application step.*

A visualization of the SCAR trace semantics is presented in Figure 3.8.

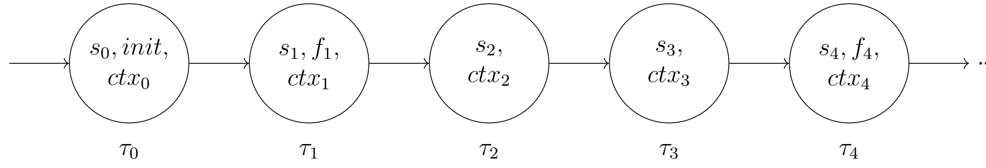


Figure 3.8: An example execution trace. Steps τ_3 and τ_4 are environment steps.

3.5 Scala Implementation of the Scar approach

Scar has been implemented in the Scala programming language. The repository [Sch24] is publicly available. The project's core is the `model` package, in which all model elements are defined as Scala classes. The top-level model element is the smart contract application `SCApp`.

The syntax of the `.scar` model is defined in an ANTLR grammar. A simplified overview grammar for the basic metamodel is shown in Figure 3.9. First, the user-defined types are specified, followed by the initial configuration of the application. Then, the contracts are declared with their state variables, initial conditions, invariants, and functions (including function contracts).

The `translation` package contains all functionality for translating a text file in the Scar format into the corresponding Scala model. This includes type checking and other checks of the application and the provided specification. The `generation` package contains functionality to translate an `SCApp` to another formalism, *e.g.*, annotated Solidity. Finally, the `analysis` package contains functionality for model-level analysis.

The Scar implementation is an instantiation of the model-driven approach sketched in Section 3.2. Developers write a file in the `.scar` format and check it for syntactic correctness and plausibility (cf. Definition 7). Then, they proceed in two directions: First, they determine what requirements the application as a whole must fulfill, and specify these properties using the approaches detailed in the following chapters. Second, they use one of the translations that the Scar project provides to translate their model to a code skeleton with formal annotations. After finishing the implementation of the functions, they prove them correct w.r.t. the generated specification. This may require some iterations, but yields an application that fulfills the requirements specified on the model level.

Apart from this envisioned use, the Scar project is also intended to provide a platform for the integration of other approaches in the domain of smart contract verification. It is therefore designed to be extensible.

ScarML can be translated into many other existing specification languages which have been designed for a specific tool, making these tools more broadly applicable. In this way, Scar may also serve as a platform for comparing dif-

app	::=	`application` : *userType∗ appInit contract+*
userType	::=	*structDef* | *enumDef*
structDef	::=	`struct` *name* : {[id : typeExpr]+}
enumDef	::=	`enum` *name* : {[id]+}
appInit	::=	`appInit` : [*id* : *contractType*(*params*)]+
contract	::=	*init*? *id state functions*
init	::=	*initParams*? *initCond+*
initParams	::=	`initParams` : [*id* : *typeExpr*]+
initCond	::=	`init` : *specExpr*
state	::=	`state` : *stateVar+*
stateVar	::=	`var` *id* : *typeExpr*
functions	::=	`functions` : *function+*
function	::=	`fun` *id* : *params*? *ret*? *pre∗ post∗ frame∗*
params	::=	`params` : [*id* : *typeExpr*]+
ret	::=	`returns` : *typeExpr*
pre	::=	`pre` : *specExpr*
post	::=	`post` : *specExpr*
typeExpr	::=	*primitiveType* | *arrType* | *mapType* | *id*
primitiveType	::=	`bool` | `int` | `uint` | `string`
arrType	::=	*typeExpr* `[]`
mapType	::=	`mapping(`*typeExpr*`=>`*typeExpr*`)`
frame	::=	`modifies` : *frameExpr*

Figure 3.9: A grammar for the basic metamodel. The *specExpr* symbol is defined in the functional specification language grammar (see Figure 3.4). The *frameExpr* symbol is defined in Figure 3.5.

ferent formal verification tools and approaches.

3.6 Consistency between Model and Code

In order to deploy an application, a model needs to be converted into platform-specific source code. The main point of our model-driven approach is that properties that were proven to hold for the model also hold for the implementation. Therefore, the conversion from model to code needs to be semantics-preserving: The implementation must be a refinement of the model. This means that for every possible execution of the implementation, when deployed, there must be a corresponding trace in the model semantics.

To show that this is the case, the semantics of the platform and of the verification tool have to be taken into account, and it has to be shown that each

execution of a deployed application corresponds to an execution trace of its model. This argument has to be made separately for each target platform.

In our approach, we achieve a refinement relationship by generating formal specification that matches the model annotation, and expecting the final implementation to conform to this generated specification.

In this section, we describe the code and annotation generation that translates a Scar model into a Solidity source code skeleton annotated in the specification language of the solc-verify tool. We then argue why (and under which circumstances) an implementation that fulfills the generated specification is indeed a refinement of the Scar model.

Solidity Code Generation

From a Scar model, code is generated as follows: First, a Solidity version header is prepended to the file. Then, a library contract `UTIL` containing the user-defined data types is created. Scar structs and enums are translated to their respective Solidity counterparts.

For every contract in the Scar model, a corresponding Solidity `contract` of the same name is created. Since Scar was written with Solidity as the main target language, each Scar type has a directly corresponding Solidity type, making the translation of state variables, function parameters, and return types straightforward. For functions, only headers are generated, which consist of the function name, the translated function parameters, the `public` modifier indicating that the function can be called from everywhere, as well as the return type.

Generation of solc-verify Annotations

solc-verify [HJ20] is a tool for formal verification of Ethereum smart contracts written in Solidity. It takes formally annotated Solidity smart contracts as input and translates them to programs in the Boogie [Bar+06] intermediate verification language. From Boogie, verification conditions are generated and then discharged using SMT solvers.

As in Scar, the main specification elements in solc-verify are invariants and function contracts. Furthermore, loop invariants can be specified to aid the verification tool. The specification language is a combination of first-order logic and a subset of Solidity. It contains quantifiers, a bounded sum operator, and an “old” construct for accessing the pre-state of a function from within a postcondition. The specification must be in prenex normal form.

ScarML is translated to solc-verify as follows:

- Boolean and arithmetic operators are translated to their direct counterparts.
- If the `\result` keyword occurs in SCAR in a function postcondition, the return value of the Solidity function is named `result`, and the expression is translated as such.
- The `\old` keyword is translated using SOLC-VERIFY's `__verifier_old_t` prefix, where `t` is the type of the translated expression.
- The special values `\caller`, `\amt`, `\systime`, and `\blocknum` are translated to the Solidity keywords `msg.sender`, `msg.value`, `block.timestamp`, and `block.number`, respectively.
- The `\send` element is translated in its de-sugared form as assignments to the balances of sender and receiver.
- Equality: SOLC-VERIFY does not provide different equality operators. In the case of composite data types, equality in SOLC-VERIFY is referential equality. For value equality on composite data types, the translation is such that equality of all elements is required. For arrays and mappings, this produces a universal quantifier. For structs, it produces a number of equalities depending on the number of fields of the struct type.
- The `\sum` operator is translated to SOLC-VERIFY's bounded sum expression.
- Array and mapping access with the `[]` operator is translated directly. If the functional access `a(i)` occurs in a function contract, an additional precondition is introduced, which states that the accessed element `a[i]` must not be the default element for that type. If `a(i)` occurs in a contract invariant, the same condition is added as an additional invariant.
- `\size(a)` is translated as `a.length` if a is an array; otherwise, the translation fails.
- The set-typed expressions are translated only when they occur in a quantification variable. The translation is as follows:
  - `\forall (x \in values(a))` for an array `a` is translated as a quantifier over a: `forall (uint i)0 <= i && i < a.length`. The matrix of the quantified expression is appended as a conjunction to the bounds on the index variable, and all occurrences of `x` are replaced with the array element `a[i]`.

- For a mapping variable `m`, `\forall (x \in values(m))` is translated as `forall (t i)m[i] != val_default`, where `t` is the key type of the mapping `m`, and val_default is the default value for the values of `m`. In the matrix of the translated expression, the occurrences of `x` are replaced by `m[i]`.
- Analogously, for a mapping variable `m`, `\forall (x \in keys(m))` is translated as `forall (t i)m[i] != val_default`, where `t` is the key type of the mapping `m`, and val_default is the default value for the values of `m`. In the matrix of the translated expression, the occurrences of `x` are replaced by `i`.

solc-verify does not support function calls in the specification. Therefore, the code generation fails if such a call occurs. The same is true for the Scar `hash` primitive, since it also constitutes a function call. If needed, developers may work around the verification tool limitations by substituting the necessary properties in the original Scar specification.

All other model elements can be translated. An example of the code generation can be seen in Figure 3.10 and Figure 3.11, showing a simple bank application modeled in Scar and the translation to Solidity, respectively.

Consistency

Consistency between a Scar model and an implementation means that every possible execution of the implementation is also a trace of the model.

The differences between Scar and Ethereum/Solidity are mainly in the area of data types. Scar supports fewer types than Solidity. Therefore, the Solidity implementation has to be limited to the variables that actually occur in the model to ensure consistency.

However, the Scar types contain mathematical integers, which are not present in Solidity; the Scar type system is not a subset of the Solidity type system. Therefore, care has to be taken to avoid situations where this discrepancy leads to a violation of consistency. To give an example, this could happen if a precondition is not satisfiable over mathematical integers, but satisfiable over Solidity's machine integers. This would lead to a situation where the function could never occur in the semantics of the model, but it could be called successfully in the implementation. Thus, the implementation would allow more behaviors than the model. This needs to be avoided by the translation, or by additional measures or precautions developers have to take.

In case of the Scar to Solidity code and annotation generation, the implementation is consistent under the following assumptions:

```
application: bank

contract bank:

  invariant: total >= 0
  invariant: total = \sum(\values(m))

  state:
    var total: uint
    var balances: mapping(account=>uint)

  init: total == 0
  init: \size(balances) == 0

  functions:
    fun deposit:
      post: balances[\caller]
            == \old(balances[\caller]) + \amt
      post: \send(\caller, \this, \amt)
      modifies: balances[\caller]
    fun withdraw:
      params: uint amount
      pre: amount <= balances[\caller]
      post: balances[\caller]
            == \old(balances[\caller]) - \amt
      post: \send(\this, \caller, amount)
      modifies: balances[\caller]
```

Figure 3.10: A simple SCAR application model

- The code generation succeeds
- The generated source code is not changed except for
  - implementing the generated functions, and
  - adding helper functions, *i.e.*, functions that cannot be called from outside their contract, and which do not change the state in any way
- the generated function preconditions and contract invariants do not contain arithmetic overflows
- The implementation is proven correct against the generated specification (function contracts and contract invariants)

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >= 0.7;

library UTIL {
  enum Role {ANY}
  function hasRole(address a, Role r) internal pure returns
     (bool) { }
}

contract bank {
     mapping(address=>uint) balances;
   int total;
   constructor () { }

  /// @notice precondition balances[msg.sender] >= amount
  /// @notice postcondition  balances[msg.sender] ==
     __verifier_uint_old(balances[msg.sender]) - amount
  /// @notice postcondition address(this).balance ==
     __verifier_uint_old(address(this).balance) - amount
  /// @notice postcondition address(msg.sender).balance ==
     __verifier_uint_old(address(msg.sender).balance) +
     amount
  function withdraw(int amount)  public   { }

  /// @notice postcondition balances[msg.sender] ==
     __verifier_uint_old(balances[msg.sender]) + msg.value
  /// @notice postcondition address(this).balance ==
     __verifier_uint_old(address(this).balance) + msg.value
  /// @notice postcondition address(msg.sender).balance ==
     __verifier_uint_old(address(msg.sender).balance) - msg.
     value
  function deposit()  public   { }

}
```

Figure 3.11: The generated Solidity code with SOLC-VERIFY annotations

With this, it is guaranteed that the original model was syntactically correct, and that the application state cannot be changed outside of the behavior specified in the model.

3.7 Applications of Scar

We sketch some applications of the Scar metamodel. While the main use case we envision is verification of application-level properties (Section 3.7), there are other possibilities. Some of them have been implemented in the Scar project and are ready to use, while others remain future work.

Verification of Application-level Properties

The most important motivation for the approach presented here is verification of application-level properties of smart contracts, such as the compliance with access control policies, or with liveness properties. These properties are difficult to specify on the source code level. On the level of a Scar model, however, the relevant specification constructs can be easily introduced in a way that keeps specification concise and intuitive. The respective approaches are described in Chapter 4 and Chapter 5.

In Scar, function contracts and contract invariants have two meanings depending on the perspective: On the source code level, they are proof obligations, and discharging them is showing the consistency between the model and the implementation. On the other hand, they work as assumptions for the application-level properties.

This clear separation between the levels can also help modularizing and continuously refactoring applications and correctness proofs. One example is the case where application-level requirements change. If these requirements are specified directly on the source code level, changes in the requirements are bound to be complex, because it is hard to judge what parts of the application may be affected. However, in Scar, if the security requirements change, the application-level analysis will point the developer precisely to the parts of the implementation that have to be changed to fulfill the new policy.

Scar can also help if an application needs to be adapted to another software version, or to another platform. This can necessitate changes in the source code and/or changes of verification tools in order to prove that the original correctness properties still hold. When maintaining a Scar model of an application, it is easy to understand which parts of an application are affected by a change. Furthermore, writing a new translation from Scar to the language of a ver-

ification tool is less effort than manually writing new specification for every verification tool.

Planned Applications of Scar

Automatic Code Generation While program synthesis from a given functional specification is generally a hard problem, in the smart contract use cases we examined, function contracts are often very straightforward to implement. Indeed, at least in simple cases where the specification can be fulfilled through assignments and simple case distinctions, the question should be asked whether our approach does not put an unnecessary burden on developers by essentially forcing them to write the same thing twice.

Given the recent advances in code completion tools based on generative AI, the Scar approach is a good candidate for AI-based program synthesis. Developers are required to prove the correctness of the implementation against the derived specification anyway, so the use of generative AI for the implementation presents less of a concern. Preliminary experiments we conducted with Github Copilot have been very promising and indicate that after a Scar model has been created, the development of a consistent implementation can be automated to a large degree.

Simulation of Application Execution Formal verification tools have been successfully used to aid developers by enabling simulations (cf., *e.g.*, the simulation capabilities of the Alloy analyzer [Tor+13]). Scar's state transition semantics make it very suitable for execution with random input parameters, and the relevant parts of the application state can easily be highlighted for visualization. This can help developers to quickly build an intuition whether their model actually captures their intention. Furthermore, the small scale hypothesis holds that many errors already manifest after just a few steps of execution. Therefore, simulation could be a lightweight way to catch errors while coming up with a model, before conducting more heavyweight analysis.

Generating Runtime Checks In cases where formal verification of the generated specification does not succeed, Scar can still be used to generate runtime checks. These can then be used as a safeguard to abort execution in case some undesired state occurs, or if function preconditions are not fulfilled on a function call.

The Scar project contains a prototypical runtime check generation, which translates preconditions to `require` checks and postconditions to `assert` statements. This is possible for the large majority of Scar's specification elements

and data types, with the exception of quantified expressions over unbounded data types, which cannot be easily expressed in Solidity without sacrificing runtime performance.

Backwards Projection Consistency between a SCAR model and an implementation is achieved when the implementation can be proven correct against the specification generated from the model. However, what happens when verification fails in this step?

For a basic metamodel, *i.e.*, one without higher-level properties like capabilities or temporal specification, a failure to prove the implementation just means that model and code may not be consistent. However, if the actual target of verification is a higher-level property, *e.g.*, an access control policy, it can happen that the possibility of a violation on the source code does not affect the implementation's conformance with the target property. For example, it is possible that we cannot prove a function postcondition, but the liveness property that we are really interested in does not depend on this postcondition.

To test whether a failure of source code verification has consequences for higher-level properties, we can develop a feedback which projects the output of the verification tool to the model level. In the above case of the superfluous postcondition, we can create a different SCAR model which does not contain the postcondition, and then re-run the model-level analysis.

3.8 Evaluation

In this section, we return to some of the examples described in Section 1.4. We present ways to formalize these use cases in SCAR, and discuss the applicability of the SCAR metamodel.

Bank

The bank example can be modeled in SCAR as an application consisting of one contract with two state variables and two functions (see Figure 3.10). The specification of the functions is straightforward when using the `\send` syntax.

Furthermore, the example includes two contract invariants expressed in the SCAR specification language that may be of interest to the developer. They state that the value of the `total` state variable cannot be negative, and that it is always the sum of all the values in the `balances` mapping.

A feasible functionality for a bank would be to compute some interest for the customers, depending on their balance with the bank. For this, a state variable

```
...
var interest: mapping(address => uint)
function computeInterest:
  post: \forall (a \in \keys(balances)):
          interest[a] = rate * balances[a]
```

Figure 3.12: Banking with interest

`interest` and a function `computeInterest` could be added to the model as in Figure 3.12.

More than the previous implementation of the bank in Figure 3.10, this example shows Scar's power of abstraction. In an implementation, *e.g.*, in Solidity, the `computeInterest` function is rather complex. For one, the universal quantifier has to be replaced by an unbounded loop. Secondly, it is not trivial to iterate over a mapping data structure in Solidity, so the set-valued `keys` abstraction introduced by Scar makes this use case much more succinct than the implementation.

Auction

Modeling the auction in Scar results in the model presented in Figure 3.13. Like the Solidity contract it is modeled after, the Scar application consists of one contract with six state variables and three functions. The function's pre- and postconditions are very similar to the Solidity code in Figure 1.4. On the one hand, this indicates some overhead for the developer, since they will still have to implement the functions; on the other hand, implementation and verification are likely to be very straightforward, and might be automated in the future using program synthesis or automated generation tools.

Escrow

The escrow can be modeled in Scar as seen in Figure 3.14. The resulting contract has four state variables and two functions.

It can be noted here that Scar offers some flexibility as to how some transfers are specified. In the `deposit` function, the precondition states that the caller must be the `buyer` account, and the function call must be accompanied by an amount greater than two times the item price.

It is possible to also specify a postcondition expressing that the caller transfers currency to the contract. Whether this is necessary depends on the use case.

```
application Auction:

enum Mode: {OPEN, CLOSED, FINALIZED}

contract Auction:
state:
  var owner: Account
  var auctionEnd: uint
  var bids: mapping(account => uint)
  var highestBid: uint
  var highestBidder: Account
  var mode: Mode

initparams: uint _duration
init: owner == \caller && mode == OPEN
  && highestBid == 0 && auctionEnd == \systime + _duration

functions:
  fun bid:
    pre: \amt > (highestBid + bids[caller])
         && mode == OPEN
    post: highestBid == \amt + bids[caller]
    post: highestBidder == \caller
    post: bids[caller] == \old(bids[\caller]) + \amt
    modifies: bids[\caller]
  fun withdraw:
    pre: \caller != highestBidder
    post: bids[\caller] == 0
    post: \send(\this, \caller, \old(bids[caller]))
    modifies: bids[\caller]
  fun close:
    pre: \systime >= auctionEnd
    pre: mode == OPEN
    post: mode == CLOSED
    post: \send(\this, owner, highestBid)
    modifies: this.balance
  fun claim:
    pre: mode == CLOSED
    post: mode == FINALIZED
```

Figure 3.13: SCAR Auction

```
application Escrow

enum Mode {AWAIT_PAYMENT, AWAIT_DELIVERY, COMPLETE}

contract Escrow:
  state:
    var mode: currMode
    var buyer: Account
    var seller: Account
    var price: uint

  initparams: account _buyer, account _seller
  init: buyer == _buyer && seller == _seller

  functions:
    function deposit:
      pre: \caller == buyer
        && mode == AWAIT_PAYMENT && \amt >= 2 * price
      post: mode == AWAIT_DELIVERY
    function confirmDelivery:
      pre: \caller == buyer && mode == AWAIT_DELIVERY
      post: \send(\this, seller, price) && mode == COMPLETE
```

Figure 3.14: Scar Escrow

Palinodia

Palinodia (see Section 1.4) is an Ethereum application that allows users to assess the integrity of software binaries downloaded from the internet. Its basic structure is as follows: There are three kinds of smart contract, *Software*, *BinaryHashStorage*, and *IdentityManagement*. A *Software* contract represents one program, for which a set of developers is responsible. The list of developers (represented by their Ethereum accounts) is stored in an *IdentityManagement* contract. This contract, in turn, is a state variable of the *Software* contract.

The software may be published on different platforms. A *BinaryHashStorage* contract represents concrete versions of some software for one specific platform. It is created from a *Software* contract, and pointers are stored in both directions. In it, the responsible maintainers (again represented by an *Identity-Management* contract) can publish and revoke hashes of downloadable binaries.

All contracts have a variable *root owner* of type address. This account is allowed to grant the roles of developers and maintainers, respectively.

The Palinodia paper ([Ste+19]) provides not only the description of this overall structure, but also a detailed description of the state variables of each contract, as well as of the individual functions and their pre- and postcondi-

Variable or Function	Description of Requirements
`root_owner`	Address of root owner
`sw_name`	Name of the software
`dev_control`	*IdentityManagement* contract used for access control
`software_platforms`	The list of different software platforms, each with a pointer to the corresponding hash storage
`storage_contracts`	The list of *BinaryHashStorage* contracts for each software platform
`constructor(_idManagement)`	The constructor is given an *IdentityManagement* contract as a parameter. The caller of the function is the root owner.
`changeRootOwner`	Changes the root owner. Can only be called by current root owner.
`setDevCtrl`	Change the identity management contract. Can only be called by the root owner.
`setSoftwareName`	Change the name of the software. Can only be called by a developer.
`registerBHS`	Register a new *BinaryHashStorage* contract. Can only be called by a developer.
`deregisterBHS`	Delete an existing *BinaryHashStorage* contract from the list of endorsed storage contracts. Can only be called by a developer.

Table 3.4: The requirements of the *Software* contract

tions. For example, the *Software* contract is described as presented in Table 3.4. Similar documents exist for the *IdentityManagement* and *BinaryHashStorage* contracts.

From this information, we construct a Scar model of Palinodia. The part of the model that describes the *Software* contract is presented in Figure 3.15.

The description of the state variables is straightforward. The `platforms` variable is modeled as a mapping, so that each platform name can directly serve as an index into the `storage_contracts` array.

The description of the functions is more complex, especially due to the `platforms_store` and `storage_contracts` variables, both of composite types,

```
application: Palinodia

contract IdentityManagement: // ...
contract BinaryHashStorage: // ...

contract Software:
  state:
    var root_owner: account
    var sw_name: string
    var dev_control: IdentityManagement
    var platforms: mapping(string => uint)
    var storage_contracts: account[]

  functions:
    fun changeRootOwner:
      params: account _newAddress
      pre: \caller == root_owner
      post: root_owner == _newAddress

    fun setDevCtrl:
      params: IdentityManagement _newDev
      pre: \caller == root_owner
      post: developer_control == _newDev

    fun setSoftwareName:
      params: string _newName
      returns: bool
      pre: dev_control.checkIdentity(\caller)
      post: sw_name == _newName

    fun registerBinaryHashStorageContract:
      params: BinaryHashStorage _binaryHashStore
      pre: dev_control.checkIdentity(\caller)
      // equal reference, not equal value
      post: storage_contracts[platforms_store
              [_binaryHashStore.platformID]]
            === _binaryHashStore
      // this contract endorses the bhs contract
      post: storage_contracts[platforms_store[
          _binaryHashStore.platformID]].software_contract
            === \this

    fun deregisterBinaryHashStorageContract:
      params: string _platformID
      pre: developer_control.checkIdentity(\caller)
      post: !(_platformID \in
                \values(storage_contracts[platforms_store]))
```

Figure 3.15: SCAR model of Palinodia Software contract

Example	LoC Solidity	LoC Scar
Bank	11	20
Escrow	24	18
Auction	38	34
Casino	58	53
Palinodia	299	174

Figure 3.16: Lines of code comparison

referencing each other.

Casino

The Casino application (see the Scar model Figure 5.4 in Chapter 5) consists of seven state variables and five functions. However, note that the Scar model contains an error for implementation purposes.

Overall Assessment

While the above set of example models does not constitute a full evaluation, some conclusions still can be drawn.

First of all, Scar is general enough: Its data types and structural model elements are sufficient to model all desired applications and their basic functionality.

Second, even though the set of examples is not large, almost all syntactic elements of the Scar specification language appear in at least one of the examples. From this, it can be concluded that the modeling language is not overly complex.

Another metric for assessing the quality of a metamodel is how succinct the models are, compared to some reference. In Figure 3.16, the length of the Scar models of the examples described in this section is compared to the length of the corresponding Solidity code. This comparison should be taken with a grain of salt; it is easy to change the relative length by leaving out or specifying additional properties in the Scar model.

When comparing the length of the Scar models and the Solidity source code the models are based on, it can be observed that the models are always shorter, but often not by much. For very simple functionality, such as assignments or transfers, the Scar specification and the corresponding Solidity code are practically identical. In such cases, Scar's approach leads to a duplication, and might increase the workload of a developer. On the other hand, functions

with such an easy specification can also be implemented without much effort, and in the future, SCAR will incorporate functionality for automatic program synthesis, or correct-by-construction code generation.

SCAR's benefits become more obvious when the modeled functionality becomes more complex. This includes properties of array and mapping data types, which can be succinctly specified in SCAR. Another recurring case was the use of an array as an index data structure to a mapping, which happens both in the simple bank example and in the real-world Palinodia application. Implementing this functionality is rather complex, requiring tens of lines of code. The abstract property describing the intended behavior, however, is very simple to describe in both cases, and this is reflected in the succinctness of the SCAR model.

Verification with SOLC-VERIFY For the examples discussed above, it was evaluated whether it was possible to prove the Solidity implementation (as presented in the overview in Section 1.4) correct against the generated specification in SOLC-VERIFY. This was possible in all cases. Since neither the implementation nor the specification were very complex, no additional auxiliary specification (such as loop invariants) were required. All proofs were conducted fully automatically.

3.9 Conclusion

This chapter presented the basic components of SCAR, a verification-based model-driven approach for developing smart contract applications. SCAR consists of a type system and a grammar to describe applications in a text-based format. Furthermore, SCAR provides a behavioral specification language to capture the intended functionality of an application's functions.

This basic model enables model-driven development workflows through code and annotation generation. However, the real value of the SCAR application metamodel is as a basis for specifying application-level security policies and temporal properties. This, we cover in the following chapters.

Chapter 4

Capability-based Security for Smart Contract Applications

The smart contract application metamodel presented in Chapter 3 is intended to work as a basis for specifying and verifying application-level properties. In this chapter, we extend the Scar metamodel to introduce a language for defining security properties. The properties defined in this language (see Section 4.2) can be analyzed on the Scar model level. The approach is then integrated into Scar's model-driven development approach by defining a process, based on formal verification, which can be refined to an implementation that conforms to the high-level specification.

We take a view of smart contract security that is based on resources, and who can access them. We identify three main resources: the state of an application, its public functions, and cryptocurrency. Inspired by the capability-based security model [Lev84], we define capabilities as access to resources (calling functions, changing state, and transferring currency) in the context of smart contracts.

Typically, capability-based security is implemented in systems where capabilities can be determined at run-time, *e.g.*, in operating systems. We argue that in the open, highly adversarial environment of smart contract platforms, all relevant capabilities for an application should be determined at design time, based on the security policy given in the requirements specification.

As seen in Section 2.2, building security mechanisms in source code is complex and error-prone. Directly implementing a security policy in a smart contract programming language poses a serious risk of exploits. This risk is exac-

erbated by the fact that smart contracts cannot easily be patched. Therefore, it is crucial to enable developers to work on a suitable level of abstraction for specifying security requirements. The model-driven SCAR approach provides this abstraction, as well as a process resulting in a consistent implementation.

After designing an application by instantiating the SCAR metamodel (see Section 3.3) and specifying the application's security properties, a developer must have a way to decide whether their model is *consistent* and *precise*. Therefore, we develop a formal definition of consistency and precision based on set-theoretic semantics of the extended metamodel. Then, we develop analysis techniques which enable a developer to detect whether their model contains contradictions, or whether the specified capabilities can be more restrictive.

Given a consistent model of an application, the next task is to develop an implementation that is correct w.r.t. the model. For this, we provide an approach for implementing an application for the Ethereum platform written in Solidity. Our approach is based on a combination of code generation and formal analysis. Since our metamodel is platform-independent, the approach can be used for other platforms in principle, but needs to be adapted accordingly.

In summary, the contribution in this chapter is three-fold:

- The SCAR metamodel for smart contract applications is extended with a capability-based security model (Section 4.1 and Section 4.2).
- A definition of consistency and precision of a given model is developed, along with analysis techniques for proving consistency and precision (Section 4.3).
- A partly automatic process for developing a platform-specific implementation which is secure in the sense of the model (Section 4.4).

Overall, this work should enable smart contract developers to turn the security policy of a requirements specification into an implementation which respects this security policy at all times.

4.1 Modeling Capabilities for Smart Contract Applications

This section describes the extension of the SCAR metamodel with roles, and a simple grammar for expressing capabilities.

To recap from Section 3.3, the set of contracts of an application a is denominated $\mathcal{C}_a$. The set $Locs$ of locations in an application is defined by all the state

variables of all contracts (see Table 3.2). Intuitively, a location is either a variable or an element of a composite data type, *i.e.*, an array element, a value in a mapping, or a struct field.

The set of all functions of an application is called $\mathcal{F}$. In this chapter, the SCAR function description is extended with capabilities.

Actors

Going forward, the SCAR metamodel is extended with elements to describe and summarize the entities in a smart contract application which possess agency, *i.e.*, which can call functions. We call such an entity an actor. The set of all actors is denominated *Acts*.

In smart contract platforms, the entities that can call functions are usually called accounts. They are uniquely identified by an *accountID*, *e.g.*, the address on the Ethereum platform. Whether an account represents a program or a person is not relevant for our purposes.

The set of all accounts is called $\mathcal{A}$ (cf. Section 3.3). To make modeling accounts feasible, we allow summarizing them in the form of roles. In our metamodel, a set of roles can be attached to an application. Here, we do not go into the details of role-based access control (some approaches have been proposed by Chatterjee, Pitroda, and Parmar [CPP20] and Töberg et al. [Töb+22]). We simply assume that the implementation of the application contains a boolean function *hasRole(Account a, Role r)* which returns true iff Account a indeed has role r.

In order to allow functions which are not access controlled at all, we define a role `any` which exists by default in all applications. If a function is supposed to be callable by every account without restriction, this role can be assigned the corresponding capability.

The set of roles of an application is $\mathcal{R}$. Each $r \in \mathcal{R}$ defines a set $A_r \subseteq \mathcal{A} := \{a \in \mathcal{A} \mid hasRole(a, r)\}$.

Apart from accounts, functions themselves can also call other functions. Therefore, we include them in the set of actors $Acts := \mathcal{A} \cup \mathcal{F}$.

Resources and Capabilities

Motivated by our perspective of smart contracts described in Section 1.2, we identify three important resources on smart contract platforms: The state, which represents the assets on the ledger; functions, which manage and grant access to the state; and cryptocurrency, which has a role similar to actual currency and can be transferred via built-in primitives.

```
CallCapability      ::=  calls funIDList | any | external
funIDList           ::=  funID [, funID]*

ModifyCapability    ::=  modifies Locset
Locset              ::=  LocExpr [, LocExpr]*
LocExpr             ::=  contractName . LocalLocExpr
LocalLocExpr        ::=  primitiveVarName |
                         CompositeVarName
                         [arraySuffix | mapSuffix | structSuffix]?
arraySuffix         ::=  [intExpr] | [intExpr .. intExpr] | [*]
mapSuffix           ::=  [mapKey] | [*]
structSuffix        ::=  . structMember | .*

TransferCapability  ::=  transfers transfer [, transfer]*
transfer            ::=  ( actorSet , actorSet , limitExpr )
actorSet            ::=  \self | any | accountID | roleID
```

Figure 4.1: Capabilities Syntax

From this, we identify three types of capabilities: Changing state, calling functions, and transferring currency. In our model, capabilities are assigned to actors, *i.e.*, to roles and functions.

Unlike postconditions in ScarML function contracts, capabilities do not specify that some behavior *must* occur. Rather, they describe an upper bound of the allowed behavior. In the context of the frame conditions of function contracts, these two notions coincide.

In this section, we introduce a grammar for defining capabilities. It is shown in Figure 4.1.

Calling functions Roles and functions can be annotated with a list of functions they are allowed to call, following the `calls` keyword. There are two special values: `external` for declaring that the function may only make external calls to functions that are not known at design time, and `any` for stating that a function may call any other function, including external functions.

Modifying State A capability to modify the application state is described after the `modifies` keyword, which is followed by a list of location sets. These include variables of primitive type, but also composite variables and their elements, *e.g.*, individual struct fields and array or mapping elements. Further-

more, array ranges can be specified, and the developer can also express that all elements of a composite type variable can be changed (but not the reference to the variable itself).

Note that for functions, the capability to modify the state coincides with the frame conditions introduced in Section 3.3.

Transferring Currency The `transfers` keyword initializes a transfer capability, which consists of three parts: A set of allowed senders, a set of allowed recipients, and an expression that limits the amount of currency being sent. The sender can be a specific account, but also the special value `\self`, describing the caller of the function. The same is true for the recipient. This makes it possible to specify a recurring pattern of smart contract applications: A caller may initiate a transfer to themselves (*e.g.*, by withdrawing money they deposited earlier), but not to anyone else. Furthermore, the sets of allowed senders and recipients can be described by a role. Finally, the `any` keyword expresses that there is no limitation as to the sender or recipient of the money.

The limit is specified as a general integer expression. It can be an integer literal, but it can also contain arithmetic or make reference to the state of the application, or to the value of function parameters at call-time. This enables the developer to specify limits that depend on the current state, *e.g.*, limiting a withdrawal to the deposited amount.

Note that this is not the same as the `send` primitive in the SCAR functional specification language, which specifies a transaction exactly.

Example We provide a simple running example in Figure 4.2 to illustrate the presented concepts. Our example application is the bank, already introduced in the previous chapters, where customers can deposit cryptocurrency and withdraw it at a later point. Furthermore, the owner of the bank can close it, thereby sending all withdrawable funds to the customers.

The application consists of only one contract, which has two state variables (the `balances` mapping storing the customer balances, and `totBal` storing the overall balance of the contract) and three functions (`close`, `deposit`, and `withdraw`). Furthermore, the application has two roles: An owner, and customers. A suitable implementation of the `hasRole()` function for this application is as follows: An account has the `owner` role if it is the one which created the contract. An account a has role customer if `balances(a)` is greater than zero.

Roles and customers are annotated with capabilities, according to the grammar defined in Figure 4.1. Since access to the `deposit` function is not meant to be limited, it is assigned to the default `any` role. The `owner` may call the

```
application: bank
roles:
  role any:
    calls: Bank.deposit
  role owner:
    calls: Bank.close
    transfers: (Bank, customer, Bank.totBal)
    modifies: Bank.balances[*], Bank.totBal
  role customer:
    calls: Bank.withdraw
    transfers: (Bank, \self, Bank.balances[\self])
    modifies: Bank.balances[\self], Bank.totBal

contract Bank:
  state:
    var balances: mapping(account => int)
    var totBal: int
  functions:
    fun close:
      transfers: (Bank, customer, Bank.totBal)
      modifies: Bank.balances[*], Bank.totBal
    fun deposit:
      params: int amt
      modifies: Bank.balances[\caller], Bank.totBal
    fun withdraw:
      params: int amt
      transfers: (Bank, \caller, Bank.balances[\caller])
      modifies: Bank.balances[\caller], Bank.totBal
```

Figure 4.2: Simple bank application with capability specification

`close` function and transfer currency to other accounts if they have the customer role. They may also modify all elements of the `balances` mapping as well as the `totBal` value. Customers, in turn, can call the `withdraw` function and access the value of the `balances` mapping, but only at their own address. They can also only initiate transfers to themselves, capped at their current balance.

The application's functions are also given capabilities: `close` may transfer money to all customers, limited only by the contract's total balance, and it may modify all state variables of the application. The `withdraw` function can transfer currency, but only to the caller of the function (denoted by the `\self` keyword).

$$\begin{array}{lll} [\![c_1.f_1, ..., c_n.f_n]\!]_C & ::= & \{(get(c_1), get(f_1)), ..., (get(c_n), get(f_n))\} \\ [\![\texttt{external}]\!]_C & ::= & \textit{external} \\ [\![\texttt{any}]\!]_C & ::= & \mathcal{F} \cup \textit{external} \end{array}$$

Figure 4.3: Call Capability Evaluation

4.2 Semantics of Capabilities

In this section, we give a formal meaning to the capability syntax given in Figure 4.1. For each kind of capability, we define an evaluation function which maps syntactical elements to an abstract set of capabilities.

We define an auxiliary function *get:* $ID \Rightarrow \mathcal{CF} \cup \mathcal{V} \cup \mathcal{A}$, which maps function identifiers, variable names, and account identifiers to the corresponding elements in the sets $\mathcal{F}$, $\mathcal{V}$ and $\mathcal{A}$.

Furthermore, some capability descriptions may contain references to the state of an application, or to function parameters. Also, the identity of the caller may be significant to evaluate a capability description. All of these depend on the context of a function call. Therefore, we define another auxiliary function *ctx* which evaluates a given expression in the context of a concrete function call.

Call Capability

Definition 11 (Call Capability Evaluation) *The call capability evaluation*

$$[\![\cdot]\!]_C : \textit{FunctionIdentifiers} \Rightarrow \mathcal{F}$$

is defined by the rules shown in Figure 4.3.

A given list of functions simply evaluates to the set of corresponding functions in $\mathcal{F}$. The `external` keyword evaluates to the corresponding set *external* of all functions which are not described in the application. While nothing is known about the behavior of these functions, it is sensible to make the developer explicitly model external calls. Furthermore, if a function should be unrestricted, the developer can use the `any` keyword to signal this.

Modification Capability

Definition 12 (Modification Capability Evaluation) *The modification capability evaluation*

$$[\![\cdot]\!]_M : \textit{LocSet} \Rightarrow \textit{Locs}$$

$$
\begin{array}{lll}
[\![l_1, l_2, ...]\!]_M & := & [\![l_1]\!]_M \cup [\![l_2]\!]_M \cup ... \\
[\![\mathit{cName}, \mathit{locExpr}]\!]_M & := & \{get(\mathit{cName})\} \times [\![\mathit{locExpr}]\!]_M \\
[\![\mathit{primitiveVarName}]\!]_M & := & \{get(\mathit{primitiveVarName})\} \\
[\![\mathit{ArrVarName}]\!]_M & := & \{get(\mathit{ArrVarName})\} \\
[\![\mathit{ArrVarName}\ \mathit{arrSuffix}]\!]_M & := & \{get(\mathit{ArrVarName})\} \times [\![\mathit{arrSuffix}]\!]_M \\
[\![\mathit{MapVarName}]\!]_M & := & \{get(\mathit{MapVarName})\} \\
[\![\mathit{MapVarName}\ \mathit{mapSuffix}]\!]_M & := & \{get(\mathit{MapVarName})\} \times [\![\mathit{mapSuffix}]\!]_M \\
[\![\mathit{StructVarName}]\!]_M & := & \{get(\mathit{StructVarName})\} \\
[\![\mathit{StructVarName}\ \mathit{structSuffix}]\!]_M & := & \{get(\mathit{StructVarName})\} \times [\![\mathit{structSuffix}]\!]_M \\
[\![\texttt{[}\mathit{intExpr}\texttt{]}]\!]_M & := & \{\mathit{ctx}(\mathit{intExpr})\} \\
[\![\texttt{[}\mathit{e1}..\mathit{e2}\texttt{]}]\!]_M & := & \{i \in Int \mid \mathit{ctx}(\mathit{e1}) \leq i \wedge i \leq \mathit{ctx}(\mathit{e2})\} \\
[\![\texttt{[*]}]\!]_M & := & \mathit{Int} \cup \mathit{MapKeySet} \\
[\![\texttt{[.*]}]\!]_M & := & \mathit{StructFieldSet} \\
[\![\mathit{mapKey}]\!]_M & := & \{\mathit{mapKey}\} \\
[\![\texttt{.}\,\mathit{structMember}]\!]_M & := & \{\mathit{structMember}\}
\end{array}
$$

Figure 4.4: Modify Capability Evaluation

is defined by the rules shown in Figure 4.4.

A list of location expressions is evaluated to the disjunction of the location sets described by each expression. Variable names evaluate to the locations of these variables. For arrays, specifications of single indices or index ranges can contain integer expressions. These are evaluated under the context of the function call. For mappings and structs, the `*` operator evaluates to all possible mapping keys or struct fields, respectively.

Transfer Capability

Definition 13 (Transfer Capability Evaluation) *The transfer capability evaluation*

$$[\![\cdot]\!]_T^{ctx} : \mathit{transfer} \Rightarrow (\mathcal{A} \times \mathbb{N})^2$$

is defined by the rules shown in Figure 4.5. It is parameterized with the call context ctx.

Transfer capabilities consist of three parts, with the first two specifying the sets of allowed senders and recipients, respectively. The third parameter sets an upper bound for the amount of transferred currency. For the sender and recipient sets, the `\self` keyword evaluates to the caller of the function (retrieved by

$$
\begin{array}{lcl}
[\![t_1, t_2 ...]\!]^{ctx}_T & := & [\![t_1]\!]^{ctx}_T \cup [\![t_2]\!]^{ctx}_T \cup ... \\
[\![\mathit{from}, \mathit{to}, \mathit{limit}]\!]^{ctx}_T & := & [\![\mathit{from}]\!]^{ctx}_{rec} \times [\![\mathit{to}]\!]^{ctx}_{rec} \times \{n \in \mathbb{N} \mid n \leq [\![\mathit{limit}]\!]^{ctx}\} \\
\\
[\![\texttt{\textbackslash self}]\!]^{ctx}_{rec} & := & \{\mathit{ctx}(\mathit{self})\} \\
[\![\texttt{any}]\!]^{ctx}_{rec} & := & \mathcal{A} \\
[\![\mathit{accountID}]\!]^{ctx}_{rec} & := & \{\mathit{get}(\mathit{accountID})\} \\
[\![\mathit{roleID}]\!]^{ctx}_{rec} & := & \{a \in \mathcal{A} \mid \mathit{ctx}(\mathit{hasRole}(a, \mathit{get}(\mathit{roleID})))\}
\end{array}
$$

Figure 4.5: Transfer Capability Evaluation

the *get* helper function). A specific account identifier evaluates to that account, and `any` evaluates to the set of all accounts. A role identifier evaluates to the set of all accounts who, in the current context, have the indicated role.

4.3 Analyzing Model Consistency

In this section, we define what constitutes a consistent model. Intuitively, a consistent model is one where capabilities cannot be violated. A violation on the model level occurs, *e.g.*, if a function has a less restrictive capability than an account which is allowed to call that function. In this section, we develop an analysis to decide whether a model is consistent. Additionally, we want to ensure that the specified privileges are tight in comparison to the required capabilities.

Due to our semantics definition, the model consistency is a simple subset relationship of the capabilities. Let $\mathcal{R}$ be the set of roles, $\mathcal{F}$ the set of functions and $\mathit{Acts} = \mathcal{R} \cup \mathcal{F}$ the set of all actors defined for an application (as above).

Definition 14 (Model consistency) *A model is consistent iff*

$$\forall a \in \mathit{Acts} : \forall f \in \mathcal{C}^a_{call} \forall ctx : [\![\mathcal{C}(f)]\!]^{ctx}_\gamma \subseteq [\![\mathcal{C}(a)]\!]^{ctx}_\gamma \quad ,$$

where $\mathcal{C}^a_{call}$ denotes the set of functions specified as callable by a role a, $\gamma \in \{C, T, M\}$ is the domain of call, transfer, or modification capabilities, and ctx an arbitrary context for the evaluation of expressions.

Note that the consistency notion is also transitive, in the sense that if a function f_1 transitively calls f_3 via f_2, the specified capabilities of f_1 need to be a superset of the specified capabilities of f_3:

$$[\![\mathcal{C}(f_3)]\!]_\gamma \subseteq [\![\mathcal{C}(f_2)]\!]_\gamma \subseteq [\![\mathcal{C}(f_1)]\!]_\gamma$$

The call capabilities are directly checkable, since they are a set of symbols. In contrast, the modification and transfer capabilities contain symbolic integer

expressions over the set of states and parameters. Therefore, further reasoning is needed to decide whether a set of modification capabilities is more restrictive than another. The same is true for transfer capabilities.

For both kinds of capabilities, we derive proof obligations in the following definitions which are automatically checkable with SMT solvers (*e.g.*, Z3 [dB08]) due to their support of integer theory.

Definition 15 *Given two sets M_1, M_2 of modify capabilities, we say M_1 is a subset of M_2 ($M_1 \subseteq_{modify} M_2$) iff the following formula is valid*

$$\bigwedge_{m_1 \in M_2} \bigvee_{m_2 \in M_2} m_1 \leq m_2 \ ,$$

where $m_1 \sqsubseteq m_2$ is defined as

$$m_1 \sqsubseteq m_2 := \begin{cases} true : \textit{if}\, m_1 = m_2 \\ \forall \underbrace{x_1, \ldots, x_n}_{x_i \in FV(e_1)} : \exists \underbrace{y_1, \ldots, y_n}_{y_i \in FV(e_2)} : \\ \qquad e_1 \geq 0 \wedge e_2 \geq 0 \rightarrow e_1 = e_2 \\ \qquad : \textit{if}\, m_1 = a[e_1] \textit{ and } m_2 = a[e_2] \\ false : \textit{otherwise} \end{cases}$$

The expressions e_1 and e_2 are the integer expressions describing the set of allowed indices.

$FV(e)$ denotes the free variable of an expression e. These arise from the context of a call, *i.e.*, the state and the parameters of the function. Since this context is not known on the model level, we abstract from the free variables in an over-approximated manner. The definition $m_1 \leq m_2$ evaluates to true for symbolically identical capabilities, *e.g.*, if both capabilities allow the modification of the same state variables.

For accesses of array indices, *e.g.*, $m[4 * x] \leq m[2 * x]$, we need to compare the described set of indices for all possible positive indices: every index of the more or equally restrictive expression must be possible in the other expression. In the example the expression $4 * x$ is more restrictive than $2 * x$ as it describes fewer indices, but not vice versa.

For transfer capabilities, the consistency definition is as follows:

Definition 16 *For transfer capabilities $t_1 = (rec_1, am_1)$ and $t_2 = (rec_2, am_2)$, we say t_1 is more restrictive than t_2 (denoted $t_1 \leq t_2$) iff*

- *receiver set rec_1 is a subset of rec_2, and*

- *the value of am_1 is lower than am_2 under all interpretations of the free variables in am_1 and am_2:*

$$\forall \underbrace{x_1, \ldots, x_n}_{x_i \in FV(am_1) \cup FV(am_2)} : \quad rec_1 \subseteq rec_2 \wedge am_1 \leq am_2$$

For a two sets T_1, T_2 of transfer capabilities, we say T_1 is a subset of T_2 iff

$$\forall t_1 \in T_1 : \exists t_2 \in T_2 : t_1 \leq t_2 \ .$$

The current consistency definitions only specify that the permissions of an actor must not be stricter than those of the functions they are allowed to invoke. Moreover, we might want to ensure that an actor only has the necessary the capabilities to work properly. This *principle of least privilege* is similar to consistency but investigates the opposite direction. For an actor $a \in$ *Acts*, we say it fulfills its principle of least privilege iff

$$\forall c \in [\![\mathcal{C}(a)]\!]_\gamma : \exists f \in \mathcal{C}^a_{call} : c \in [\![\mathcal{C}(f)]\!]_\gamma$$

is valid. The definition requires that every capability of the actor a is required for at least one specified callable function. As this notion also relies on the model-level specification of the callable functions, it might also be an over-approximation in contrast to the actual called function of the implementation.

4.4 A Secure Solidity Implementation

While Section 4.3 describes how to achieve a consistent model of a smart contract application with a formally specified security policy, this section gives a definition of what properties an implementation needs to fulfill to be considered conforming. Based on this definition, we sketch how to arrive at an implementation in the Solidity programming language that is conforming w.r.t. the model according to the definition.

For this, we provide a definition of when an implementation is conforming to its model:

Definition 17 (Implementation conformance) *An implementation is conforming to its model iff*

1. *for each function $f \in \mathcal{F}$, an account $a \in \mathcal{A}$ only has access to f if a has a role $r \in \mathcal{R}$ s.t. $f \in \mathcal{C}^r_{call}$*
2. *each function f conforms to its capability specification, i.e.*
   a) *f calls only functions g where $g \in \mathcal{C}^f_{call}$*

b) *during any execution of* f*, any location* l *where* $l \notin \mathcal{C}^f_{state}$ *remains unchanged*

c) f *only makes transfers* $(to, amt) \in \mathcal{C}^f_{transfer}$

Whether an implementation conforms to its model depends on the platform where it is implemented. Therefore, we sketch a process for the Solidity programming language of the Ethereum platform, although the general ideas might apply for any platform.

Solidity has several characteristics and mechanisms relevant to developing a secure application in the proposed process (cf. Section 1.3). First, in Ethereum, accounts are identified by addresses in the form of 160-bit integers. Second, Solidity provides the `requires` keyword, which checks a boolean condition at runtime and reverts if the condition is not met. Furthermore, Solidity has *modifiers*, which wrap functionality (*e.g.*, parameter checks) and can be added as a keyword to a function header. Both the `requires` mechanism and modifiers can be applied to implement access control.

The basis for generating the proof obligations described in Section 4.3 and the source code stubs as described below is the Scala implementation of the SCAR metamodel (cf. Section 3.5). After arriving at a consistent model, a smart contract application developer can achieve a conforming implementation via a combination of code and annotation generation on the one hand, and formal methods and static analysis on the other.

Code Generation

From the model, the code generator generates source code stubs as described in Section 3.6: First, it generates a `contract` file for each contract in $\mathcal{C}$. The file contains a variable declaration for each state variable, and a function header consisting of name, parameters and return type for every function.

Then, we generate a smart contract which is responsible for access control to functions. There are two possible approaches here, which are applicable depending on whether the access control is static or dynamic. In the case of a static access control, *i.e.*, in a setting where the roles are constant sets of accounts, the generated access control smart contract contains an enum of the roles specified in the model, and a function `hasRole(address a, Role r)`, which can be called to check at runtime whether a given address has a certain role.

For a dynamic setting, the code generation also creates an admin account, which is initialized at the time of contract initialization. Furthermore, two functions `addRole(account a, Role r)` and `removeRole(account a, Role r)` are added, which can only be called by the administrator. These functions add

```
modifier onlyOwner {
  require(hasRole(msg.sender, Roles.OWNER));
  _;
}

function close() onlyOwner {
  ...
}
```

Figure 4.6: Example access control modifier

or remove the given account to the role. While this is not the only possibly implementation of a dynamic role-based access control policy, it is one which fits all of the example use cases, and is also applied in related work (cf. [CPP20]). However, note that in the setting described here, the developer is completely free to implement the access control as desired, and the above is just a quality-of-life feature.

Furthermore, we generate access control modifiers: For each function $f \in \mathcal{F}$, we compute the set $r_f \subseteq \mathcal{R}$ of roles that may access f. Then we generate a modifier which checks whether a given account with address a has any of the roles in r_f. Of course, if the access sets of two functions are equal, the corresponding modifier only needs to be generated once. An example is shown in Figure 4.6, where a modifier is defined to check that the caller has the Owner role, and the `close` function is generated with this modifier in the header.

Furthermore, for each transfer capability $t \in \mathcal{C}^f_{transfer}$ of a function f, we generate a *wrapped transfer function* which ensures at runtime that the function adheres to its capabilities. The function is internal (*i.e.*, it can only be called from within the contract) and takes an address *addr* and an amount *amt* as a parameter. For a capability *t* consisting of a recipient set expression *rs-exp* and a limit expression *limit-expr*, the wrapped function is implemented as follows: At first, it checks whether the address parameter matches the recipient set. This is done by a `require` statement, with a condition depending on *rs-expr*: If *rs-expr* is an address, it must be equal to *addr*. If it is `\self`, *addr* must be equal to `msg.sender` (note that because the wrapped function is marked as `internal`, the `msg.sender` variable is passed on to it from the calling function). If *rs-expr* is a role *r*, it is checked whether `hasRole(addr, r)` returns true. Finally, if *rs-expr* is `any`, the check is omitted. Afterwards, it is checked whether *amt* is less than or equal to *limit-expr*.

We give an example in Figure 4.7. The function `withdraw()` of our running bank example has a capability of sending currency, but only to the caller, and the amount of currency is limited by the balance of the caller. By using

```
function wrappedTfWithdraw(address a, uint amt) internal {
  require(a == msg.sender);
  require(amt <= Bank.balances[msg.sender])
  a.transfer(amt);
}
```

Figure 4.7: Example Wrapped Transfer Function

the generated function `wrappedTfWithdraw`, the developer can be sure that the implementation adheres to the model.

Formal Analysis

While transfers and access of accounts to functions can be handled by code generation in a correct-by-construction manner, this is not possible for ensuring that a function only calls the functions it is allowed to call, and only modifies those parts of the application's state specified in the model. Checking these properties is only possible on the finished implementation. It can be done using static analysis and formal verification tools.

For analyzing whether all functions conform to their call capabilities, we developed a simple static source code analysis based on a publicly available Solidity grammar [Bon+24]. For every function *f*, our analysis collects all function calls that occur explicitly, as well as all occurrences of the `call` and `callcode` keywords and their parameters. It then compares the names of the called functions to the functions in $\mathcal{C}^f_{call}$. If $\mathcal{C}^f_{call}$ contains the *external* value, then only functions within the application itself are forbidden. If the capability specification contains the `any` keyword, the analysis is skipped.

If the analysis finds a function call that is not allowed per capability specification, it fails. This is a deliberate over-approximation, as our analysis will flag some legitimate calls as not allowed (for example, in Solidity, functions can be passed as parameters and assigned to variables and then called). However, it is easy to write the implementation in a way that satisfies the analysis, and we think application developers will benefit from the increased clarity.

For analyzing whether all functions adhere to their modification capabilities, we have to prove that a function modifies only those locations specified in its capabilities. Formally, we need to prove that

$$\forall f \in \mathcal{F} : \ \forall l \in \mathcal{C}^f_{state} : \mathcal{S}^{pre}_f(l) = \mathcal{S}^{post}_f(l)$$

where $\mathcal{S}^{pre}_f$ and $\mathcal{S}^{post}_f$ are the states before and after the execution of f.

```
/// @notice modifies balances[msg.sender]
/// @notice modifies totBal
function withdraw(uint amt) onlyCustomer {
  ...
}
```

Figure 4.8: Example SOLC-VERIFY Frame Condition

For this, we employ automated generation of formal specification in combination with the SOLC-VERIFY formal verification tool [HJ20]. SOLC-VERIFY takes as an input solidity source code that is annotated with formal specification, such as invariants and function contracts. Function contracts consist of a pre- and postcondition, but can also include a *frame condition*, *i.e.*, a statement about what parts of the state a function is allowed to modify.

We utilize SOLC-VERIFY as follows: During code generation (cf. Section 4.4), we annotate every function with one frame condition per *LocExpr* in its modification capability specification. The annotations are in SOLC-VERIFY's annotation language. An example is shown in Figure 4.8: The `withdraw()` function is annotated with frame conditions which state that the function may only modify the caller's own `balances` mapping element and the overall balance of the contract. This corresponds to the modification capabilities defined for the function in the example (Figure 4.2). If the SOLC-VERIFY tool successfully proves that a given implementation adheres to this specification, then it follows that the function adheres to its capability specification.

After the implementation is finished, the developer conducts a formal proof of correctness with SOLC-VERIFY. If the proof succeeds, this means that all functions adhere to their modification capabilities.

Note that, again, this is an over-approximation: It is possible that the proof of correctness of the frame conditions does not succeed although the implementation is actually correct w.r.t. the modification capabilities. This can happen, *e.g.*, if auxiliary specification such as loop invariants are missing or not sufficient.

As an alternative to our over-approximating approach, a developer could also conduct a different static analysis, such as an information flow analysis. SLITHER [FGG19] is one of several tools which have been developed for this purpose. It automatically analyses a given application statically and can be configured to output the call graph and all variables written by a certain entry point (*e.g.*, a function). An alternative to our approach would be to run SLITHER in this way and inspect its output. If the call graph for a function contains functions that are not contained in $\mathcal{C}^f_{call}$, or if a function writes a variable which

is not in $\mathcal{C}^f_{state}$, then the function does not adhere the capabilities specified in the model, and its implementation needs to be corrected.

One drawback of using SLITHER or comparable static analysis tools is the possibility of false positives and false negatives. With our analysis and SOLC-VERIFY's proof of frame conditions, a developer can be sure that an implementation is correct; SLITHER gives no such guarantee. On the other hand, a formal proof of correctness can require developer involvement. For example, it can be necessary to provide auxiliary specification, *e.g.*, loop invariants, for an automated proof to succeed. The decision for a specific analysis tool depends on the specifics of the application (or even individual functions) and has to be made on a case-by-case basis.

Correctness

At the beginning of this section, we gave a definition of when we consider an implementation to be conforming to a model. We argue that the process sketched in this section leads to an implementation which is correct in that sense, if the following assumptions hold (cf. Section 3.6): The developer does not change the generated code, but only adds the function bodies; they only use the auto-generated wrapped transfer methods instead of Solidity's built-in transfer methods; and all source code analyses (cf. above) correctly return a successful result (*e.g.*, a proof that the implementation adheres to the generated frame annotations).

If these assumptions hold, condition 1) is guaranteed by the access control modifiers. Condition 2a) and 2b) are guaranteed by our static call analysis and by SOLC-VERIFY's proof that all the generated frame conditions hold. Finally, condition 2c) holds because the generated transfer functions ensure at runtime that all functions adhere to their specified transfer capabilities.

4.5 Evaluation

In this section, a lightweight evaluation of the capability extension to the SCAR approach is conducted. The main use case considered here is the Palinodia application (cf. Section 3.8), because it is a large application in which access control plays a crucial role.

Bank

The bank example with capabilities (see Figure 4.2) is not shorter than the previous bank examples (see Section 3.8). However, the capability specification

makes a statement which was not possible in the original example, namely that a customer can cause the contract to send money up to their balance. Furthermore, the capability specification also contains statements about who can call what functions. This shows that the extended version of SCAR presented here can specify properties which were not expressable in the basic model.

Palinodia

Palinodia (cf. Section 1.4) is an application that enables users to assess the integrity of software binaries downloaded from the internet. At its core, Palinodia is about access control: Developers and maintainers of software can endorse versions of it by publishing hashes on the Ethereum blockchain. Users who download a software binary can compute a hash of it locally and compare it to a published hash. The trust in the system is established by controlling who can publish (and thereby endorse) these hashes.

Therefore, Palinodia is a good candidate for evaluating SCAR's capability-based security approach. In Section 3.8, Palinodia was discussed for evaluating the general SCAR metamodel and code generation approach; in this section, we analyze whether the capability-based security approach proposed in this chapter is capable of modeling the access control policies that need to be enforced in Palinodia.

Static Application The approach presented in this chapter assumes a static set of contracts. Palinodia as a concept consists of a set of *Software* contracts of unknown size, each of which references a set of *BinaryHashStorage* contracts, the size of which changes over the execution of the application. For this, the capability specification presented here does not work: If a role is assigned a capability, *e.g.*, in the *BinaryHashStorage* contract, it is not clear what this means, since there can be multiple instances of this contract.

In order to make the static approach work, the perspective on what constitutes a Palinodia application needs to be narrowed down. If an application is considered to consist of a concrete *Software* contract for a concrete platform, then it can be described statically, and the role-based capability specification presented in this chapter actually is a very good fit.

In this static description, an application consists of one *Software* contract and one *BinaryHashStorage* contract, as seen in Figure 4.9.

Much of the functionality in the original version of Palinodia is concerned with access control, *e.g.*, adding addresses as trusted developers or maintainers, and checking access control requirements within functions. One of the three contract types in Palinodia (*IdentityManagement*) is exclusively designed for implementing a role-based access control policy.

```
application: Palinodia

roles:
  role any:
    calls BinaryHashStorage.checkHash

  role developer:
    calls setSoftwareName
    modifies sw_name

  role maintainer:
    calls BinaryHashStorage.setPlatformID, BinaryHashStorage
       .publishHash, BinaryHashStorage.revokeHash
    modifies BinaryHashStorage.platformID, BinaryHashStorage
       .hashStore[*], BinaryHashStore.publishCounter

contract BinaryHashStorage:
  state:
    var platformID: string
    var hashStore: uint[]
    var publishCounter: uint

  functions:
    fun setPlatformID:
      params: string _newID
      post: platformID == _newID

    fun publishHash:
      params: uint _hash
      post: hashStore[\old(hashStore.size)] == _hash
      post: publishCounter == \old(publishCounter) + 1

    fun revokeHash:
      params: uint _hash
      post: ! (\exists (i: uint): hashStore[i] == _hash)
      post: publishCounter == \old(publishCounter) - 1

    fun checkHash:
      params: _hash: uint
      returns: bool
      post: \result == \exists (i: uint): hashStore[i] ==
          _hash


contract Software:
  state:
    var sw_name: string

  functions:
    fun setSoftwareName:
      params: string _newName
      post: sw_name == _newName
```

Figure 4.9: Scar model of static Palinodia application with roles and capabilities

All the access control parts of Palinodia can be subsumed in the SCAR description with added role and capability specifications. Therefore, this description is extremely concise compared to both the Palinodia source code and the original SCAR model (cf. the excerpt in Figure 3.15): In its Solidity implementation, Palinodia consists of 299 non-blank, non-comment lines of code. The SCAR model developed in Section 3.8 still has 174 lines of code. The version with capabilities only consists of 37 lines.

Of course, this version still requires a developer to think about the access control smart contract, which is assumed to exist in this chapter. However, the code generation of SCAR turns out to be a very good fit for Palinodia: A developer can automatically generate an access control smart contract where roles can be dynamically added and revoked by an `admin` address (see Section 4.4). The generated contract already contains all defined roles and offers a function `hasRole(address a, Role r)` to check whether a given address has the given role. This is exactly the functionality needed in Palinodia.

Dynamic Application A dynamic instance of Palinodia is one consisting of a set of *Software* contracts, each of which references an unbounded list of platforms. Each software has a set of developers with certain rights, and each platform-specific version of a software has a set of maintainers who have the right to publish valid hashes for the software.

This cannot be modeled in SCAR directly, since roles and capabilities can only be annotated at the application level, not at the level of contracts. In practice, SCAR would still be helpful for this case: Developers could start by modeling a static application and use SCAR's code generation. Then, the generated source code for the different contract types and the respective access control smart contracts would only have to be slightly adapted before deployment of a new *Software* or *BinaryHashStorage* contract. Specifically, the address of the root owner and of the access control smart contract would have to be set manually.

In order to allow for a dynamic set of contracts directly, capability and role definitions and their semantics could be adapted as follows: Roles and capability definitions can either be on the level of an application (as proposed in this chapter), or at the level of a contract type (as introduced in Section 3.3). Contract-level roles are separate for each individual instance of the contract; for the code generation, this also means that for each contract, there needs to be a separate access control smart contract. Contract-level capabilities can only refer to functions and state variables of this contract type, and they, like the roles, are specific to individual instances. As an example, a call capability defined within contract type `c` that allows role `r` to call function `f` would

mean that for every instance c_i of `c`, there is a role r_i which is allowed to call the function $c_i.f$.

In contrast, application-level capabilities would refer to *all* instances of a contract type, and the roles defined for an application would only exist once. In analogy to the above example, if role `r` is defined on the application level and has the capability to call function `c.f`, this means it can call $c_i.f$ for all instances c_i of `c`.

This extension of the capability mechanism would allow the specification of Palinodia in an even more direct manner, since the `developer` and `maintainer` roles could be specified in the *Software* and *BinaryHashStorage* contracts, respectively.

4.6 Conclusion and Future Work

In this chapter, a solution is proposed for developing secure smart contract applications from existing requirements specifications. We argue that implementing a security policy directly in source code is error-prone and likely leading to vulnerabilities. Therefore, we extend the SCAR metamodel of smart contract applications with an attached capability-based security specification. This makes it very easy for smart contract developers to abstract away from the complexity of the source code and focus on the security-relevant aspects of an application.

In the lightweight evaluation we conducted, it was shown that for a realistic application, the approach presented here allows for an extremely succinct specification of security policies in comparison to either the Solidity source code or the basic SCAR metamodel. Furthermore, the separation of concerns achieved by isolating the security policy from the business logics may be beneficial.

Going forward, we define a notion of consistency and precision on our model, based on set-theoretic semantics. We also develop analyses to check whether a given model conforms to these notions. Furthermore, we describe how to turn a model into a conforming implementation. This is done by a combination of code generation and simple static analysis that we implemented on the one hand, and existing formal verification tools on the other hand.

A natural way of extending our methodology is integrating analysis tool results into the model automatically. A further interesting research directory is extending the capability model by adding conditional capabilities, *i.e.*, capabilities that are only in effect if some condition, such as a time constraint, is met.

The SCAR approach is platform-independent, and our methodology can easily be adapted for other platforms or other programming languages. In particular, our approach may be used for programming languages which have built-in

concepts of access control and/or resource management, or tooling which supports these concepts. As an example, it would be possible to derive an implementation in the Vyper language, and ensure conformance to transfer capabilities with support of the 2VYPER verification tool.

So-called permission-based blockchain platforms like Hyperledger Fabric [And+18] do not fully match our notion of smart contracts, since they enforce a closed world in which all participants are known and identifiable. This allows defining role models and security policies which can be implemented above the source code level, partly negating the advantage that our modeling approach brings. However, the methods we developed for analyzing a given model can still be used to detect inconsistency or imprecision in the capability definitions of a Fabric application. This raises the confidence in the security of the application, and may shift the detection of errors from runtime to design time, thereby lowering the cost of fixing them.

Other platforms have yet other built-in mechanisms for access control. Solana smart contracts written in the Rust programming language can require multiple signatures for a program to be executed. In order to work for Solana smart contract applications, the capability-based SCAR approach would need to be adapted in terms of its code generation and source-level verification.

Chapter 5

A Practical Notion of Liveness in Smart Contract Applications

Due to their unique characteristics, it is very important that smart contracts are correct upon deployment. In this chapter, we propose a novel perspective on an important and challenging class of correctness properties, namely *liveness*, in the context of smart contracts. In general, liveness properties can take many forms, depending on the application domain. One example is termination: Given a function, we may ask whether it always finishes execution. In other domains, especially in distributed or parallel settings, deadlock freedom is essential: Is there always a way to continue execution, or is it possible to reach a situation where no progress can be made?

In this chapter, we argue that in the domain of smart contracts, liveness properties typically require that a certain functionality is (or becomes) accessible to an actor. In Section 5.1, we substantiate this intuition by analyzing the examples given in existing literature on smart contract liveness verification. In Section 5.2, we formalize our notion and develop the ScarTL specification language, in which common temporal properties of smart contract applications can be expressed. ScarTL, the Scar temporal logic language, is based on a subset of LTL. We also sketch possibilities for verifying that a given Scar model fulfills its ScarTL specification. In Section 5.4, we demonstrate the use of the language on some examples from literature.

5.1 Liveness Properties in Smart Contracts

Smart contract platforms have several characteristics that influence what kind of liveness properties are important in a smart contract application. First, smart contracts exist in an open world: As a matter of principle, anyone can call any function and thereby trigger a transaction. Furthermore, smart contract platforms specifically exist for use cases where participants in the network do not necessarily trust each other. Therefore, participants generally cannot be assumed to behave in any particular way, at least in the absence of incentives.

A second defining characteristic of smart contracts is money: Most smart contract platforms have some form of cryptocurrency built in, and transferring currency or tokens is a part of almost all real-world smart contract applications. This means that there are usually financial incentives.

These characteristics lead to a special kind of liveness property which is highly common in smart contracts: "If I transfer money to a smart contract, will I get it back?" Or, more generally: Will some desired state change eventually happen? In the following, we elaborate on this kind of liveness property at the hand of the simple examples introduced in Section 1.4, and demonstrate its pervasiveness by a brief review of example liveness properties in the literature.

Simple Bank An example often used to showcase basic functionality is a simple smart contract version of a bank, which allows other accounts to deposit money, logs the balance of each account, and enables withdrawing funds according to the caller's balance (which is stored in a mapping `bals`). There are only two public functions, `deposit` and `withdraw`. They both have no precondition. The postcondition is that money is transferred from the caller to the bank contract (or vice versa for `withdraw`) and that the `bals` mapping is updated accordingly.

Intuitively, the main correctness property of this application can be viewed as a liveness property:

If I deposit money in the bank, I will eventually get it back.

Escrow Another common example is an escrow, where a smart contract application acts as an intermediary for a purchase. For a successful purchase, the application proceeds through a succession of predefined states, according to the actions of buyer and seller. There are several liveness properties integral to the correctness of the application, depending on the exact implementation. For example, after the buyer confirms they received the purchased item, the seller should eventually be refunded their deposit.

Auction In the auction application (see Section 1.4), the owner of a item puts it up for sale. Potential buyers can submit bids. The bids are transferred when making the bid to ensure that the auction owner can get payed eventually. Unsuccessful bidders, *i.e.*, those who are not currently leading the auction, can withdraw the money they transferred so far. After a predefined end date, the auction can be closed, after which no new bets are possible. Then, the auction winner can claim the bought item.

Like in the bank example, the main correctness property of this application is a liveness property:

If an actor makes a bid, that actor will eventually either win the auction and be assigned ownership of the desired item, or they will get their money back.

Casino In the Casino example, the operator commits to a secret number by publishing a hash of it in the smart contract. A player can then bet on the parity of the secret. The main correctness property of this application (as presented in Section 1.4) is also a liveness property:

If the player guesses correctly, they will eventually receive their reward.

Examples from Literature As Chapter 2 shows, formal verification of smart contracts has been a very active field of research. However, only a few tools consider temporal properties, and even fewer target liveness properties.

Sergey, Kumar, and Hobor give some examples of properties over multi-step executions and so-called lifetime properties, and sketch how to prove such properties via an embedding in Coq [SKH18]. VerX [Per+20] introduces Past LTL temporal operators, but focuses on specification and verification of safety properties only.

To our knowledge, there are two tools for specification and verification of liveness properties. Both are specific to the Solidity programming language.

VeriSolid [Mav+19] is a tool for developing Solidity smart contracts through modeling them as state transition systems. The state of an application is modeled explicitly. Transitions are written directly in the supported Solidity subset. This model is translated into a BIP (Behavior, Interaction, Priority) model, which can be model-checked, *e.g.*, for safety and liveness properties like deadlock freedom. VeriSolid allows specification of liveness properties in CTL. However, the properties that can be proven are concerned with successful termination after a function is called. There is no notion of fairness assumptions or the ability of an actor to effect a transfer.

SmartPulse [Ste+21] is a tool for checking safety and liveness properties of a given Solidity smart contract. Properties are specified in SmartLTL, which

contains primitives for functions being called, functions finishing execution, reverting, and sending ether. Fairness assumptions can be specified if necessary to prove liveness properties. In addition to source code and property specifications, an environment is specified, consisting of an attacker model (*e.g.*, bounds on the number of re-entrant calls) and a blockchain model (*e.g.*, gas costs of function calls).

The SmartPulse paper [Ste+21] lists 23 safety and liveness properties of 10 applications that can be verified with their tool. Of these, 13 are liveness properties, signified by the *eventually* keyword. All of them fall into one of two categories: In the first category are properties that represent postconditions of a function, *e.g.*, the following property of an escrow application: *"If a user withdraws funds after refunds are enabled, they will eventually be sent the sum of their deposits."* The paper on VeriSolid [Nel+20] only gives a single example of a liveness property, which also falls into this category.

The second category is of the type described above, stating that some desired action will happen eventually. One of the examples in this category is an auction smart contract exactly like the one described above. Another example is the following statement about a crowdfunding application: *"If the campaign fails, the backers can eventually get their refund."*

In SmartPulse, liveness properties can be specified in a variant of LTL. Properties of the second category require a *fairness assumption* about the actors' behavior in order for verification to succeed: If a withdraw functionality is available, but never called, then losing bidders will not get their money back, although they could! The fairness assumption in this auction scenario is that any losing bidder will eventually call the `withdraw` function.

From Liveness to Enabledness

From the examples above, we note several important points. First, many liveness properties in smart contract application can be reduced to postconditions and termination of a single function. This is already covered by several smart contract verification tools. Therefore, we focus on the second category of liveness property, which states that a desired state change will happen eventually.

Concerning this kind of "real" liveness property, we observe that the crucial point about a desirable state change is whether an actor is able to effect it. There is a subtle difference: Liveness in smart contracts is not about whether something will definitely happen, but about whether someone can make it happen. In the auction example, what we want to prove is not whether every losing bidder actually gets their money back, it is that they *can* get it back (if they take the appropriate action).

This phrasing leads to the insight that in the context of smart contracts, it should be possible to specify liveness properties without having to specify assumptions regarding the actors' behaviors at all. What should be specified is *enabledness*, *i.e.*, the ability to effect a desired result. The simplest cases, where the desired property is just that an event occurs eventually, can be expressed by stating that the event must be permanently enabled. This requires an operator to specify that some property holds forever.

In more complex examples, the ability to trigger an event pertains to a specific actor. Furthermore, on some examples, the desired change can only be effected after some fixed amount of time has elapsed, or if some other condition is met.

Another recurring specification pattern is that some event, *e.g.*, a state change or a function call, is enabled until it happens, but not afterwards. Examples include closing the auction (which should be always enabled after the duration of the auction has elapsed, but which cannot be enabled any more once the auction is closed) and withdrawing money from the bank (depending on the implementation). This resembles the meaning of the *weak until* operator from temporal logic.

One last observation is that liveness is connected to resources, *e.g.*, the built-in cryptocurrency of a blockchain network. Liveness may correspond to ownership: If an actor is able to effect a transfer of an amount of currency from a smart contract to themselves, then they own this amount, even though it is not stored in their own account.

5.2 ScarTL

We formalize the insights from the previous section. As a practical way of specifying liveness properties for smart contract applications, we propose ScarTL, the Scar temporal logic language.

For this, we build on the Scar metamodel as defined in Section 3.3, and extend the ScarML specification language. The extensions consist of special constructs intended to capture the concept of enabledness, and of operators from LTL.

Preliminary: LTL

Linear temporal logic (LTL, first introduced in 1977 by Pnueli [Pnu77]) is a widely used logic for describing and verifying properties of execution traces. In its original form, LTL formulas consist of a set of propositional variables, the

$$
\begin{aligned}
\phi ::= \;& \phi_{\text{ScarML}} \\
& \mid \mathbf{G}\phi \mid \phi_1 \, \mathbf{U_W} \, \phi_2 \\
& \mid f\,\texttt{[}a?\; par?\texttt{]} \\
& \mid \texttt{enabled[}a?\; par?\; amt?\texttt{](}[\phi_1 \mid f]\texttt{)} \\
& \mid \texttt{enabledUntil[}a?\; par?\; amt?\texttt{](}[\phi_1 \mid f]\texttt{)} \\
& \mid \texttt{owns(}a, amt\texttt{)}
\end{aligned}
$$

Figure 5.1: The syntax of ScarTL. ϕ_{ScarML} is a placeholder representing a ScarML specification expression (as defined in Section 3.3). $f \in \mathcal{F}$ is a function of the application, $par \in P_f$ a list of parameters for f, amt an integer expression, and $a, b \in \mathcal{A}$ are accounts. ϕ_1 and ϕ_2 are boolean ScarTL expressions.

standard boolean connectors, and some temporal operators that enable statements about traces:

The *Next* operator $\mathbf{X}\phi$ states that some formula ϕ holds in the next step of the trace. The *Until* operator $\phi \; \mathbf{U} \; \psi$ states that eventually, ψ will hold, and ϕ must hold in every step until that point. $\mathbf{F}$ ("finally" or "eventually", also written $\Diamond$) with the meaning $\mathbf{F}\phi := true \; \mathbf{U} \; \phi$ and $\mathbf{G}$ ("globally", also written $\Box$, with $\mathbf{G}\phi := \neg\mathbf{F}\neg\phi$) are commonly used derivations. Furthermore, the *Weak Until* operator $\phi \; \mathbf{U_W} \; \psi$ states that ϕ must be true until a state is reached where ψ holds, but unlike $\mathbf{U}$, ψ does not necessarily have to become true at some point. The *Releases* operator $\mathbf{R}$ is the dual of $\mathbf{U}$ with $\phi \; \mathbf{R} \; \psi ::= \neg(\neg\phi \; \mathbf{U} \; \neg\psi)$.

Every LTL formula can be transformed into Negation Normal Form (NNF), where the only operators are $\mathbf{U}$, $\mathbf{R}$, and $\mathbf{X}$, and where only atomic formulas are negated [Zhu+17].

ScarTL Syntax

ScarTL is an extension of ScarML with temporal operators and some special constructs for expressing enabledness and the related concepts. The syntax of ScarTL language is defined in Figure 5.1.

In addition to all ScarML expressions, ScarTL consists of the LTL operators $\mathbf{U_W}$ and $\mathbf{G}$. Furthermore, there is a syntax construct for expressing that a certain function is called in one state. The keywords `enabled` and `enabledUntil` formalize the concept of enabledness. The `owns` keyword relates liveness properties with resource ownership.

Note that for the `enabled` and `enabledUntil` expressions, the account, the

parameter list, and the transferred amount are optional, and the argument can be either a transaction or a boolean formula. For the function call expression, parameter list and account are also optional.

Semantics of ScarTL

To recap, for a given smart contract application a, $\mathcal{F}$ is the set of all functions of all contracts in an a, $\mathcal{V}$ the set of all state variables, and $Vals$ the set of all possible values of the variables. Then the state $\mathcal{S} : \mathcal{V} \to Vals$ is a function which assigns each state variable a value.

For a function $f \in \mathcal{F}$, P_f is the set of all possible concrete parameter lists for f. Furthermore, we say that $\mathcal{A}$ is the set of all accounts. $pre_f : \mathcal{S} \times \mathcal{A} \times P_f \to \mathbb{B}$ is the precondition of f, a predicate over the application state, the caller, and the parameters. Likewise, $post_f : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \times P_f \to \mathbb{B}$ is a predicate over the state before and after the execution of a function, as well as the caller and the parameters of the call.

We view an execution of a smart contract application as a trace, as described in Section 3.4. Each step τ_i of a trace τ consists of the application state s_i as well as the transaction which led to τ_i and the time of the transaction t_i. The transaction description consists of the name of the called function, which also contains information about the caller, the sent amount, and the parameters with which it was called (f_i, $caller_i$, amt and $params_i$, respectively).

The semantics of the temporal $\mathbf{U_W}$ and $\mathbf{G}$ operators is identical to the standard LTL semantics.

We define enabledness as a specification construct that is evaluated in one step of an execution. For a function f (with precondition pre_f over the state and the execution context) and an account a, we define that *enabled[a, par, amt](f)* is true in a step τ_i iff a can call f successfully with parameters par and amount amt in the state represented by that step:

$$\tau, i \models \textit{enabled[a, par, amt](f)} \text{ iff } pre_f(s_i, (a, par, amt))$$

The context, or parts of it, can be left out to indicate universal quantification, *i.e.*, enabledness for all callers regardless of the parameters and the amount transferred with the call:

$$\tau, i \models \textit{enabled[](f)} \text{ iff } \forall a \in \mathcal{A}\, \forall par \in P_f : pre_f(s_i, (a, par, amt))$$

We extend this notion to boolean expressions over the state: For a boolean expression c, we say that c is enabled in step τ_i iff there exists a function that is

enabled, and which results in a state that implies the desired state:

$$\begin{aligned}&\tau, i \models \textit{enabled[a, par,amt](c)} \text{ iff} \\ &\quad \exists f \in \mathcal{F} : \tau, i \models \textit{enabled[a, par, amt](f)} \\ &\quad \wedge\, post_f(s_{i-1}, s_i, (a, par, amt)) \rightarrow c\end{aligned}$$

Note that this allows two-state predicates, *i.e.*, boolean expressions which relate two states of the application by expressing a condition over the new state in terms of the previous state. Since function postconditions can also reference the state before the function was executed, the semantics are exactly the same as for enabledness of one-state predicates.

Since liveness properties in smart contract applications are often about resource ownership, we introduce a special construct to express that an account a owns some amount of currency. Recalling the ScarML `\send` keyword, we formalize ownership as the ability of an account to effect a transfer of currency to itself from the contract where the property is specified (`this`):

$$\tau, i \models \textit{owns(a, amt)} \text{ iff } \tau, i \models \mathbf{G}\ (\textit{enabled[a](\textbackslash send(this, a, amt))})$$

This property only makes sense when the `amt` expression refers to a variable which stores the amount, and which is updated in case of a transfer. One example is the mapping storing the balances in the Bank contract. We think this pattern is prevalent enough to justify the `owns` abbreviation.

Furthermore, we introduce a way to express that a function call happened in a given step τ_i:

$$\tau, i \models \textit{f[ctx]} \text{ iff } f = f_i \wedge ctx = ctx_i$$

As with *enabled[]()*, the calling account and the parameters can be left out: *tx[]* is true in τ_i iff `tx` = f_i. This is useful for example when specifying that a certain condition always holds after some function was called.

Lastly, we provide a construct which describes that something (*e.g.*, a transaction or state change) is always possible at least until it actually happens:

$$\tau, i \models \textit{enabledUntil[ctx](f)} \text{ iff } \tau, i \models \textit{enabled[ctx](f)}\, \mathbf{U_W}\, \textit{f[ctx]}$$

As above, the calling account and the parameters can be left out to indicate universal quantification, and the construct can also be used with a predicate instead of a transaction.

5.3 Verification

While the main focus of this chapter is on specification, this section will also discuss the model-level verification of temporal properties. Scar is intended as a platform which enables different tools and techniques for verification both on the model and on the source code level. For application-level temporal properties, this section introduces two verification methodologies: First, a translation to SMT-LIB proof obligations, and second, a translation to the SMV format, and subsequent verification with the nuXmv tool.

In general, for each property specified in an application model, the verification goal is to show that all possible executions of this application fulfill the property. For real liveness properties, as expressed by the **F** operator, this requires additional assumptions, *e.g.*, about the behavior of the actors in an application. Because we restrict our specification language to safety formulas, we can, in principle, prove them by induction.

SMT Translation In this section, we discuss a translation of a Scar model with temporal properties to the SMT-LIB [BST+10] language, which is then input to an SMT solver. The approach is incomplete in that it works only for invariants that are implied by the postcondition of every function in the application. However, the approach is also very lightweight in terms of implementation and runtime, and it is already sufficient to prove most of the properties in the above examples.

The Scar types are translated as follows: `int` and `uint` are translated to `Int`. For constants of unsigned integer type, an assertion is generated which states that the constant cannot be less than zero. The Scar types `bool` and `string` get translated to their direct counterparts. SMT-LIB also has an array type. Scar mappings are translated to uninterpreted functions which take a parameter of the mapping's key type and return the mapping's value type. Enums, structs, and contracts are translated with SMT-LIB's `define-datatype` command.

For each state variable `v`, two constants `v` and `v'` are introduced. If the property to be proven contains an `old` expression nested within a temporal operator, a third constant `v''` is defined. Furthermore, one constant is defined for each of the context variables (`caller`, `amt`, `systime`, `blocknum`), and for each function parameter in the application.

As for the specification expressions, the translation of the arithmetic and logic operators is straightforward. Currently, there is no translation of the set-valued operators, since SMT-LIB does not include set theory; however, there is a definition of the theory of finite sets for the CVC5 SMT solver [CVC24]. In

the future, the Scar to SMT translation will be extended, and the performance of CVC5 evaluated.

With this, the pre- and postcondition condition of each function f_i are translated as predicates $pre(f_i)$ and $post(f_i)$ in SMT-LIB. This, in turn, makes it possible to also translate the temporal constructs presented in this chapter to SMT-LIB:

- A transaction expression *f [ctx]* is translated by the postcondition of f with the values from ctx substituted, if applicable
- An expression *enabled[ctx](f)*, where ctx is the combination of conditions on the context, is translated to the implication $ctx \to pre(f)$
- An expression *enabled[ctx](c)* with c some predicate over the state, is translated as follows:

  $$ctx \wedge \bigvee_{f_i} (pre(f_i) \wedge (post(f_i) \to c))$$

  This represents the existence of a function with the desired postcondition (cf. Section 5.2). Each occurrence of a state variable in $post(f_i)$ and c is replaced by its primed version, except when it occurs in an `old` expression.
- A box expression $\mathbf{G}\ (p)$ is translated as follows: first, the initial state of the application is translated to a predicate $init$, and it is asserted that this predicate evaluates to true. Second, a new predicate $p()$ is defined, with one parameter for each state variable referenced by the expression.

  Then, the entire expression is translated as follows:

  $$(init \to p) \wedge (\bigvee_{f_i} post(f_i) \Rightarrow p(v1', v2', ...))$$

  That is, the initial condition must already fulfill the property, and the property must be implied by every function postcondition.

The top-level property is asserted in its negated form. When a solver is run on the translation and proves it is unsatisfiable, it is thereby shown that the target property is indeed valid.

In Figure 5.2 and Figure 5.3, a simple application with an invariant is shown alongside with the corresponding SMT-LIB translation. The translation is shown to be unsatisfiable, showing that the Scar model's invariant holds.

```
application Example

tmp: G ( ex.a > 0 )

contract ex:
  state:
    var a: int

  init: a == 1

  functions:
    fun f1:
      post: a == \old(a) + 1

    fun f2:
      post: a == \old(a) * \old(a)
```

Figure 5.2: Scar simple invariant example

```
(declare-const ex.a Int)
(declare-const ex.a_ Int)

(define-fun init() Bool (= ex.a 1))
(assert init)

(define-fun p ((x Int)) Bool (> x 0))

(define-fun post-f1 () Bool (= ex.a_ (+ ex.a 1)))
(define-fun post-f2 () Bool (= ex.a_ (* ex.a ex.a)))

(define-fun initiallyValid () Bool (=> init (p ex.a)))

(define-fun ind () Bool (=> (or post-f1 post-f2) (p ex.a_)))

(assert (not (and initiallyValid ind)))
(check-sat)
```

Figure 5.3: SMT-LIB translation for simple invariant example

nuXmv Translation The translation to SMT-LIB discussed above is limited to very simple properties. Therefore, this section discusses another verification methodology, namely the nuXmv model checker [Cav+14].

The nuXmv input language allows specifying a state transition system (STS) that is similar to the Scar semantics. On this STS, properties like invariants or LTL formulas can be specified, and a range of technologies to verify whether a system fulfills these properties.

The translation of the data types is more limited than in SMT-LIB. All types are bounded in size by a variable `bitlen` that has to be provided by the prover. Then, the Scar integer types are translated to the nuXmv `signed word` and `unsigned word` types, respectively. The primitive `bool` type and the enumeration types can be translated directly to their nuXmv counterparts. Arrays and mappings are translated to the nuXmv type `array word` of the specified bit length.

The nuXmv state transition machine is defined in terms of input variables (`IVAR`), state variables (`VAR`), an initial condition (`INIT`), invariant constraints (`INVAR`), and a transition function (`TRANS`). The Scar state variables are translated to state variables. Function parameters and context variables are translated as input variables. Furthermore, an input variable `fun` of enumeration type is created, with one value for every function in the application.

Contract invariants are translated as invariant constraints, *i.e.*, they are assumed to be true, which limits the search space. The initial condition is translated directly.

The `TRANS` constraint defines whether a pair of states is a valid transition. It can contain variables in `next` expressions, signifying that they are evaluated in the second state. Let pre_{f_i} and $post_{f_i}$ be the translation of the pre- and postcondition of f_i, respectively. In the postcondition, variables are translated as `next()` expressions. The transition relation is then defined as follows:

$$\bigwedge_{f_i}(fun = f_i \ \wedge \ pre_{f_i} \rightarrow post_{f_i})$$

The temporal specification is straightforward to translate, since nuXmv supports LTL (among many other temporal logics). `enabled` expressions (along with the constraints on the call context) are translated in the same way as in the SMT-LIB translation, *i.e.*, a disjunction over the application's functions and their effects.

nuXmv only supports a limited type system, and it does not support quantifiers in the translation of the function pre- and postconditions and contract invariants. On the other hand, the state transition machine is semantically close to Scar, which makes the translation straightforward. Concerning verification, nuXmv is more complete for our purposes, since it also takes into account the

function preconditions, and it is able to prove a valid invariant even if it is not inductive by inferring an inductive invariant.

5.4 Evaluation

We describe how to specify the liveness properties of the examples discussed in Section 5.1. We also sketch how the properties can be verified, and discuss the limitations and advantages of our approach in general.

Bank

The main correctness property of the simple bank application is that every customer can withdraw all their funds whenever they want. The customer balances are stored in a key-value mapping bals.

```
\forall (a: account): G enabled{a}
    (a.balance == \old(a.balance) + \old(bals[a]))
```

Here, we can also specify where the money comes from, and use the `\send` shorthand:

```
\forall (a: account): G enabled{a}
    (\send(\this, a, bals[a]))
```

In this specific case, we can even use the owns abbreviation and simply write

```
\forall (a: account): owns(a, bals[a])
```

Verification is straightforward: The `withdraw` function does not have a precondition. It is therefore always enabled for every caller, and the postcondition matches the desired property exactly.

Escrow

In the escrow example, one liveness property is that the seller can get their deposit back after the buyer confirms the reception of the item. In Scar, we model this property as follows:

```
confirmPurchase{buyer} =>
    enabledUntil{seller}(\send(\this, seller, deposit))
```

This means that after the `confirmPurchase` method is called successfully, the seller is able to effect a refund of their deposit – of course, only until this actually happens.

This can be verified by showing that the only function that is enabled after the successful call to `confirmPurchase` is `refundSeller`, and that its postcondition implies the desired effect.

Auction

For the auction example, we consider two properties: Bidders must be refunded if they do not win, and the seller should be able to claim the winning bid after the auction closes.

For the losing bidders, the property is similar to that of the bank, with the difference being that the current highest bidder can not withdraw:

```
\forall (a: account):
  G(a==highestBidder||enabled{a}(\send(\this, a, bals[a])))
```

Note that in this example, we cannot use the `owns` shortcut, because it includes a **G** operator, so that the resulting property might actually not be true: After all, a losing bidder might increase their bid to become the highest bidder again.

Verification is also similar to the bank example. The `withdraw` function has only one precondition, which is that the caller must not be the current highest bidder. Therefore, it is always enabled for all other accounts. From this, it follows that the desired property is indeed an invariant.

For the seller in the auction, the desired property is that after the auction is closed, they get paid. This can be specified in two steps. First, after the auction ends, it can be closed:

```
G (time > endTime => G (enabledUntil(close)))
```

Second, after the auction is closed, the seller can call the `claim` function to be paid the auction price from the contract:

```
G (close => enabledUntil{seller}(claim))
```

Another correct formalization of this second property can be expressed with the application state instead of a transaction expression:

```
G (state == State.CLOSED => enabledUntil{seller}(claim))
```

Yet another possible formalization could express the ability to effect a transfer, instead of the enabled-ness of the claim function. In this example, there are many different reasonable ways of specifying the desired property.

Verification relies on the fact that after the `close()` function is successfully executed, the `state` variable is in the state `closed` (as implied by the postcondition). This means that the `claim()` function is enabled for the seller. Since no other function is enabled, we derive that the enabled-ness of the `claim()` function is an invariant until it is actually called.

At first glance, it seems that the seller property it would be easier to specify with the LTL **F** operator, like this:

```
F (enabled{seller}(claim))
```

This property might hold, but additional assumptions about the seller's behavior would have to be given in order to be able to prove this.

Casino The Casino example (see Section 1.4), if modeled as in Figure 5.4, contains a vulnerability: If the player guessed correctly, but the operator never calls the `decideBet` function, the player will not receive their reward. This violates the application's main correctness property, which states that if a player guesses the parity correctly, they must be able to withdraw the money they bet plus the same amount as a reward. In SCAR, this property can be formalized as follows:

```
\forall (secret: uint):
    placeBet{a,_guess,amt}
  && _guess == secret % 2
  && hashedNumber == \hash(secret)
  => enabledUntil{\caller==a}(transfer(\this,a,2*amt))
```

Since there is no function that the player can call to collect their reward, the verification fails. However, the error can be corrected; one way of doing so is adding a `timeout` variable, and a function that can be called by the player after the timeout to withdraw the reward irrespective of the game's outcome. In this formalization, the above property (adapted to include the timeout) is maintained and can be proven correct.

General Remarks

Our specification language can be used to express all properties yielded by our literature research on smart contract liveness properties. This shows that the introduce specification constructs combined with the $\mathbf{G}$ and $\mathbf{U_W}$ operators are sufficient to specify properties which are commonly perceived to be liveness properties.

In SCARTL, it is not possible to specify that a strategy exists for achieving some goal, *i.e.*, a desired state change. This means that the specification must be explicit about how a desirable state can be reached: *e.g.*, in the auction, the specification cannot just state $\Diamond$*enabled[](close)*, but has to show the way: `time > end` $\rightarrow$ *enabledUntil[](close)*. This not only simplifies verification, but also forces clarity in the specification. If a complex sequence of function calls is necessary to reach some goal, this might point to an overly complex implementation and possible simplification. At the very least, our approach will force the developer to document the necessary steps.

There are plausible scenarios where our notion of liveness fails to express all relevant properties. One example would be a vote with a quorum: Some desirable action will be taken according to a vote, but only after a fixed percentage

```
application Casino

enum Mode {IDLE, GAME_AVAILABLE, BET_PLACED}

contract Casino:
  state:
    var operator: account
    var player: account
    var pot: uint
    var bet: uint
    var hashedNumber: uint
    var guess: uint
    var mode: Mode

  init: operator == \caller
        && mode == IDLE
        && pot == 0 && bet == 0

  functions:
    fun addToPot: ...
    fun removeFromPot: ...

    fun createGame:
      params: uint _hashedNumber
      pre: mode == IDLE
      pre: \caller == operator
      post: hashedNumber == _hashedNumber
      post: mode == GAME_AVAILABLE

    fun placeBet:
      params: uint _guess
      pre: _guess == 0 || _guess == 1
      pre: \caller != operator
      pre: \amt <= pot
      post: mode == BET_PLACED
      post: \caller == player
      post: bet == \amt
      post: guess == _guess

    fun decideBet:
      params: uint secretNumber
      pre: mode == BET_PLACED
      pre: \caller == operator
      pre: hashedNumber == \hash(secretNumber)
      post: secretNumber % 2 == guess
            => pot == \old(pot) - bet
               && \send(\this, player, 2 * bet)
               && bet == 0
      post: secretNumber % 2 != guess
            => pot == \old(pot) * 2
               && bet == 0
                     post: mode == IDLE
```

Figure 5.4: Scar model of the Casino application

of those entitled to vote have cast their vote. Will the action be taken eventually? Whether or not the participants are incentivized to vote depends on the specifics of the application. If they are sufficiently incentivized, this would constitute a case where a fairness condition makes sense, and our simpler notion would not be sufficient to specify and verify that any action will be taken. However, cases like this do not seem to be common in the smart contract world, and deciding whether a fairness assumption is plausible can be very challenging. We leave this kind of question to future research.

Our model-driven approach for specification and verification enables developers to specify liveness properties on a level where the implementation of the functions is abstracted via function contracts. Therefore, we cannot rely on the implementation itself for verification. Working on the abstraction means that, in general, the properties that can be proven in our approach are a subset of the properties that would be provable directly on the implementation. However, since verification in our approach is straightforward for all example liveness properties we could find in the literature, we argue that this limitation hopefully does not matter much in practice.

5.5 Conclusion and Future Work

In this chapter, we analyze the concept of liveness properties for smart contract applications. We find that all properties commonly perceived as liveness in the literature are not classical liveness, but can be expressed as an actor's access to some functionality. Based on this finding, we develop ScarTL, and extension to ScarML. ScarTL provides a limited set of temporal operators as well as concise constructs for specifying enabledness. The constructs offer fine-grained control for specifying the context in which an event is supposed top be enabled. Furthermore, we sketch how this perspective simplifies the verification task, and evaluate our approach on some typical examples.

In the future, we will look to further automate the verification task. Furthermore, we will develop processes for different platforms to achieve implementations which adhere to the liveness properties specified in the model. Conversely, it would be possible to translate an annotated Solidity smart contract implementation to a Scar model, and use our approach to specify and verify liveness properties on it.

Chapter 6

Specification of Frame Conditions for Solidity and Hyperledger Fabric

The previous chapters have presented a platform-independent model-driven approach for developing secure smart contract applications. However, the SCAR approach relies on platform-specific formal methods for creating a correct implementation of a model. In this chapter, we present two specification languages for frame conditions – one for Solidity and one for Hyperledger Fabric smart contracts written in Java.

6.1 Introduction

Smart contracts are a prime target for formal methods because they manage resources, bugs cannot be fixed easily, and they are relatively small. Many formal analysis methods require the contract's program code to be annotated with formal specifications (invariants, pre- and postconditions). These may express the top-level functional requirements for the contract that are to be analyzed, verified, or checked at runtime. Or they are auxiliary specifications of components to facilitate a modular analysis.

In this chapter, we propose an approach and a formalism to enrich specifications with *frame conditions*, *i.e.*, a specification of what a function *cannot* resp. *will not* do. Even though this information can also be given as part of a postcondition, frame conditions focus on a particular kind of information, namely which parts of the storage is changed – as opposed to specifying how it is changed.

Note that it is usually much easier to specify which parts of the state a function (potentially) changes (its frame) than specifying what is not changed. The changed part is closely tied to its functionality and in most cases is much smaller than the unchanged part.

Simple, intuitive frame annotations can help programmers express their expectations as to the effect of a function, and a proof that a function actually adheres to its frame – whether successful or not – will give valuable feedback concerning the correctness of an implementation.

The idea of dynamic frames, which we use in our approach, aims to solve the frame problem by making location sets first-class citizens in the specification language, so that a programmer can refer to such sets directly. This allows proofs that two such sets are disjoint, or that one location is not part of a given set of locations. A dynamic frame is an abstract set of locations; it is "dynamic" in the sense that the set of locations to which it evaluates can change during program execution.

Smart contract networks can be categorized according to the general availability of their services: In a public platform, everyone can set up a node to replicate the ledger and validate the correctness of all transactions. And there are no limitations as to who may make a transaction. In contrast, private networks are set up to serve a group of stakeholders; setting up a node requires permission of some sort, and access to smart contract functions is regulated. This also implies the necessity of an identity management. For our approach to smart contract frame specifications, we select representatives of both categories: Ethereum [Woo14] as the most relevant public blockchain platform, and Hyperledger Fabric [And+18] as a permission-based approach that targets industrial applicability. In Sections Section 6.3 and Section 6.4, we discuss the relevant properties of Ethereum resp. Hyperledger Fabric, define languages for specifying frame conditions that fit these properties, respectively, define their semantics, and show some examples.

Even though our main focus is on the specification language, we give some hints as to how the frame specifications can be translated into assertion statements and can thus be analyzed and verified (Section 6.5).

6.2 Motivation

Smart contracts are computer programs, written in turing-complete, high-level programming languages, which offer services to clients in distributed, decentralized networks. The services are made available through public function calls. The main benefit of smart contracts is that they ensure deterministic and reproducible execution despite the inherently decentralized architecture.

Though there is no universally accepted definition for what constitutes a smart contract, the general consensus is that they run on a blockchain infrastructure, *i.e.*, they work in conjunction with a distributed, immutable ledger; they can take control over assets on that ledger; and they do so in an automated and deterministic fashion, thereby enabling parties who do not necessarily trust each other to rely on them. Depending on the platform, a smart contract can be deployed either by every network participant, or by the network administration. Interaction with a smart contract is done by calling its (public) functions. Reading from and writing to the distributed ledger is possible only through smart contracts.

Since their inception, smart contracts have been a prime target for research in the area of formal methods, for two main reasons: First, the main use cases of smart contracts, *i.e.*, managing resources in networks in which participants do not trust each other, means that programming errors can have severe consequences. Furthermore, once discovered, bugs in the code cannot be fixed as easily as with other programs, since smart contracts are either immutable once deployed, or require the explicit consent of all concerned parties for every change to the code. This makes static proofs of correctness before deployment highly desirable. Second, the characteristics of smart contracts make them rewarding targets for formal specification and verification: smart contracts are usually short, do not have many dependencies on outside libraries, and tend to refrain from using complex program structures (to the point where unbounded loops are considered bad practice [Con20]). This makes functional verification of single smart contract functions feasible [APS18]. However, verifying the correctness of a smart contract application, *i.e.*, a set of functions operating on the same part of the state, is still challenging. As others have noted [Per+20], smart contract applications should be viewed to have an implicit enclosing loop, within which functions and parameters are non-deterministically chosen (while smart contract architectures eventually create a shared order of transactions, it is not possible to statically predict that order). This encourages a modular approach to formal verification in the style of design-by-contract, where function calls are abstracted through pre- and postconditions.

Even if verifying the functional correctness of a single smart contract function may be within reach due to the lack of complex control structures and data types, two difficulties remain: (a) reasoning about calls from one smart contract function to other functions, (b) proving that an invariant of a smart contract application is maintained by all its functions. We argue that these difficulties should be approached by modularization of the verification effort: Smart contract functions should be annotated with pre- and postconditions. Correctness properties of a smart contract application are formulated as an invariant. A proof of correctness for a function must then include a proof that it preserves

the invariant. However, proofs of this kind often require some sort of reasoning about a frame condition, *i.e.*, a specification of what a function *cannot* resp. *will not* do. As a very simple motivating example, consider a smart contract with only two functions, one called `pay` that accepts a positive amount of currency, and another one called `lookup` which allows anyone to observe the accumulated amount. A plausible invariant states that the accumulated amount never decreases. However, that the invariant is preserved by `lookup` may not immediately follow from `lookup`'s postcondition, which typically does not mention any changes to the contract's balance. This can be solved by a frame annotation which states that `lookup` does not change the state at all.

6.3 Frame Conditions for Solidity

We recall the relevant features of the Solidity programming language (cf. also Section 1.3), and discuss syntax and semantics of the proposed specification language.

Relevant Features of Solidity

Ethereum is a distributed computing platform for smart contracts, which are executed on the Ethereum Virtual Machine (EVM). There are several higher-level languages that can be compiled into EVM bytecode; the most popular representative of these languages is Solidity. Execution of bytecode instructions on the EVM consumes a resource called *gas*. The caller of a smart contract function has to provide sufficient gas. If the execution costs exceed the provided amount of gas, the execution is aborted and has no effect.

In Ethereum, there are two types of accounts: External accounts, which only have the functionality to send and receive money, and contract accounts, which can have arbitrary functions. Every account has a balance in the built-in *Ether* cryptocurrency.

Each account in Ethereum has an address in the form of a 160-bit integer. For contract accounts, this address also serves as a namespace where the contract's state is stored.

In the Solidity programming language, the balance of a contract with address `addr` is obtained with `addr.balance`. The address of the contract to which a function belongs is accessed with `address(this)`; the address of the caller of a function is accessed with the Ethereum built-in `msg.sender` construct.

Solidity differentiates between *memory* and *storage*. Memory is not persistent between function calls; it is used, *e.g.*, for function parameters and tempo-

rary variables. Storage refers to the data persisted on the blockchain. In our approach for frame specifications, frames only contain locations in the persistent memory. In which way a function may or may not have affected the volatile memory is irrelevant after its termination anyway. In the following, we therefore always refer to `storage`.

Solidity has two kinds of variables types: *value types* of a fixed size, like booleans, integers, fixed-size arrays etc., and *reference types*, including dynamically sized arrays, mappings, and structs. Variables of reference type point to a location (in the namespace of their contract).

Syntax and Semantics of Frame Conditions for Solidity

We define a specification language for frame analysis of Solidity smart contracts. The basic building blocks are frame conditions, which can be attached to a function and define its frame. A frame condition starts with the keyword `modifies`, which is followed by one or more location expressions or the `nothing` keyword. A location expression is a combination of an address expression and a variable expression. We also call these frame conditions *modifies clauses*.

As for addresses, `msg.sender` and `this` are special address expressions. Furthermore, expressions of `int`, `contract` and `bytes20` types can be cast to addresses.

As for variable expressions, simple variables are referred to by their name. In addition, for array, struct, and mapping expressions we allow suffixes that denote which part of the data structure may be modified: `arr` would express that the pointer to the array can be modified, while `arr[4]` refers to the fifth entry in the array to which `arr` points. `arr[0..4]` allows the modification of the first through fifth entry of the array. Finally, `arr[*]` includes all elements of the array. Similar short-hand constructs exist for structs and mappings.

The full syntax for our Solidity frame-condition language is given in Table 6.1. There,

- *addrExpr* is any Solidity expression of type address. This includes address literals as well as compositional expressions like `this.x` where `x` is a variable of type address, such that `this.x.y` becomes a valid location expression.
- *primitiveTypeVarName* is any name of a variable of primitive static type (boolean, integer, address).
- *fixedArrVarName* is any name of a variable of a fixed-sized (static) array type.

Table 6.1: Syntax of the Solidity frame condition language

`modifies` *locExpr*+ | `nothing`

locExpr	::=	*addrExpr* . *loc-identifier*
loc-identifier	::=	*primitiveTypeVarName* | *fixedArrVarName arrSuffix* | *refToArrExpr* | *refToArrExpr arrSuffix* | *refToMapExpr* | *refToMapExpr mapSuffix* | *refToStructExpr* | *refToStructExpr structSuffix*
arrSuffix	::=	`[`*intExpr*`]` | `[`*intExpr* .. *intExpr*`]` | `[*]`
mapSuffix	::=	`[`*mapKey*`]` | `[*]`
structSuffix	::=	. *struct-member* | .*

- *refToArrExpr*, *refToMapExpr*, *refToStructExpr* are any Solidity expressions that evaluate to a reference of the appropriate (dynamic) type.

Note that there is a difference between dynamic and static arrays (fixed-sized byte arrays): If `arr` is dynamic, then '`modifies arr`' and '`modifies arr[i]`' are both valid modifies clauses, while only the latter is allowed for static arrays. The reason is that the reference to a static array cannot be changed. If a user were to write '`modifies arr`' for a static array, then either that clause would be redundant or it would have meant to be '`modifies arr[*]`' – both an indication of some misconception on the user's part.

To formalize the semantics of modifies clauses, we first define the set $Locs$ of *locations* and, based on that, the concept of *state*.

Definition 18 *The set of Solidity locations is*

$$\begin{aligned} Locs \;=\; \mathbb{N}_{160} \times (&\mathit{primitiveTypeVarName} \cup \\ &(\mathit{fixedArrVarName} \times int) \cup \\ &ptr \cup \\ &(ptr \times (\mathit{arrayIndices} \times \mathit{mapKeySet} \times \mathit{structMemberSet}))) \end{aligned}$$

where

- $\mathbb{N}_{160}$ *is the set of 160-bit numbers (addresses),*
- *primitiveTypeVarName is the set of all names of variables of primitive type,*
- ptr *is the set of all references to storage,*
- *arrayIndices* $= int$ *is the set of possible array indices (integers), mapKeySet is the set of all possible map keys (all primitive types), and* $structMemberSet$ *contains the names of all struct members.*

A state *is a function*

$$state : Locs \rightarrow Vals$$

that assigns values to locations, where the set of possible values $Vals$ *contains all primitive types as well as the elements of* ptr*, i.e., the references to storage.*

Instead of the set *Locs* from Definition 18, we could alternatively have used $\mathbb{N}_{160} \times ptr$ as the set of locations, staying closer to the Ethereum semantics. With our more structured definition, however, we encode in the structure of locations that certain location expressions cannot alias to the same position in storage while others can.

Consider for example the location expression `this.arr[4]`. It represents the location $(17, (25, 4))$ if `this` has the address 17 and `a` is a variable of dynamic array type that refers to an array at position 25 in the storage of `this`. Depending on the type of `a`, the value of that location can be a primitive value, say 42, but it can also be a pointer to, *e.g.*, a particular struct in storage if `arr` is an array of structs. We can immediately conclude – without further analysis of the state – that the location expression `this.a[5]` evaluates to $(17, (25, 5))$ and is, therefore, a different location (no aliasing). The location expression `this.b[4]`, on the other hand, may or may not evaluate to the same location as `this.a[4]`, depending on the values of `a` and `b` in the particular state (aliasing possible). Similarly, if `x` and `y` are variables of static primitive type, then `this.x` and `this.y` evaluate to $(17, \mathtt{x})$ resp. $(17, \mathtt{y})$ and are, thus, different independently of the state. If, on the other hand, `x` and `y` are variables of dynamic type, their locations are of the form $(17, 25)$ (the second component is a pointer), and they may be the same.

The full semantics of the location expression lists which occur in `modifies` clauses is defined by the function $[\![\cdot]\!]^s$:

Definition 19 *Given a state* s*, the evaluation*

$$[\![\cdot]\!]^s : \textit{LocationExpressions} \rightarrow Locs$$

of lists of location expressions is defined by the rules shown in Table 6.2.

In these rules, $[\![e]\!]^s_{Eth}$ *denotes Ethereum's evaluation function that evaluates an expression* e *in a state* s *according to Ethereum's semantics.*

Note that, in the above definition, $[\![\cdot]\!]^s$ evaluates to a set of locations, while $[\![\cdot]\!]^s_{Eth}$ returns a single value (a primitive value, and address, or a storage pointer). The function $[\![\cdot]\!]^s_{Eth}$ is also responsible for giving values for the special expressions `this`, `msg.sender`, `msg.value` etc., which we evaluate according to Ethereum's semantics.

Table 6.2: Rules for the evaluation function $[\![\cdot]\!]^s$ (see Definition 19)

If Exc is any expression that throws an exception:		
$[\![Exc]\!]^s$	$:=$	$\emptyset$
Otherwise:		
$[\![\texttt{nothing}]\!]^s$	$:=$	$\emptyset$
$[\![locExpr_1, locExpr_2]\!]^s$	$:=$	$[\![locExpr_1]\!]^s \cup [\![locExpr_2]\!]^s$
$[\![\mathit{addrExpr}.\,\mathit{loc\text{-}identifier}]\!]^s$	$:=$	$([\![\mathit{addrExpr}]\!]^s_{Eth}, [\![\mathit{loc\text{-}identifier}]\!]^s)$
$[\![\mathit{primitiveTypeVarName}]\!]^s$	$:=$	$\{\,\mathit{primitiveTypeVarName}\,\}$
$[\![\mathit{fixedArrVarName}\ \mathit{arrSuffix}]\!]^s$	$:=$	$\{\,\mathit{fixedArrVarName}\,\} \times [\![\mathit{arrSuffix}]\!]^s$
$[\![\mathit{refToArrExpr}]\!]^s$	$:=$	$\{\,[\![\mathit{refToArrExpr}]\!]^s_{Eth}\,\}$
$[\![\mathit{refToArrExpr}\ \mathit{arrSuffix}]\!]^s$	$:=$	$\{\,[\![\mathit{refToArrExpr}]\!]^s_{Eth}\,\} \times [\![\mathit{arrSuffix}]\!]^s$
$[\![\mathit{refToMapExpr}]\!]^s$	$:=$	$\{\,[\![\mathit{refToMapExpr}]\!]^s_{Eth}\,\}$
$[\![\mathit{refToMapExpr}\ \mathit{mapSuffix}]\!]^s$	$:=$	$\{\,[\![\mathit{refToMapExpr}]\!]^s_{Eth}\,\} \times [\![\mathit{mapSuffix}]\!]^s$
$[\![\mathit{refToStructExpr}]\!]^s$	$:=$	$\{\,[\![\mathit{refToStructExpr}]\!]^s_{Eth}\,\}$
$[\![\mathit{refToStructExpr}\ \mathit{structSuffix}]\!]^s$	$:=$	$\{\,[\![\mathit{refToStructExpr}]\!]^s_{Eth}\,\} \times [\![\mathit{structSuffix}]\!]^s$
$[\![\texttt{[}\mathit{intExpr}\texttt{]}]\!]^s$	$:=$	$\{\,[\![\mathit{intExpr}]\!]^s_{Eth}\,\}$
$[\![\texttt{[}\mathit{intExpr}_1\texttt{..}\mathit{intExpr}_2\texttt{]}]\!]^s$	$:=$	$\{\,i \in int \mid [\![\mathit{intExpr}_1)]\!]^s_{Eth} \leq i \wedge i \leq [\![\mathit{intExpr}_2]\!]^s_{Eth}\,\}$
$[\![\texttt{[*]}]\!]^s$	$:=$	$int \cup \mathit{mapKeySet}$
$[\![\texttt{.*}]\!]^s$	$:=$	$\mathit{structMemberSet}$
$[\![\mathit{mapKey}]\!]^s$	$:=$	$\{\,\mathit{mapKey}\,\}$

Finally, we can define what it means for a function be correct w.r.t. a modifies clause:

Definition 20 *A Solidity smart contract function f is correct w.r.t. a modifies clause '*`modifies` *$locExpr$' iff the following holds:*

For all $preState, postState \in states$ such that f terminates in $postState$ when started in $preState$,

$$\forall\, l \in \mathbb{L} : preState(l) \neq postState(l) \implies l \in [\![locExpr]\!]^s$$

Note that the correctness condition of the above definition is equivalent to

$$\{l \mid preState(l) \neq postState(l)\} \subseteq [\![locExpr]\!]^s$$

According to the first line in Table 6.2, if a location expression throws an exception, such as division by zero or out-of-bounds array access, its semantics is the empty set. This is useful in the context of Ethereum, because if such an exception occurs, the function is rolled back and has no effect. In this case, no location is modified, and the modifies clause is trivially satisfied.

Modifies clauses are always evaluated in the prestate of a function execution, not in the state in which an assignment happens during the execution. For example, if the modifies clause contains `this.a[i]` and `this.i`, then the assignments ‘`a[i] = a[i]+1; i = i+1;`’ are fine, but ‘`i = i+1; a[i] = a[i]+1;`’ violates the clause. The latter would need `this.a[i+1]` to be included.

We do not allow pointer arithmetic in modifies clauses, *i.e.*, `(a+`n`)[1]` cannot be used to refer to `a[2]` for any n. Modifies clauses with pointer arithmetic would be confusing, a source of errors, and a way of obfuscating the clauses for malicious programmers. Moreover, pointer arithmetic is notoriously hard to analyze for verification tools.

Example: A Simple Solidity Bank Contract

The example in Figure 6.1 shows a simple version of a Solidity contract for a bank, where clients can deposit and withdraw their money. When someone deposits their funds, the money is transferred to the bank contract, and their balance with the bank is adapted (in the `bals` mapping).

The modifies clauses of `deposit` and `withdraw` include `bals[msg.sender]` to indicate that one – and only one – entry of the `bals` mapping is changed; the other entries of the mapping remain unchanged, which is important for checking the ‘`this.balance == \sum(bals)`’ (see Section 6.5).

The modifies clauses also mention the contract’s as well as the caller’s balance (`this.balance` and `msg.sender.balance`), which are modified because the function either accepts funds (because it is `payable`) or transfers funds to the caller.

A modifies clause has also been added to function `lookup` to specify that *only* the deposit and withdraw methods can change the bookkeeping balance, and `lookup` modifies `nothing`.

Example: Uninitialized Pointers in Ethereum

The example in Figure 6.2 illustrates one of the pitfalls of the solidity programming language (or, more precisely, a recent version of Ethereum).

The `Surprise` contract declares a struct `Thing`, a public integer variable `x` and a mapping `things`. The function `addThing` declares a variable `t` of the `Thing` type, but does not initialize it (which is problematic); afterwards, the struct’s boolean field is set to false, and the new thing is added to the contract’s `things` mapping. Unintuitively, calling this function overwrites `x`: Since `t` is an uninitialized storage pointer, it automatically points to slot 0 of the contract’s

```
contract Bank {
 //@ invariant this.balance == \sum(bals);
  mapping (address => uint) private bals;

 //@ postcondition
 //@  this.balance == \old(this.balance) + amt
 //@  && \bals[msg.sender] == \old(bals[msg.sender]) + amt);
 //@
 //@ modifies bals[msg.sender];
 //@ modifies this.balance, msg.sender.balance;
  function deposit() public payable returns (uint) {
    bals[msg.sender] += msg.value;
    return bals[msg.sender];
  }

 //@ postcondition bals[msg.sender] >= amt
 //@  ==> (this.balance == \old(this.balance) - amt
 //@  && \bals[msg.sender] == \old(bals[msg.sender]) - amt);
 //@ postcondition bals[msg.sender] < amt
 //@  ==> (this.balance == \old(this.balance)
 //@  && \bals[msg.sender] == \old(bals[msg.sender]);
 //@
 //@ modifies bals[msg.sender];
 //@ modifies this.balance, msg.sender.balance;
  function withdraw(uint amt) public returns (uint) {
    if (amt <= bals[msg.sender]) {
      bals[msg.sender] -= amt;
      msg.sender.transfer(amt);
    }
    return bals[msg.sender];
  }

 //@ modifies \nothing;
  function lookup() public returns (uint) {
    return bals[msg.sender];
  }
}
```

Figure 6.1: A simple bank contract written in Solidity

```
contract Surprise {

  struct Thing {
    bool b;
  }

  uint public x = 100;
  mapping(string => Thing) public things;

  //@ modifies things[name];
  function addThing(string name) public {
    Thing storage t;
    t.b = false;
    things[name] = t;
  }
}
```

Figure 6.2: An example of unintuitive behavior in Ethereum (cf. [Hit18])

storage, which in fact is also the position of the storage variable `x`. The boolean value `false` is cast to the type of `x`, *i.e.* `uint`, which yields 0. The `addThing` function modifies the value of a variable that it does not syntactically refer to, which is not what a programmer would expect. A framing condition that clearly states which locations a method may modify (along with a tool to prove that the function indeed fulfills the specification) would render errors of this kind harmless (of course, the tool needs to know about the intricacies of the memory model). In the example, the function is annotated with a `modifies` clause stating that only the `things` mapping at the key of the newly created object may be modified. A proof of correctness against this specification would fail, leading the programmer to detect the error. A programmer may then have the (wrong) idea that the problem can be fixed by adding `t` to the modifies clause; but that would lead to a syntax error because the modifies clause is not within the scope of `t`'s declaration.

Beginning with Solidity version 0.5 (Nov. 2018), uninitialized storage pointers lead to a compiler error, because the behavior described above was considered too dangerous and unintuitive. Nevertheless, the example shows that modifies clauses help in cases where a programmer has a misconception of what a function does. Moreover, a programming language does not even have to be unintuitive; mistakes always happen. Framing specifications can help to uncover these mistakes before they cause potentially serious errors.

6.4 Frame Conditions for Hyperledger Fabric

We discuss the relevant features of the Hyperledger Fabric platform, where smart contracts can be written in Java. We then design a specification language for frame conditions specific to the Hyperledger Fabric platform.

Hyperledger Fabric

Hyperledger Fabric emerged as one of the projects from the Linux Foundation's Hyperledger umbrella project. It aims to offer an "operating system for permissioned blockchains" [And+18]. Unlike Ethereum, a Fabric network consists of agents who know each other's identity. Fabric smart contracts are programs written in one of a number of languages, *e.g.*, Go, Java, or Javascript. Function calls are regulated by access control and submitted to a number of nodes. If these nodes compute the same results, the resulting state changes are submitted to an ordering service in the form of a read/write set (*i.e.*, a set of locations and values that are read or written, along with versions in order to detect read-/write conflicts), and finally broadcast to all participating nodes.

Hyperledger Fabric has its own storage nomenclature. The fundamental data structure is a blockchain which stores the assignments made as a result of smart contract function calls. However, the data structure that a smart contract developer interacts with is not the blockchain, but an abstraction called the *world state*. The world state is a database that holds the current values of the ledger state, expressed as key-value pairs. The world state at key `s` is defined as the value of the last assignment to `s` in the blockchain. The *ledger* is the combination of the blockchain and the world state determined by the blockchain.

In the following, we consider the world state that represents the storage of one fabric network. Our frame condition language for Fabric allows for less granularity than the one for Solidity: It does not mention specific data structures (like structs or arrays) nor does it allow field access. This is due to the fact that Fabric chaincode can be written in one of several programming languages with different characteristics and built-in structures. We want our proposed language to capture the process of accessing the world state, which is similar in the different APIs. Our language can then be instantiated (*i.e.*, refined) for the actual chaincode programming languages.

In Fabric, data is (only) stored in the form of byte arrays. Thus, all data structures have to be serialized before they can be written to the ledger, and deserialized after reading. This does not immediately lead to problems because we treat the data stored at one location as a monolithic block, and do not allow expressions accessing sub-units of data structures (like fields of a stored object, or elements of a stored array). This has the drawback that, if more fine-grained

information about which parts of a data structure are changed is required, this has to be achieved through cumbersome auxiliary specifications that refer to serialization and deserialization (see Section 6.4).

In the Java API for Fabric, basic storage access is provided by the `getState`, `putState` and `delState` methods. The ledger object on which these methods operate is passed as a parameter of each chaincode function. Range access is also possible: `getStateByRange(String start, String end)` returns all keys which are lexicographically between the two parameters, and the corresponding values. This is reflected in our specification language. We allow location expressions to end with an asterisk `*` to express that all locations with a given prefix may be modified. Furthermore, we allow expressions that evaluate to sets of strings, such as lists or arrays.

Composite keys are another way of storing and querying data in Fabric. A composite key consists of a list of attributes, and a string which denotes the types of these attributes. For example, an item in an auction (cf. Section 6.4) could be represented as a Java struct with two fields for the ID of the item (`itemID`) and the ID of the owner (`ownerID`). A composite key for one specific item could have the form `(itemID~ownerID, 42, "john")`. This enables queries for either the ID of the item or the owner; in this example, it would allow a function to retrieve all items belonging to a particular ID without having to read all items from the state and filter them by their owner. Since the composite key mechanism is present in the APIs for all chaincode programming languages, it is included in our specification language in the string set expressions.

In Hyperledger Fabric, the caller of a function is obtained via the `CID` (for Client Identification) interface, which guarantees that each agent in the network is identifiable by a unique ID.

Syntax and Semantics of Frame Conditions for Fabric

In Fabric, a location is uniquely described by a string. Our specification language (see Table 6.3) therefore consists of the `modifies` keyword followed by one or more expressions which return strings or sets of strings, such as string ranges (*e.g.*, "`modifies itemA .. itemZ`" would express that all locations whose identifiers are lexically between itemA and itemZ can be modified), prefix expressions using the asterisk (*e.g.*, "`modifies item*`" would include all locations where the key starts with "item"), and side-effect-free expressions that return a string or a collection of strings (such as string arrays in Java or Go).

The full syntax for our Fabric frame-condition language is given in Table 6.3. There,

Table 6.3: Syntax of the Fabric frame condition language

`modifies` *locExpr*+ | `nothing`

locExpr	::=	*strExpr* | *rangeExpr* | *starExpr* | *strSetExpr*
rangeExpr	::=	*strExpr* `..` *strExpr*
starExpr	::=	*strExpr* `*`

- *strExpr* is expression in the programming language that evaluates to a string,
- *strSetExpr* is any expression that evaluates to a collection or set of strings (*e.g.*, string arrays).

To formalize the semantics of modifies clauses, we again have to define a concept of state. The definition is similar to that for Solidity frame conditions (Definition 18), except that now locations are strings, and their values are byte arrays.

Definition 21 *The set of Fabric locations is*

$$Locs \;=\; \mathit{String}$$

where String is the set of all strings. A state *is a function*

$$state : Locs \to Vals$$

that assigns values to locations, where the set of possible values $Vals$ *is the set of all (finite) byte arrays.*

The semantics of location expression lists, which occur in `modifies` clauses is again defined by giving rules for the function $[\![\cdot]\!]^s$:

Definition 22 *Given a state* s*, the evaluation*

$$[\![\cdot]\!]^s : \mathit{LocationExpressions} \to Locs$$

of lists of location expressions is defined by the rules shown in Table 6.4.

In these rules, $[\![e]\!]^s_{\mathit{Fab}}$ *denotes the evaluation function that evaluates an expression* e *in a state* s *according to the semantics of the programming language that is used to write Fabric contracts.*

The definition of when a contract function satisfies its modifies clause is the same as that for Solidity (Definition 20):

$$\forall\, l \in \mathbb{L} : preState(l) \neq postState(l) \implies l \in [\![locExpr]\!]^s$$

Table 6.4: Rules for the evaluation function $[\![\cdot]\!]^s$ (see Definition 22). $\leq_{lex}$ is the lexicographic ordering relation.

If Exc throws an exception or if its evaluation is undefined:

$[\![Exc]\!]^s$	$:=$	$\emptyset$

Otherwise:

$[\![\texttt{nothing}]\!]^s$	$:=$	$\emptyset$
$[\![locExpr_1, locExpr_2]\!]^s$	$:=$	$[\![locExpr_1]\!]^s \cup [\![locExpr_2]\!]^s$
$[\![strExpr]\!]^s$	$:=$	$\{ [\![strExpr]\!]^s_{Fab} \}$
$[\![strSetExpr]\!]^s$	$:=$	$\{ s \in \mathit{String} \mid s \in [\![strSetExpr]\!]^s_{Fab} \}$
$[\![strExpr_1 \,..\, strExpr_2]\!]^s$	$:=$	$\{ s \in \mathit{String} \mid [\![strExpr_1]\!]^s_{Fab} \leq_{lex} s$ and $s \leq_{lex} [\![strExpr_2]\!]^s_{Fab} \}$
$[\![strExpr*]\!]^s$	$:=$	$\{ s \in \mathit{String} \mid strExpr$ is a prefix of $s \}$

Example: A Fabric Auction Contract

In Figure 6.3, excerpts of an auction smart contract are shown. Items that can be bought or sold are represented as structs with fields for their own ID, the ID of their owner, and the ID of the auction in which the item is offered (if any). Besides items, auction objects are stored on the ledger. They declare a list of strings which signify the locations of the items that are sold in the auction. Furthermore, they have an ID, a minimum bid, and an ending time. When an auction is created, it is checked that all items actually belong to the caller of the function, *i.e.*, the identity which creates the auction. If all checks succeed, the items are given a non-empty auction ID, signifying they are currently being auctioned and cannot be offered in another auction.

When an auction is closed, the ownership of all its items is transferred to the highest bidder, and the auction ID of the items is set to the empty string, signifying that they are currently not being auctioned. The `closeAuction()` function can modify the auction object which is referred to in the `auctionID` parameter. Furthermore, it can modify all the items which are being sold in this auction object. However, there is no direct way to refer to this set of items, since their locations are not directly passed as parameters but only indirectly as a field of the auction object. Therefore, to refer to the item list in the `modifies` clause, the object stored at the location of the `auctionID` string needs to be read from the ledger with `getState`, deserialized, and then cast to an object of the auction type: ‘`(Auction) deserialize(ledger.getState(auctionID))`’. The `itemLocs` field of this object yields a list, and therefore constitutes a string set expression as defined by our specification language. If this frame condition is proven to be correct, it can then be used to prove more complex properties of

```
public class Auction extends ChaincodeBase {

  //@ modifies auctionID, itemLocs;
  Response createAuction(...) {
      ...
  }

  //@ modifies
  //@   auctionID,
  //@   ((Auction) deserialize(ledger.getState(auctionID))).
     itemLocs
  Response closeAuction(String auctionID, ChaincodeStub
     ledger) {
    Auction a = deserialize(ledger.getState(auctionID));
    if (getCurrentTime() < a.ending) return newErrorResponse
       ();
    a.closed = true;
    for (String s: a.itemLocs) {
      Item i = deserialize(ledger.getState(s));
      i.owner_id = a.highestBidderID;
      i.auctionID = "";
      ledger.putState(i.itemID, serialize(i));
    }
    ledger.putState(auctionID, serialize(a));
    return newSuccessResponse();
  }
  ...
}
```

Figure 6.3: Fabric chaincode from an auction contract

the auction smart contract, *e.g.*, that items can only ever be modified with the consent of their current owner.

6.5 Towards Analysis and Verification of Frame Conditions

Though the focus of this work is on the specification of frame conditions, we give short overview of how they can be analyzed and verified, and how frame conditions can be used in the analysis and verification of other formal specifications such as invariants and pre-/postconditions.

The most obvious way to express what needs to be proved to establish correctness of a smart contract function `f` w.r.t. its frame condition is to use the

formula from Definition 20 as a postcondition and to prove that it holds after all executions of `f`:

$$\forall\, l \in \mathbb{L} : \mathit{preState}(l) \neq \mathit{postState}(l) \implies l \in [\![\mathit{locExpr}]\!]^s \tag{6.1}$$

That is actually what is typically done in deductive verification tools with an expressive program logic such as the KeY tool [Ahr+16], which supports both automatic and interactive verification of Java code with annotations such as pre- and postconditions and modifies clauses. KeY's support for user interaction allows verification w.r.t. expressive specifications, but proving frame conditions can require a lot of effort for interactive proof construction.

For systems based on software (bounded) model checking or runtime checking, Equation (6.1) is problematic because it quantifies over all locations and, moreover, requires to store the prestate values of locations so that they can be compared to their poststate values. For such systems, it is better to add an assertion for each assignment such that the assertion fails if the assignment writes to a location not mentioned in the frame condition (for simplicity, we only consider assignments but this approach is applicable to other state-changing operations as well). Notice that only assignments to storage locations need to be covered, as assignment to volatile memory is always legal.

Consider, for example, the assignment '`bals[msg.sender] += msg.value ;`' in function `deposit` (Figure 6.1). It leads to the assertion that $[\![\texttt{bals[msg.sender]}]\!]^s$ must be an element of the locations in the modifies clause, where s is the state in which the assignment happens and the modifies clause is evaluated in the prestate of the function. This assertion is true, but note that even in this simple case the two states (s and the prestate) are not the same as the function is `payable`, and the transfer of funds happens before the assignment `deposit` is executed. Still, checking this assertion is easier then proving Equation (6.1), as the assertion does not contain a universal quantifier and the set of locations in the modifies clause is much smaller than the set of all locations mentioned in Equation (6.1). An implementation that generates the appropriate assertions needs to be aware and make use of the evaluation $[\![\cdot]\!]^s$ for modifies clauses (Definition 19).

Note, that an analysis which checks every assignment may actually raise false alarms. According to our definition, an assignment to a storage locations is allowed – even if the location is not mentioned in the modifies clause – if it has no effect (*e.g.*, '`x=x+0`') or is temporary (*e.g.*, '`x=x+1;x=x-1`'). However, such false alarms are rare in practice and mostly indicate redundant code or bad programming style even if they do not actually violate the modifies clause.

Once the correctness of a function w.r.t. a modifies clause has been established, that knowledge can be used in further correctness proofs. For example,

consider the invariant shown at the beginning of contract `bank` (Figure 6.1). To prove that this invariant is preserved by function `deposit`, one can analyze `deposit`'s implementation. But if `deposit` has already been shown to satisfy its specification, the proof can be modularized, *i.e.*, one can prove that the invariant is implied by the specification. That requires to prove that (a) the invariant implies the precondition and that (b) the postcondition implies the invariant. The latter step, however, is only possible using the modifies clause. The postcondition alone is not sufficient as it only expresses what the function does, not what it does not do. The postcondition does not say anything about the elements of `bals`[c] for $c \neq$ `msg.sender` in the poststate. But the modifies clause comes to the rescue: since these locations are not mentioned in the modifies clause they must be unchanged, which implies that their value from the prestate is preserved.

6.6 Related Work

Several approaches to security analysis and formal verification of smart contracts have been proposed. They range from simple static analysis for detecting known anti-patterns (*e.g.*, [Luu+16]), over dynamic approaches (*e.g.*, [EP18]), trace properties [Per+20] and functional specification and verification [Bha+16; Bec+18]) to full formalizations of the Ethereum virtual machine in the Isabelle/HOL framework [Ama+18] (see [Pra+20] for a recent overview).

Frame analysis is an established field of research, and several logic frameworks have been proposed. The two most prominent approaches are *separation logic* [Rey02] and *dynamic frames* [Kas06].

Separation logic is an extension of Hoare Logic which enables reasoning about programs with complex pointer structures. It has been used for verification and program analysis in a variety of tools, such as jStar [DP08], a verification tool for Java programs, or the RustBelt project [Jun+17] for verification of the core of the Rust programming language.

The theory of dynamic frames has been widely used for program verification; for example, the Java Modeling Language [LC06] (and, subsequently, verification tools that build upon it) use a dynamic-frames approach (see [Ahr+16]).

Combinations of both approaches have also been proposed [SJP09]. For example, *Chalice* [PS12] is a language and program verifier for reasoning about concurrent programs with built-in specification constructs written in the style of implicit dynamic frames.

Our proposed specification languages for smart contracts use the dynamic frames approach. This has the advantage that the programmer can write the specification in the same terms that they need to know to write the program

code. There is no additional learning required; the specification language is very similar to the programming language itself. This is a difference to separation logic, which requires more knowledge about the logic behind the specification.

From our experience, this does not make the proofs of correctness more difficult: While separation logic may have an advantage for reasoning about complex pointer data structures, these do not occur in the reality of smart contracts. Therefore, we can use the simple, intuitive specification language of dynamic frames without sacrificing expressive power or efficient proofs.

6.7 Conclusion

In this chapter, we argue that formal specification and verification of smart contracts would benefit from frame conditions. We propose framing specification languages for two smart contract platforms, Ethereum and Hyperledger Fabric.

We plan to implement verification support by translating the proposed modifies clauses into a standard assertion language supported by existing tools. That will allow to automatically generate and then discharge proof obligations from our frame annotations. An implementation that generates the appropriate assertions needs to be aware and make use of the evaluation function $[\![\cdot]\!]^s$ for modifies clauses (Definition 19).

For Fabric specification, it would also be useful to support more complex location expressions. Regular expressions, which many programmers are familiar with, would make a useful addition. This would, however, also bring new challenges for the verification of the resulting proof obligations.

Chapter 7

Conclusion

Smart contracts are programs which give some guarantees even if executed on somebody else's computer. This makes them potential building blocks for decentralized systems. However, the characteristics of smart contract platforms make the applications built on them hard to fix, while at the same time, every bug in the code is a potential security issue. Formal methods are necessary to ensure that smart contract applications behave as intended.

7.1 Summary

In this work, we present a formal approach for the development of correct and secure smart contracts. The SCAR framework consists of a platform-independent metamodel of smart contract applications. On this basis, application-level temporal and security properties can be specified and verified. Furthermore, SCAR provides methods to create source code implementations which fulfill the properties specified on the model level.

SCAR The core of the SCAR approach is a metamodel of smart contract applications. These are described in terms of their contracts, state variables, and functions. The behavior of the functions is defined in terms of function contracts consisting of pre- and postconditions, as well as frame conditions. SCAR provides a formal language in which developers can instantiate the metamodel in a text format.

From a given model, developers can generate source code annotated with formal specification. Making use of existing verification tools, developers prove that their implementation is consistent with the model.

Capability-based Security The basic SCAR metamodel is extended with security properties. These enable developers to specify what actors have access to which resources in an application. Actors can be accounts, but also functions. Furthermore, accounts can be summarized into roles.

For these actors, developers can specify capabilities, which encompass access to functions, state changes, and cryptocurrency transfers.

Actors and capabilities are specified on the model level. Then, a model can be analyzed for consistency, which is violated if actors are able to access functions that have greater capabilities than themselves. If a model is judged to be consistent, a combination of annotation generation and source-level formal methods ensures that the implementation is a refinement of the model and fulfills the same security properties.

The evaluation shows that for an application where access control is an important concern, the SCAR model is much more concise than the Solidity implementation. Furthermore, the SCAR tooling greatly reduces the necessary effort for implementing the access control policy.

Liveness Properties In the existing literature, there are few approaches for verification of liveness properties in smart contracts. After a review of the commonly cited use cases, we observe that in the adversarial environment of smart contract applications, liveness is often better specified as enabledness.

Continuing from this observation, the SCAR framework is extended with a language for specifying temporal properties. The concept of enabledness, along with some other constructs, is reified in the language to make common properties easy to specify.

Furthermore, we sketch two ways for verification of temporal properties in a SCAR model. In SCAR's model-driven approach, verification is done on the model level. In the implementation, the proven properties are guaranteed to hold due to the refinement relationship between model and code.

Figure 7.1 presents a final overview of the SCAR approach, similar to Figure 3.1, but including the extensions for capability-based security and liveness.

Frame Conditions In the last chapter, we develop two platform-specific approaches. Formal verification tools can benefit from the modularity introduced by method contracts, but for this effect, it is necessary to also specify what parts of the state are *not* changed by a function.

To this end, we design two specification languages for frame conditions, building on the theory of dynamic frames. The first is for Solidity and the Ethereum platform, while the second targets Hyperledger Fabric contracts written in Java.

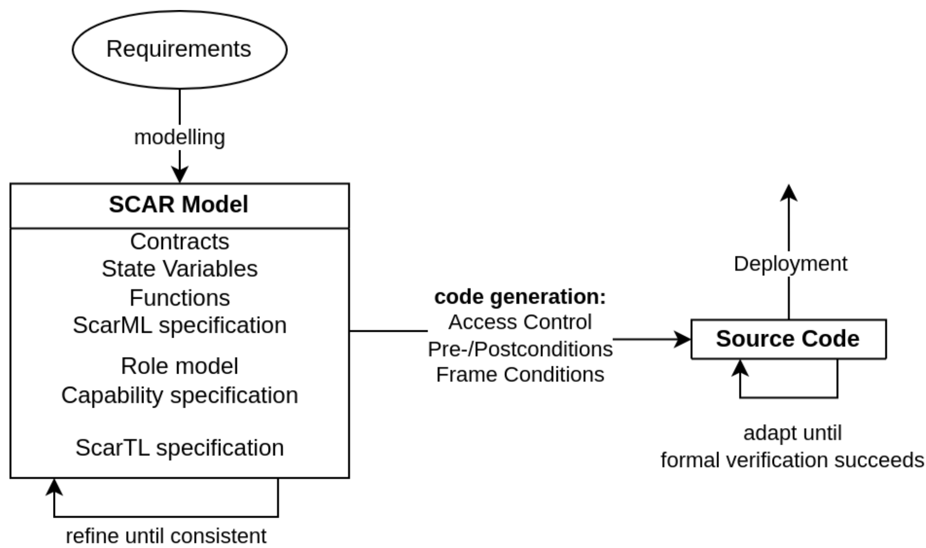


Figure 7.1: The SCAR process of smart contract application development

The languages are intended to be incorporated by developers of formal verification tools. Furthermore, they can serve as inspiration for specification language creators in the smart contract domain.

7.2 Future Work

Composition and Relational Properties SCAR models describe correctness and security properties of smart contract applications. In the future, it would be interesting to investigate whether the properties of different models, or parts thereof, are maintained after (partly) composing the models.

Furthermore, relational properties of SCAR models should be investigated. For example, a notion of equality on SCAR models should be developed. Furthermore, it should be investigated whether the abstraction provided by the SCAR approach can be used to make relational verification more efficient.

Support for more Platforms In order to develop smart contracts with SCAR, it is necessary to have a definition of consistency between the SCAR model and the target implementation language. Currently, most smart contracts are written in Solidity, and therefore, Solidity has been the main target of the SCAR approach. At this time, Solidity is the only language for which the SCAR approach has clearly defined consistency, and SOLC-VERIFY is the only verification tool for which an automated translation has been implemented.

In the future, we want to extend the SCAR approach to other platforms. Suitable candidates are Solana smart contracts written in Rust, and Hyperledger Fabric smart contracts written in Java. For both languages, source-level formal verification tools exist, so that specification generation is feasible. An interesting research question is how access control and other security properties can be translated, because the other platforms have built-in features which may constitute a better alternative to Solidity's function-level access control.

Implementation and Tool Support The SCAR approach has been implemented, albeit in a somewhat prototypical fashion. In the future, we will work to make the implementation more complete and more usable. For example, it should be easier for a developer to integrate a new formal verification tool. If source-level verification fails, it should also be possible to give feedback on the model level, guiding the developer to possible sources of error, or proving that a source-level verification failure is not actually relevant to a desired application-level property.

Integration with other Formal Methods The main benefit of the SCAR approach is its platform character: It is intended to serve as a basis for the application and integration of other tools. Therefore, in the future, we will pursue the integration of different formal methods in SCAR. We would also like to position SCAR as a common format to define formal verification benchmarks.

List of Figures

1.1 Illustration of the blockchain data structure 5
1.2 A basic Solidity smart contract 16
1.3 Solidity simple bank application 18
1.4 Solidity auction application 19
1.5 A Solidity escrow smart contract, taken from [Zyn] 21
1.6 ERC20 token standard 22
1.7 Palinodia overview 24

3.1 The Scar process of smart contract application development 51
3.2 Scar core metamodel 55
3.3 A graphical description of the type system 56
3.4 ScarML grammar 57
3.5 ScarML frame condition grammar 58
3.6 The ScarML evaluation function $[\![\cdot]\!]$ 62
3.7 ScarML frame condition evaluation 63
3.8 An example execution trace. Steps τ_3 and τ_4 are environment steps. 67
3.9 Scar metamodel grammar 68
3.10 A simple Scar application model 72
3.11 The generated Solidity code with solc-verify annotations 73
3.12 Banking with interest 77
3.13 Scar Auction 78
3.14 Scar Escrow 79
3.15 Scar model of Palinodia Software contract 81
3.16 Lines of code comparison 82

4.1 Capabilities Syntax 88
4.2 Simple bank application with capability specification 90
4.3 Call Capability Evaluation 91
4.4 Modify Capability Evaluation 92
4.5 Transfer Capability Evaluation 93
4.6 Example access control modifier 97

4.7 Example Wrapped Transfer Function 98
4.8 Example SOLC-VERIFY Frame Condition 99
4.9 Static Palinodia SCAR model 102

5.1 SCARTL syntax 112
5.2 Simple temporal invariant 117
5.3 SMT-LIB translation for simple invariant example 117
5.4 SCAR model of the Casino application 122

6.1 A simple bank contract written in Solidity 134
6.2 An example of unintuitive behavior in Ethereum (cf. [Hit18]) 135
6.3 Fabric chaincode from an auction contract 140

7.1 The SCAR process of smart contract application development 147

List of Tables

3.1 *ValsOf* function and default values 59
3.2 *LocsOf* function 60
3.3 *ValsOf* function and default values 61
3.4 The requirements of the *Software* contract 80

6.1 Syntax of the Solidity frame condition language 130
6.2 Rules for the evaluation function $[\![\cdot]\!]^s$ (see Definition 19) 132
6.3 Syntax of the Fabric frame condition language 138
6.4 Rules for the evaluation function 139

Bibliography

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[AEP18] Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. “Monitoring Smart Contracts: ContractLarva and Open Challenges Beyond”. In: *Runtime Verification*. Ed. by Christian Colombo and Martin Leucker. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 113–137.

[Ahr+16] Wolfgang Ahrendt et al., eds. *Deductive Software Verification – The KeY Book: From Theory to Practice*. Programming and Software Engineering. Springer International Publishing, 2016.

[Ama+18] Sidney Amani et al. “Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL”. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs - CPP 2018*. Los Angeles, CA, USA: ACM Press, 2018, pp. 66–77.

[And+18] Elli Androulaki et al. “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains”. In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys ’18. New York, NY, USA: ACM, 2018, 30:1–30:15.

[Ann+21] Danil Annenkov et al. “Extracting Smart Contracts Tested and Verified in Coq”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2021. New York, NY, USA: Association for Computing Machinery, Jan. 2021, pp. 105–121.

[ANS20] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. “ConCert: A Smart Contract Certification Framework in Coq”. In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New York, NY, USA: Association for Computing Machinery, Jan. 2020, pp. 215–228.

[Ant+22] Pedro Antonino et al. "Specification Is Law: Safe Creation and Upgrade of Ethereum Smart Contracts". In: *Software Engineering and Formal Methods*. Ed. by Bernd-Holger Schlingloff and Ming Chai. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 227–243.

[Ant+24] Pedro Antonino et al. "A Refinement-Based Approach to Safe Smart Contract Deployment and Evolution". In: *Software and Systems Modeling* (Jan. 2024). ISSN: 1619-1374.

[APS18] Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. "Smart Contracts: A Killer Application for Deductive Source Code Verification". In: *Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of His 60th Birthday*. Ed. by Peter Müller and Ina Schaefer. Cham: Springer International Publishing, 2018, pp. 1–18.

[AR21] 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. New York, NY, USA: Association for Computing Machinery, Apr. 2021, pp. 1788–1797.

[Ast+22] Vytautas Astrauskas et al. "The Prusti Project: Formal Verification for Rust". In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Cham: Springer International Publishing, 2022, pp. 88–108.

[AWH21] Wolfgang Ahrendt, Alexander Weigl, and Paula Herber. *VerifyThis Long-Term Challenge*. 2021. URL: `https://verifythis.github.io/02casino/` (visited on 08/28/2024).

[Bac02] Adam Back. *Hashcash - a Denial of Service Counter-Measure*. 2002.

[Bar+06] Mike Barnett et al. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer et al. Berlin, Heidelberg: Springer, 2006, pp. 364–387.

[Bar+11] Clark Barrett et al. "CVC4". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer, 2011, pp. 171–177.

[Bar+24] Massimo Bartoletti et al. *Towards Benchmarking of Solidity Verification Tools*. Feb. 2024. arXiv: `2402.10750 [cs]`. (Visited on 08/17/2024).

[Bau+08] Patrick Baudin et al. *ACSL: ANSI/ISO C Specification Language (Version 1.4)*. 2008.

[Bec+18] Bernhard Beckert et al. "Formal Specification and Verification of Hyperledger Fabric Chaincode". In: *SDLT 2018: The 3rd Symposium on Distributed Ledger Technology*. 2018.

[Bha+16] Karthikeyan Bhargavan et al. "Formal Verification of Smart Contracts: Short Paper". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security - PLAS'16*. Vienna, Austria: ACM Press, 2016, pp. 91–96.

[Bli+15] Simon Bliudze et al. "Formal Verification of Infinite-State BIP Models". In: *Automated Technology for Verification and Analysis*. Ed. by Bernd Finkbeiner, Geguang Pu, and Lijun Zhang. Cham: Springer International Publishing, 2015, pp. 326–343.

[Bon+24] Federico Bond et al. *GitHub - Solidityj/Solidity-Antlr4: Solidity Grammar for ANTLR4*. 2024. URL: `https://github.com/solidityj/solidity-antlr4` (visited on 08/28/2024).

[BP17] Massimo Bartoletti and Livio Pompianu. "An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns". In: *Financial Cryptography and Data Security*. Ed. by Michael Brenner et al. Vol. 10323. Cham: Springer International Publishing, 2017, pp. 494–509.

[Brä+21] Christian Bräm et al. "Modular Verification of Collaborating Smart Contracts". In: *ArXiv* abs/2104.10274 (2021).

[Bra19] Harris Brakmić. "Bitcoin Script". In: *Bitcoin and Lightning Network on Raspberry Pi: Running Nodes on Pi3, Pi4 and Pi Zero*. Ed. by Harris Brakmić. Berkeley, CA: Apress, 2019, pp. 201–224.

[BS20] Bernhard Beckert and Jonas Schiffl. "Specifying Framing Conditions for Smart Contracts". In: *Leveraging Applications of Formal Methods, Verification and Validation: Applications*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 43–59.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. "The Smt-Lib Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[But13] Vitalik Buterin. *Ethereum White Paper*. 2013. URL: `https://github.com/ethereum/wiki/wiki/White-Paper` (visited on 08/22/2024).

[Cas+23] Franck Cassez et al. “Formal and Executable Semantics of the Ethereum Virtual Machine in Dafny”. In: *Formal Methods: 25th International Symposium, FM 2023, Lübeck, Germany, March 6–10, 2023, Proceedings*. Berlin, Heidelberg: Springer-Verlag, Mar. 2023, pp. 571–583.

[Cav+14] Roberto Cavada et al. “The nuXmv Symbolic Model Checker”. In: *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 2014, pp. 334–342.

[Cer24] CertiK. *CertiK - Securing The Web3 World*. 2024. URL: `https://www.certik.com` (visited on 08/20/2024).

[CFA22] Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. “Formal Verification of the Ethereum 2.0 Beacon Chain”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 167–182.

[CFQ22] Franck Cassez, Joanne Fuller, and Horacio Mijail Antón Quiles. “Deductive Verification of Smart Contracts with Dafny”. In: *Formal Methods for Industrial Critical Systems*. Ed. by Jan Friso Groote and Marieke Huisman. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 50–66.

[Cha+23] Stefanos Chaliasos et al. *Smart Contract and DeFi Security: Insights from Tool Evaluations and Practitioner Surveys*. Apr. 2023. arXiv: `2304.02981 [cs]`. (Visited on 04/17/2023).

[Cha+81] Donald D Chamberlin et al. “A History and Evaluation of System R”. In: *Communications of the ACM* 24.10 (1981), pp. 632–646.

[Cob+20] Michael Coblenz et al. “Obsidian: Typestate and Assets for Safer Blockchain Programming”. In: *ACM Trans. Program. Lang. Syst.* 42.3 (Nov. 2020), 14:1–14:82. ISSN: 0164-0925.

[Cob17] Michael Coblenz. “Obsidian: A Safer Blockchain Programming Language”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. May 2017, pp. 97–99.

[Coi24a] CoinGecko. *Top Smart Contract Platform Coins by Market Cap*. 2024. URL: `https://www.coingecko.com/en/categories/smart-contract-platform` (visited on 08/22/2024).

[Coi24b] CoinMarketCap. *Top Smart Contracts Tokens by Market Capitalization*. 2024. URL: https://coinmarketcap.com/view/smart-contracts/ (visited on 06/22/2024).

[Con20] Consensys. *Known Attacks - Ethereum Smart Contract Best Practices*. 2020. URL: https://consensys.github.io/smart-contract-best-practices/known_attacks/ (visited on 05/26/2020).

[Con24] Concordium. *Concordium*. 2024. URL: https://www.concordium.com/ (visited on 06/23/2024).

[CPP20] Arnab Chatterjee, Yash Pitroda, and Manojkumar Parmar. "Dynamic Role-Based Access Control for Decentralized Applications". In: *Blockchain – ICBC 2020*. Ed. by Zhixiong Chen et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 185–197.

[Cro+16] Kyle Croman et al. "On Scaling Decentralized Blockchains". In: *Financial Cryptography and Data Security*. Ed. by Jeremy Clark et al. Berlin, Heidelberg: Springer, 2016, pp. 106–125.

[Cui+22] Siwei Cui et al. "VRust: Automated Vulnerability Detection for Solana Smart Contracts". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 639–652.

[CVC24] CVC5. *Theory Reference: Sets and Relations — CVC5 Documentation*. 2024. URL: https://cvc5.github.io/docs/cvc5-1.0.0/theories/sets-and-relations.html#finite-sets (visited on 07/23/2024).

[dB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[DH76] W. Diffie and M. Hellman. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (Nov. 1976), pp. 644–654. ISSN: 1557-9654.

[Dha+21] Samvid Dharanikota et al. "Celestial: A Smart Contracts Verification Framework". In: *2021 Formal Methods in Computer Aided Design (FMCAD)*. Oct. 2021, pp. 133–142.

[Di +18] Giovanni Di Stasi et al. "Routing Payments on the Lightning Network". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData).* July 2018, pp. 1161–1170.

[Doc] Solidity Documentation. *Solidity by Example — Solidity 0.8.27 Documentation.* URL: `https://docs.soliditylang.org/en/latest/solidity-by-example.html#simple-open-auction` (visited on 08/28/2024).

[DP08] Dino Distefano and Matthew J. Parkinson J. "jStar: Towards Practical Verification for Java". In: *ACM SIGPLAN Notices* 43.10 (Oct. 2008), pp. 213–226. ISSN: 0362-1340.

[Dur+20] Thomas Durieux et al. "Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* ICSE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 530–541.

[EP18] Joshua Ellul and Gordon J Pace. "Runtime Verification of Ethereum Smart Contracts". In: *2018 14th European Dependable Computing Conference (EDCC).* IEEE, 2018, pp. 158–163.

[eth24a] etherscan.io. *EthereumDIDRegistry.* 2024. URL: `https://etherscan.io/address/0xdca7ef03e98e0dc2b855be647c39abe984fcf21b` (visited on 08/28/2024).

[eth24b] ethr-did-registry. *Uport-Project/Ethr-Did-Registry.* Veramo core development. July 2024. URL: `https://github.com/uport-project/ethr-did-registry` (visited on 07/19/2024).

[Fer+20] João F. Ferreira et al. "SmartBugs: A Framework to Analyze Solidity Smart Contracts". In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* ASE '20. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 1349–1352.

[FGG19] Josselin Feist, Gustavo Grieco, and Alex 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).* May 2019, pp. 8–15.

[Fou23] Ethereum Foundation. *ERC-721 Non-Fungible Token Standard.* 2023. URL: `https://ethereum.org/en/developers/docs/standards/tokens/erc-721/` (visited on 07/18/2024).

[Fou24a] Aptos Foundation. *Aptos Developer Documentation | Aptos Docs.* Apr. 2024. URL: `https://aptos.dev/` (visited on 06/23/2024).

[Fou24b] Ethereum Foundation. *ERC-20 Token Standard.* 2024. URL: `https://ethereum.org/en/developers/docs/standards/tokens/erc-20/` (visited on 07/18/2024).

[Fri+21] Sebastian Friebe et al. "Coupling Smart Contracts: A Comparative Case Study". In: *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS).* Sept. 2021, pp. 137–144.

[Gar+18] Péter Garamvölgyi et al. "Towards Model-Driven Engineering of Smart Contracts for Cyber-Physical Systems". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W).* June 2018, pp. 134–139.

[Gar+22] Ikram Garfatta et al. "Model Checking of Vulnerabilities in Smart Contracts: A Solidity-to-CPN Approach". In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing.* SAC '22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 316–325.

[Gen+18] Adem Efe Gencer et al. "Decentralization in Bitcoin and Ethereum Networks". In: *Financial Cryptography and Data Security.* Ed. by Sarah Meiklejohn and Kazue Sako. Berlin, Heidelberg: Springer, 2018, pp. 439–457.

[Ger24] David Gerard. *Attack of the 50 Foot Blockchain.* June 2024. URL: `https://davidgerard.co.uk/blockchain/` (visited on 06/24/2024).

[Gol18] Phillip Goldberg. *Smart Contract Best Practices Revisited: Block Number vs. Timestamp.* Mar. 2018. URL: `https://medium.com/@phillipgoldberg/smart-contract-best-practices-revisited-block-number-vs-timestamp-648905104323` (visited on 08/20/2024).

[Gri+20] Gustavo Grieco et al. "Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ISSTA 2020. New York, NY, USA: Association for Computing Machinery, July 2020, pp. 557–560.

[GS23] Zachary Grannan and Alexander J. Summers. “Resource Specifications for Resource-Manipulating Programs”. In: *CoRR* abs/2304.12530 (2023). arXiv: 2304.12530.

[Gur+15] Arie Gurfinkel et al. “The SeaHorn Verification Framework”. In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 343–361.

[HBS23] Sebastian Holler, Sebastian Biewer, and Clara Schneidewind. “HoRStify: Sound Security Analysis of Smart Contracts”. In: *36th IEEE Computer Security Foundations Symposium, CSF 2023, Dubrovnik, Croatia, July 10-14, 2023*. IEEE, 2023, pp. 245–260.

[He+23] Daojing He et al. “Detection of Vulnerabilities of Blockchain Smart Contracts”. In: *IEEE Internet of Things Journal* (2023), pp. 1–1. ISSN: 2327-4662.

[Hei+13] Matthias Heizmann et al. “Ultimate Automizer with SMTInterpol”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nir Piterman and Scott A. Smolka. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 641–643.

[Hil+18] E. Hildenbrandt et al. “KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine”. In: *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. July 2018, pp. 204–217.

[Hir17] Yoichi Hirai. “Defining the Ethereum Virtual Machine for Interactive Theorem Provers”. In: *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 520–535.

[Hit18] Rob Hitchens. *Storage Pointers in Solidity*. Nov. 2018.

[HJ20] Akos Hajdu and Dejan Jovanovic. “Solc-Verify: A Modular Verifier for Solidity Smart Contracts”. In: *Verified Software. Theories, Tools, and Experiments*. Ed. by Supratik Chakraborty and Jorge A. Navas. Cham: Springer International Publishing, 2020, pp. 161–179.

[HMQ20] Mohammad Hamdaqa, Lucas Alberto Pineda Metz, and Ilham Qasse. “iContractML: A Domain-Specific Language for Modeling and Deploying Smart Contracts onto Multiple Blockchain Platforms”. In: *Proceedings of the 12th System Analysis and Modelling Conference*. SAM ’20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 34–43.

[Iva+23] Nikolay Ivanov et al. “Security Threat Mitigation for Smart Contracts: A Comprehensive Survey”. In: *Acm Computing Surveys* 55.14s (2023), 326:1–326:37.

[Jia+20] Jiao Jiao et al. "Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1695–1712.

[Jun+17] Ralf Jung et al. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017), 66:1–66:34.

[Jur+23] Mantas Jurgelaitis et al. "MDA-Based Approach for Blockchain Smart Contract Development". In: *Applied Sciences* 13.1 (Jan. 2023), p. 487. ISSN: 2076-3417.

[Kal+18] Sukrit Kalra et al. "Zeus: Analyzing Safety of Smart Contracts." In: *Ndss*. 2018, pp. 1–12.

[Kas06] Ioannis T. Kassios. "Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions". In: *FM 2006: Formal Methods*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 268–283.

[Kol+20] John Kolb et al. "Quartz: A Framework for Engineering Secure Smart Contracts". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2020-178* (2020).

[Lah+20] Asma Lahbib et al. "An Event-B Based Approach for Formal Modelling and Verification of Smart Contracts". In: *Advanced Information Networking and Applications*. Ed. by Leonard Barolli et al. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2020, pp. 1303–1318.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[LC06] Gary T Leavens and Yoonsik Cheon. "Design by Contract with JML". In: *UCF Department of Electrical Engineering and Computer Science* (Sept. 2006), p. 12.

[Lei10] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer, 2010, pp. 348–370.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. USA: Butterworth-Heinemann, 1984.

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. "A Solver for Reachability Modulo Theories". In: *Computer Aided Verification*. Ed. by P. Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer, 2012, pp. 427–443.

[Luu+16] Loi Luu et al. "Making Smart Contracts Smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. Vienna, Austria: ACM Press, 2016, pp. 254–269.

[Mar+20] Matteo Marescotti et al. "Accurate Smart Contract Verification Through Direct Modelling". In: *Leveraging Applications of Formal Methods, Verification and Validation: Applications*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 178–194.

[Mav+19] Anastasia Mavridou et al. "VeriSolid: Correct-by-Design Smart Contracts for Ethereum". In: *Financial Cryptography and Data Security*. Ed. by Ian Goldberg and Tyler Moore. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 446–465.

[MB22] Diego Marmsoler and Achim D. Brucker. "Isabelle/Solidity: A Deep Embedding of Solidity in Isabelle/HOL". In: *Arch. Formal Proofs* 2022 (2022). URL: `https://www.isa-afp.org/entries/Solidity.html` (visited on 08/15/2024).

[Meh+19] Muhammad Izhar Mehar et al. "Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack". In: *Journal of Cases on Information Technology (JCIT)* 21.1 (Jan. 2019), pp. 19–32. ISSN: 1548-7717.

[Mer88] Ralph C. Merkle. "A Digital Signature Based on a Conventional Encryption Function". In: *Advances in Cryptology — CRYPTO '87*. Ed. by Carl Pomerance. Berlin, Heidelberg: Springer, 1988, pp. 369–378.

[Mey92] B. Meyer. "Applying 'Design by Contract'". In: *Computer* 25.10 (Oct. 1992), pp. 40–51. ISSN: 0018-9162.

[Mis+24] Debani Prasad Mishra et al. "Smart Contract: Tools and Challenges". In: *2024 1st International Conference on Cognitive, Green and Ubiquitous Computing (IC-CGU)*. Mar. 2024, pp. 1–6.

[Mit24] Mitre. *CVE - CVE*. 2024. URL: `https://cve.mitre.org/` (visited on 06/25/2024).

[ML18] Anastasia Mavridou and Aron Laszka. "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach". In: *Financial Cryptography and Data Security*. Ed. by Sarah Meiklejohn and Kazue Sako. Berlin, Heidelberg: Springer, 2018, pp. 523–540.

[Mov24] Move on Aptos. *Move-Language/Move-on-Aptos*. The Move Programming Language. Aug. 2024. URL: `https://github.com/move-language/move-on-aptos` (visited on 08/14/2024).

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A Verification Infrastructure for Permission-Based Reasoning". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Berlin, Heidelberg: Springer, 2016, pp. 41–62.

[MT23] Sundas Munir and Walid Taha. *Pre-Deployment Analysis of Smart Contracts – A Survey*. Jan. 2023. arXiv: `2301.06079 [cs]`. (Visited on 01/27/2023).

[Mue18] Bernhard Mueller. *Smashing Ethereum Smart Contracts for Fun and Real Profit*. 2018. URL: `https://github.com/muellerberndt/smashing-smart-contracts/blob/0663ad015b0a6ce08053d48731cdee1e7bc4e726/smashing-smart-contracts-1of1.pdf` (visited on 10/14/2022).

[Nak08] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: `https://static.upbitcare.com/931b8bfc-f0e0-4588-be6e-b98a27991df1.pdf` (visited on 08/28/2024).

[Nel+20] Keerthi Nelaturu et al. "Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid". In: *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. May 2020, pp. 1–9.

[Nic21] Nassim Nicholas Taleb. "Bitcoin, Currencies, and Fragility". In: *Quantitative Finance* 21.8 (2021), pp. 1249–1255.

[Nik+18] Ivica Nikolić et al. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18. New York, NY, USA: ACM, 2018, pp. 653–663.

[NIS02] NIST. *Announcing Approval of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard; a Revision of FIPS 180-1*. Aug. 2002. URL: https://www.federalregister.gov/documents/2002/08/26/02-21599/announcing-approval-of-federal-information-processing-standard-fips-180-2-secure-hash-standard-a (visited on 06/17/2024).

[NPD18] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. "Model-Checking of Smart Contracts". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. July 2018, pp. 980–987.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002.

[Obs] Obsidian. *Using the Compiler — Obsidian 0.1 Documentation*. URL: https://obsidian.readthedocs.io/en/latest/compiling.html (visited on 08/10/2024).

[Off24] Office of Public Affairs, USDOJ. *Office of Public Affairs | Man Convicted for $110M Cryptocurrency Scheme | United States Department of Justice*. Apr. 2024. URL: https://www.justice.gov/opa/pr/man-convicted-110m-cryptocurrency-scheme (visited on 08/22/2024).

[OM14] Karl J. O'Dwyer and David Malone. "Bitcoin Mining and Its Energy Footprint". In: *25th IET Irish Signals & Systems Conference 2014 and 2014 China-Ireland International Conference on Information and Communications Technologies (ISSC 2014/CIICT 2014)*. June 2014, pp. 280–285. (Visited on 06/24/2024).

[Oto+23] Rodrigo Otoni et al. "A Solicitous Approach to Smart Contract Verification". In: *ACM Trans. Priv. Secur.* 26.2 (Mar. 2023), 15:1–15:28. ISSN: 2471-2566.

[Ott24] OtterSec. *Solana Formal Verification: A Case Study*. Feb. 2024. URL: https://osec.io (visited on 02/17/2024).

[Pau11] Christine Paulin-Mohring. "Introduction to the Coq Proof-Assistant for Practical Software Verification". In: *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer

and Martin Nordio. Vol. 7682. Lecture Notes in Computer Science. Springer, 2011, pp. 45–95.

[Per+20] Anton Permenev et al. "VerX: Safety Verification of Smart Contracts". In: *2020 IEEE Symposium on Security and Privacy (SP).* May 2020, pp. 1661–1677.

[Pet+18] Jack Peterson et al. "Augur: A Decentralized Oracle and Prediction Market Platform". In: (2018). arXiv: `1501.01042 [cs]`. (Visited on 08/22/2024).

[Pnu77] Amir Pnueli. "The Temporal Logic of Programs". In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977.* IEEE Computer Society, 1977, pp. 46–57.

[Pra+20] Purathani Praitheeshan et al. "Security Analysis Methods on Ethereum Smart Contract Vulnerabilities: A Survey". In: *arXiv:1908.08605 [cs]* (Jan. 2020). arXiv: `1908.08605 [cs]`. URL: `http://arxiv.org/abs/1908.08605` (visited on 05/26/2020).

[PS12] Matthew J. Parkinson and Alexander J. Summers. "The Relationship Between Separation Logic and Implicit Dynamic Frames". In: *Logical Methods in Computer Science* 8.3 (July 2012), p. 1. ISSN: 18605974. arXiv: `1203.6859`. (Visited on 05/19/2020).

[PT22] Andrea Pinna and Roberto Tonelli. "On the Use of Petri Nets in Smart Contracts Modeling, Generation and Verification". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).* Mar. 2022, pp. 1207–1211. (Visited on 08/12/2024).

[Rey02] J.C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* July 2002, pp. 55–74.

[Ris09] Tore Risch. "Stored Procedure". In: *Encyclopedia of Database Systems.* Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 2831–2831.

[Riv92] Ronald Rivest. *RFC1321: The MD5 Message-Digest Algorithm.* 1992.

[RȘ10] Grigore Roșu and Traian Florin Șerbănuță. "An Overview of the K Semantic Framework". In: *The Journal of Logic and Algebraic Programming.* Membrane Computing and Programming 79.6 (Aug. 2010), pp. 397–434. ISSN: 1567-8326.

[SB24] Jonas Schiffl and Bernhard Beckert. “A Practical Notion of Liveness in Smart Contract Applications”. In: *OASIcs FMBC 2024*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

[SC21] Abdurrashid Ibrahim Sanka and Ray C. C. Cheung. “A Systematic Review of Blockchain Scalability: Issues, Solutions, Analysis and Future Research”. In: *Journal of Network and Computer Applications* 195 (Dec. 2021), p. 103232. ISSN: 1084-8045.

[Sch+20] Clara Schneidewind et al. “eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 621–640.

[Sch+21] Jonas Schiffl et al. “Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control”. In: *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies*. SACMAT ’21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 125–130.

[Sch24] Jonas Schiffl. *Jonas Schiffl / Scar · GitLab*. June 2024. URL: `https://gitlab.kit.edu/jonas.schiffl/Scar` (visited on 07/03/2024).

[SCr23] SCribble. *Introduction | Scribble*. July 2023. URL: `https://docs.scribble.codes` (visited on 08/12/2024).

[SCS24] SCSVS. *GitHub - ComposableSecurity/SCSVS: Smart Contract Security Verification Standard*. 2024. URL: `https://github.com/ComposableSecurity/SCSVS` (visited on 06/25/2024).

[SDH20] Oliver Stengele, Jan Droll, and Hannes Hartenstein. “Practical Trade-Offs in Integrity Protection for Binaries via Ethereum”. In: *Proceedings of the 21st International Middleware Conference Demos and Posters*. Middleware ’20 Demos and Posters. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 9–10.

[Ser+19] Ilya Sergey et al. “Safer Smart Contract Programming with Scilla”. In: *Benchmarks accompanying the OOPSLA 2019 paper Safer Smart Contract Programming with Scilla* 3.OOPSLA (Oct. 2019), 185:1–185:30. (Visited on 08/15/2024).

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. “Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic”. In: *ECOOP 2009 – Object-Oriented Programming*. Ed. by Sophia Drossopoulou. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 148–172.

[SKH18] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. "Temporal Properties of Smart Contracts". In: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*. Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 323–338.

[Sol24a] Solana. *Developing with Rust | Solana*. 2024. URL: `https://solana.com/docs/programs/lang-rust` (visited on 08/22/2024).

[Sol24b] Solana. *getBlockTime RPC Method | Solana*. 2024. URL: `https://solana.com/docs/rpc/http/getblocktime` (visited on 08/20/2024).

[SP20] Marek Skotnica and Robert Pergl. "Das Contract - A Visual Domain Specific Language for Modeling Blockchain Smart Contracts". In: *Advances in Enterprise Engineering XIII*. Ed. by David Aveiro, Giancarlo Guizzardi, and José Borbinha. Lecture Notes in Business Information Processing. Cham: Springer International Publishing, 2020, pp. 149–166.

[Ste+19] Oliver Stengele et al. "Access Control for Binary Integrity Protection Using Ethereum". In: *Proceedings of the 24th ACM Symposium on Access Control Models and Technologies*. SACMAT '19. New York, NY, USA: Association for Computing Machinery, May 2019, pp. 3–12.

[Ste+21] Jon Stephens et al. "SmartPulse: Automated Checking of Temporal Properties in Smart Contracts". In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 555–571.

[Sul+22] Evan Sultanik et al. "Are Blockchains Decentralized?" In: *Trail Of Bits* (2022).

[SWB23] Jonas Schiffl, Alexander Weigl, and Bernhard Beckert. "Static Capability-based Security for Smart Contracts". In: *2023 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. Athens, July 2023.

[SWC23] SWCRegistry. *Overview · Smart Contract Weakness Classification and Test Cases*. 2023. URL: `http://swcregistry.io/` (visited on 03/30/2023).

[Sza97] Nick Szabo. "Formalizing and Securing Relationships on Public Networks". In: *First Monday* 2.9 (Sept. 1997). ISSN: 13960466. (Visited on 09/20/2018).

[Tav22] Tien N. Tavu. "Automated Verification Techniques for Solana Smart Contracts". Thesis. Aug. 2022. URL: https://oaktrust.library.tamu.edu/handle/1969.1/196530 (visited on 02/17/2024).

[Tea] Solidity Team. *Home*. URL: https://soliditylang.org/ (visited on 08/28/2024).

[Tea24] Vyper Team. *Vyper*. 2024. URL: https://docs.vyperlang.org/en/latest/index.html (visited on 06/22/2024).

[Töb+22] Jan-Philipp Töberg et al. "Modeling and Enforcing Access Control Policies for Smart Contracts". In: *2022 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. Aug. 2022, pp. 38–47.

[Tor+13] Emina Torlak et al. "Applications and Extensions of Alloy: Past, Present and Future". In: *Mathematical Structures in Computer Science* 23.4 (Aug. 2013), pp. 915–933. ISSN: 0960-1295, 1469-8072.

[Tsa+18] Petar Tsankov et al. "Securify: Practical Security Analysis of Smart Contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 67–82.

[VIL23] Fernando Richter Vidal, Naghmeh Ivaki, and Nuno Laranjeiro. "OpenSCV: An Open Hierachical Taxonomy for Smart Contract Vulnerabilities". In: *arXiv preprint arXiv:2303.14523* (2023). arXiv: 2303.14523.

[Wan+19] Yuepeng Wang et al. "Formal Specification and Verification of Smart Contracts for Azure Blockchain". In: (Apr. 2019).

[Wan+20] Yilei Wang et al. "Incentive Compatible and Anti-Compounding of Wealth in Proof-of-Stake". In: *Information Sciences* 530 (Aug. 2020), pp. 85–94. ISSN: 0020-0255. (Visited on 06/24/2024).

[Wes+22] Scott Wesley et al. "Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernd Finkbeiner and Thomas Wies. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 425–449.

[Whi22] Molly White. *Cryptocurrency 'market Caps' and Notional Value*. July 2022. URL: https://blog.mollywhite.net/cryptocurrency-market-caps-and-notional-value/ (visited on 06/24/2024).

[Whi24] Molly White. *Web3 Is Going Just Great*. Feb. 2024. URL: `https://web3isgoinggreat.com/` (visited on 02/13/2024).

[Woo14] Gavin Wood. "Ethereum: A Secure Decentralised Generalised Transaction Ledger". In: *Ethereum project yellow paper* 151 (2014), pp. 1–32.

[WZ18] Maximilian Wöhrer and Uwe Zdun. "Design Patterns for Smart Contracts in the Ethereum Ecosystem". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. July 2018, pp. 1513–1520.

[XLF13] Tao Xie, Fanbao Liu, and Dengguo Feng. *Fast Collision Attack on MD5*. 2013. URL: `https://eprint.iacr.org/2013/170` (visited on 08/22/2024).

[Yak18] Anatoly Yakovenko. "Solana: A New Architecture for a High Performance Blockchain v0. 8.13". In: *Whitepaper* (2018).

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications". In: *Correct Hardware Design and Verification Methods*. Ed. by Laurence Pierre and Thomas Kropf. Berlin, Heidelberg: Springer, 1999, pp. 54–66.

[Zha+23] Zhuo Zhang et al. "Demystifying Exploitable Bugs in Smart Contracts". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. May 2023, pp. 615–627. (Visited on 12/10/2024).

[Zho+20] Jingyi Emma Zhong et al. "The Move Prover". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 137–150.

[Zhu+17] Shufang Zhu et al. "A Symbolic Approach to Safety LTL Synthesis". In: *Hardware and Software: Verification and Testing: 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings 13*. Springer, 2017, pp. 147–162.

[Zyn] Ed Zynda. *Code An Escrow Smart Contract In Solidity | Codementor*. URL: `https://www.codementor.io/@edzynda/code-an-escrow-smart-contract-in-solidity-14piv60xb6` (visited on 07/18/2024).