École Polytechnique Fédérale de Lausanne

ETH Zürich

Low-overhead functional property checking for smart-contract systems

by Valentin Quelquejay-Leclère

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Dr. Valentin Wüstholz
Thesis Supervisor

Joran Honig, Dr. Dimitar Bounov
Co-Supervisors

Consensys Diligence

February 21, 2023

Logic will get you from A to B. Imagination will take you everywhere.
— Albert Einstein

Acknowledgments

This thesis would not have been possible without the precious support and advice of many people. Unfortunately, it is impossible to write an exhaustive list of all the people who have supported me in one way or another these last years. I sincerely thank each of them.

I would like to thank my family: my mother and my father without whom all this would never have happened, but also my siblings: Arthur, Augustin, Clémentine, and Antoine for supporting me during my studies.

I also want to thank my close friends who are and have been extremely important to me, especially in difficult times. Special thanks to my roommate Jean for our amazing discussions, projects, and ideas that always brought me motivation and opportunities.

Many thanks to Valentin Wüstholz, my thesis supervisor, who enabled me to write this thesis in such a great team, and provided me with invaluable advice, feedback, and ideas. Thank you to Joran and Dimitar for sharing their time, support, and feedback, even with the eleven-hour time difference. Of course, I would also like to thank the rest of the incredible Dili team, with whom I shared moments that I will remember forever.

Finally, I would like to thank my thesis advisor, Matthias Payer, who devoted his valuable time to advising this thesis.

Zürich, February 21, 2023 Valentin Quelquejay-Leclère

Abstract

Smart contracts secure high-value digital assets on the blockchain. Thus, they are the target of many attacks, and ensuring their security is critical. Looking at past attacks, we realize that most of the smart-contract hacks are due to business-logic bugs. Unfortunately, catching these bugs is difficult. Property-based testing is a good candidate for detecting these vulnerabilities. It enables to express key system specifications using few functional properties. Existing approaches have mostly been used for checking non-trivial functional properties of local contracts since they rely on code instrumentation. This makes it especially difficult to check certain functional properties on live contracts. We design a smart-contract specification language, *Jessy*, that enables expressing functional properties checkable at runtime without code instrumentation. We also introduce a prototype tool that replays past transactions and dynamically checks Jessy properties. We evaluate our prototype on eight recent real-world contracts that were attacked. We manage to write a property that captures the bug for five of them, validating our language design decisions. Our evaluation also provides us with insights into new language constructs that would help capture additional vulnerabilities.

Contents

Acknowledgments 1

Abstract 2

1 Introduction 5

2 Background 9
- 2.1 Ethereum and the EVM . . . 9
- 2.2 Solidity . . . 10
- 2.3 Solidity contracts from an object-oriented perspective . . . 12
- 2.4 Smart contract vulnerabilities . . . 14
- 2.5 Properties . . . 14

3 Design 16
- 3.1 Specification Language . . . 17
- 3.2 Checker contract . . . 20
- 3.3 Language semantics . . . 21
  - 3.3.1 Contract Invariants . . . 21
  - 3.3.2 Call specifications: functional specifications . . . 24
  - 3.3.3 Correctness of a contract . . . 27

4 Implementation 29
- 4.1 High-level design . . . 29
- 4.2 Parser and checker contract generator . . . 30
- 4.3 Transaction analyzer and simulator . . . 31
- 4.4 Property checking . . . 34

5 Evaluation 35
- 5.1 Experimental Setup . . . 35
- 5.2 Omni protocol . . . 36
- 5.3 Rari Fuse . . . 37
- 5.4 Audius . . . 39

5.5 Qubit Finance . . . 40
5.6 Nomad bridge . . . 42
5.7 Poly Network Hack . . . 43
5.8 Compound . . . 44
5.9 Harvest Finance . . . 46
5.10 Summary and discussion . . . 47

6 Related Work **48**
6.1 Smart contract verification . . . 48
6.2 Property based testing . . . 49
6.3 Specification languages and runtime property checking . . . 49
6.4 Runtime monitoring . . . 50

7 Conclusion **51**

Bibliography **53**

A Code of vulnerable functions **57**
A.1 Compound: `distributeSupplierComp` . . . 57
A.2 Rari Fuse: `borrow` . . . 58
A.3 Audius: `initialize` . . . 59
A.4 Qubit Finance: `deposit`, `depositETH` . . . 61
A.5 Nomad Bridge: `process`, `acceptableRoot` . . . 63

B Additional Material **65**

C Extension Ideas **67**

Chapter 1

Introduction

Smart contracts are programs running on a blockchain. As of now, one of the most popular blockchains is Ethereum. Smart contracts running on Ethereum are often written in Solidity, a statically-typed language that compiles to bytecode executed on the Ethereum Virtual Machine (EVM). Smart contracts manipulate digital assets. The value of assets stored in smart contracts increases every day. Therefore, ensuring their correctness and their security is crucial. Several automated vulnerability detection tools have been developed specifically for smart contracts. These tools rely on different methods such as static analysis or dynamic analysis, e.g., fuzzing or symbolic execution.

Traditional software library testing usually aims at detecting vulnerabilities such as type and memory safety vulnerabilities. One of the challenges with smart-contract systems is that they often interact with external systems and are vulnerable to blockchain-specific threats, e.g., transaction ordering vulnerabilities or oracle attacks. This is especially true for complex systems such as decentralized finance (DeFi) protocols or decentralized anonymous organizations (DAOs) which are subject to business logic or economic attacks. Interoperability adds value to protocols but also usually leads to increased complexity and thus, increased likelihood of an attack. It creates parts of the code that the protocol developers do not control. Developers must fully understand the code they interact with and all the side effects on the protocol itself before implementing such interoperability. DeFi interoperability brings a lot of power to the DeFi ecosystem but also leads to increased security risks. It creates the opportunity to mount multi-step attacks involving several protocols. These threats are challenging to detect with security tools such as static analysis or runtime verification tools. One needs proper specifications and test oracles to detect them.

Property-based testing can help detect those vulnerabilities by expressing key functional specifications of systems in a few properties. Property-based testing has the advantage that it does not require writing complete program specifications to be effective. Few key properties are enough. Property-based testing has been applied to several systems and programming languages, including smart contracts. It requires developers or auditors to provide specifications of their code using a specification language. These languages usually focus on simple logical patterns. Our idea is to use property testing to check functional requirements of smart contract systems at runtime. This can help developers detect business logic vulnerabilities in smart contracts by catching discrepancies between code execution and specified functional properties. Traditionally, developers detect these kinds of bugs using example-based testing, i.e., by writing unit tests: test cases that exercise parts of the code on given inputs and compare the output of the code with the expected output. By contrast, property-based testing takes these concrete scenarios and generalizes them by focusing on which features of the scenario are essential and which are allowed to vary. Property-based testing is powerful as it enables to specify exactly under which conditions the output of a function can be considered valid and by extension, to detect when the output deviates from specifications.

As of today, blockchain usage is mostly split across three major use cases: Decentralized Finance protocols (DeFi protocols), Decentralized Anonymous organizations (DAOs), and Non-fungible tokens (NFTs). Unfortunately, as blockchain usage increased, systems from all of these categories were attacked. The attack vectors vary from protocol to protocol but some regular patterns remain the same. Attacks can occur at several layers: at the decentralized application (dapp) level, both on the frontend and the backend, at the smart-contract level, on the blockchain itself, or at the user level, usually stealing private keys with malware or phishing. In this thesis, we only focus on the smart-contract level. As of February 2022, according to the REKT database, more than $3B funds were lost in smart contract hacks. Most of these hacks are due to bugs such as improper access control checks, price oracle attacks, and other business logic vulnerabilities. For instance, BzX suffered two consecutive hacks on Feb. 21, allowing the attacker to open under-collateralized positions because of an improper position collateralization check and a price oracle attack. A few days later, Alpha Finance suffered a $37.5M hack in February 2021 because of a bug enabling users to open undercollateralized loans. A few months later, in October 2021, CREAM finance suffered a $120M loss because of a price oracle attack. Governance smart contracts also suffered attacks: in October 2020, attackers abused the MakerDAO governance smart contract and made a proposal pass using MKR tokens borrowed using a flash loan. It clearly shows that these attacks involving business logic vulnerabilities can lead to dramatic consequences in the Blockchain ecosystem. That is why addressing such vulnerabilities before dishonest attackers exploit them is crucial. By leveraging property-based testing tools, one can detect this kind of business logic vulnerabilities and fix them accordingly.

Related work Security testing of smart-contract systems is extensively studied [61][24][51][45]. In the past few years, researchers have developed a variety of tools to detect vulnerabilities in smart-contract systems automatically. These tools can be divided into two categories: static analysis and dynamic analysis tools. Static analysis tools operate without executing the program by examining the source code, byte code, or application binaries. In contrast, dynamic testing tools aim to detect bugs while running the program. They usually examine the program in its running state and try to manipulate it to discover bugs. These tools can efficiently detect bugs with known patterns such as integer overflows, reentrancy vulnerabilities, or unprotected ether withdrawal. However, they cannot detect bugs related to the program's business logic by themselves. These business-logic bugs can be detected by proper unit testing, manual code audit, formal verification methods, and property-based testing. Formal verification proves the correctness of the system by checking the system against its specifications. Formal verification guarantees that the system strictly follows its specifications. As opposed to testing which demonstrates the presence of bugs, formal verification aims at proving that system is exempt from certain bugs. Successful formal verification of a system provides high security guarantees. However, formal verification tools are subject to memory and runtime limitations, so they are generally unable to check complete system specifications. Also, they require an extensive definition of program specifications in a formal language expressing logic predicates such as Hoare logic or linear temporal logic. Formally defining such specifications is a difficult and time-consuming task. It requires a full understanding of the system and specific knowledge of logic and formal language. This is why property-based testing is useful. It can be seen as a middle-ground between testing and complete formal verification. A successful property-based testing approach does not require to specify complete specifications to be useful. Instead, it requires the user to specify a few key functional properties aiming to capture most of the functional specifications of the system. Property-based testing tools have already been developed for smart contract systems [48][19]. These tools rely on test harnesses or instrumentation of the source code, which can then be checked statically using techniques like symbolic or concolic execution or at runtime through fuzzing. Unfortunately, instrumenting smart contract source code is not always possible, for instance, when monitoring deployed systems or when sources are not available. Our approach relies on an external checker that does not need to instrument the source code.

Method We start by studying popular smart-contract systems running on the Ethereum blockchain along with smart contracts that were previously attacked. We derive functional properties and annotate them using Scribble [48], a smart-contract specification language. This provides us with insights about properties, i.e., invariants and pre/post-conditions that should be monitored at runtime to ensure proper functionality of the system. Inspired by Scribble, we proceed by developing a smart-contract specification language to express those properties. Our specification language can express both contract invariants and call-specific post-conditions and error pre-conditions using if_succeeds and if_aborts annotations. Our language needs to be checkable at runtime on deployed contracts and thus, does not rely on any instrumentation. Existing Scribble instrumentation performs source code rewriting. However, this process is impossible to do for systems already deployed or systems for which we don't have the sources. It also creates side effects such as increased gas usage that can alter program execution. Our intuition is that writing and checking higher-level properties at the Application Binary Interface (ABI) level still captures interesting specifications without adding as much overhead. We design a low-overhead runtime checker that replays Ethereum transactions and checks for property violations. Our checker does not need any program instrumentation meaning it can work with any contract and check properties on systems deployed on production and test chains. Our goal is to capture vulnerabilities in the business logic of the systems that would enable an attacker to exploit the system. Our checker is not dependent on any particular way of generating the sequences of transactions. Transactions can be artificially generated by fuzzers or directly read from blockchains. This allows us to monitor functional properties on production and test chains continuously. Eventually, this enables us to detect business logic vulnerabilities and mitigate them if possible. Finally, we evaluate the efficiency of our tool by running it on different smart contracts.

We focus on answering the following research questions:

- What kind of functional properties are important to capture in smart-contract systems?
- How to design a specification language that expresses blockchain-specific smart contract properties that can be checked at runtime without instrumentation?
- How to design a low-overhead checker that checks blockchain-specific functional properties at runtime on non-instrumented code?
- Is it possible to monitor and mitigate vulnerabilities at runtime and limit the impact of a potential exploit on smart contract systems?

We also make the following contributions:

- We introduce *Jessy*, a specification language built on top of Scribble that expresses properties that can be checked at runtime without instrumentation.
- We present a checker that monitors transactions at runtime and checks for property violations.
- We annotate existing smart contract systems and evaluate our tool, demonstrating how it contributes to improving smart contract security.

Chapter 2

Background

2.1 Ethereum and the EVM

Ethereum is a cryptographically secure, transaction-based state machine initially described in a white paper published by V. Buterin in 2014 [7]. The design was then specified and formalized by G. Wood in the "Ethereum yellow paper" [58] describing the *Ethereum virtual machine* (EVM) for the first time. Ethereum incrementally executes transactions inside the EVM, a *quasi-Turing complete stack-based virtual machine*[1], to make changes to its global state also called *world state*. Ethereum world state is a mapping between addresses (160-bit identifiers) and account states, a serialized data structure that has a balance, a transaction count (nonce), and eventually an EVM bytecode and a storage state.

Informally, the terms *smart contract* and *contract* are commonly used either to refer to the piece of EVM bytecode associated with an account or to an *Autonomous Object*[2] as defined in the Ethereum Yellow paper. In Ethereum, a piece of EVM bytecode is associated with an account. The code is responsible for controlling the state of this particular account. For clarity, we refer to such an autonomous object as a *contract account*. In contrast to contract account, we use the term *externally owned account* (EOA) to refer to an account controlled by anyone possessing the account's private key. In the state trie, an EOA is represented similarly to a contract account except that the bytecode and the storage trie associated with the account object are both empty.

A transaction is a serialized data object that contains a sender, a recipient, a value corre-

[1]We cannot qualify Ethereum as a Turing-complete machine as the computation are bounded by a parameter called "gas" limiting the maximum amount of computations done, to avoid infinite recursions that would lead to a DoS attack on the blockchain.

[2]An autonomous object is a notional object that has an intrinsic address, and thus an associated account. This account has a non-empty EVM bytecode. Like any account, it also has a storage trie, a balance, and a nonce.

sponding to the amount of Ether sent, a gas limit, a gas price and optional data passed as a byte string. A transaction is passed between two contract accounts, two EOAs or an EOA, and a contract account. Transactions are used to interact with contracts by invoking their functions. Functions defined inside contracts can perform arbitrary computations, call (invoke) functions from external contracts, and eventually return a value. A transaction with empty data but a strictly positive value moves funds from one account to another. A valid transaction with non-empty data either calls a function from a contract or creates (deploys) a new contract. Gas value depends on the amount of computation that needs to be done to process the transaction. Transaction execution benefits from two key properties namely *atomicity* and *determinism*. Atomicity guarantees that a multi-step transaction either succeeds or reverts. Contracts may contain instructions to revert transactions if a condition is not satisfied during its execution. Determinism guarantees the absence of randomness in the state transition when executing the transaction and thus, a unique transaction outcome for a given blockchain state.
When interacting with each other, contracts can use several opcodes that correspond to different types of calls with different semantics. The primarily used opcodes are CALL, DELEGATECALL, and STATICCALL [58]. The CALL opcode executes a standard call to a contract account similarly to any other EOA account. DELEGATECALL executes the code of another account in the context of the current account. Finally, STATICALL executes a call to another contract but disallows any modification to the state (and reverts in that case).

2.2 Solidity

Most smart contracts are written in Solidity; a Turing-complete statically typed object-oriented language. The Solidity compiler compiles source code to bytecode that can be deployed on the blockchain and executed by the EVM. Smart contracts can inter-operate by calling functions from each other. This enables composability of contracts to augment the functionalities of single contracts. Below we describe the general structure of solidity contracts [11].

Contract structure A Solidity contract is composed of *state variables, functions, function modifiers, events, errors, struct and enums.* State variables are stored in contract account storage. This storage is permanent and part of the global blockchain state. Functions and variables can have different visibility. Depending on their visibility, they can be called (resp. invoked) from external accounts, derived contracts (we describe contract inheritance in the next section), or only the contract in which they are defined. Modifiers are pieces of code that can be used to augment the behavior of functions. Solidity provides a default set of modifiers and enables the user to implement custom modifiers. For instance, one can add the *view* or *pure* modifiers to functions that do not modify (resp. read or modify) contract state. Similarly, one needs to mark a function as *payable* so that it can receive Ether. Solidity defines two special functions: the *receive function* and the *fallback function*. The receive function is executed on a plain ether transfer to the contract, i.e., on a transaction with a positive value and empty calldata. The fallback function is executed if a call to the contract does not match any function signature or if the transaction has empty data, but no receive function is defined. Events are used to leverage EVM logging capability, and errors can be used to provide additional details to revert statements. Solidity also supports defining structs and enums.

Storage The EVM can access four memory areas: *storage, memory, calldata* and *stack*. The storage layout of a Solidity contract directly maps to its EVM storage layout. In several cases, data locations of the variables need to be specified to the Solidity compiler using data location keywords. The *storage* is a word-addressable memory area that persists across contract calls. It is a key-value mapping with 256-bit slots. It stores all the state variables of a solidity contract, and every variable declared as a "storage" variable. It is allocated by the EVM when a contract is deployed and is part of Ethereum global state. Account storage is governed by the code of the account and can eventually be written by other bytecode during a DELEGATECALL or a CALLCODE. The *memory* stores temporary values for the scope of a given call. However, it is wiped after each call. The *calldata* is a byte-addressable read-only memory. It is allocated by the caller and contains the input of the call, i.e., the function selector and the arguments passed to the function. Reading from this memory section requires specifying the exact number of bytes to read along with the position (offset) to start reading from. The *stack* holds small local variables of size at most 256 bits. It is limited in size and cannot store complex structures. The EVM stack has a maximum depth of 1024 slots and is used for local computations. However, it can access directly only the top 16 slots.

Library Solidity introduces the concept of *libraries*. A library is similar to a contract, with the exception that it can be deployed only once at a specific address on the blockchain. Contracts can then *DELEGATECALL* into library functions, meaning the code of a function defined in the library is executed in the context of the caller contract. A library can thus modify the storage of the calling contract. Libraries can also contain internal functions meaning a contract calling these functions will follow the "internal" call paradigm, i.e., invoking the function using a JUMP instead of a CALL. To make this possible, the Solidity compiler inlines every internal library function in the contract at compilation time. Unlike contracts, libraries cannot have state variables (they must be executed in the context of the calling contract using DELEGATECALL), cannot receive Ether, cannot self-destruct, and do not support inheritance.

Interface Interfaces instruct the Solidity compiler on how to invoke functions from external contracts. Solidity compiler assumes that all contracts will have the interface definitions of any external contract called available at compilation time. An interface contains a list of function definitions corresponding to external or public functions of a contract callable from external contracts. All functions declared in an interface must have *external* visibility even if the corresponding function implemented in the contract are *public*. That is because functions defined in interfaces must be called using the "external" calling paradigm. Solidity interfaces support inheritance.

Application Binary Interface (ABI) ABI stands for application binary interface. There exists a one-to-one mapping between a Solidity interface and the contract ABI. The ABI describes the encoding needed to call functions of a contract and read the returned data. The ABI is mandatory to interact with a contract but is not stored on-chain. Transaction data encoding is not self-describing, meaning a Web3 client requires the schema to encode and decode data. One must note that ABI abstraction is not part of the core Ethereum protocol. A smart contract developer can decide to define their own ABI. Any user interacting with the contract would have to comply with that particular ABI. Yet, for practical and interoperability reasons, the Solidity compiler and Ethereum client libraries agree on an ABI standard represented by a JSON object. It contains an array of function descriptions, event descriptions, and error descriptions. Thanks to these descriptions, client libraries can encode transaction data accordingly.

Smart contracts and object-oriented programs exhibit many similarities. Consequently, most concepts from object-oriented programming theory can be adapted to smart contracts. This is useful as object-oriented programming theory has been studied extensively since the 80s [38][6][50]. It is based on four core concepts: *encapsulation, abstraction, inheritance, and polymorphism*. The following section demonstrates how these concepts apply to Solidity contracts.

2.3 Solidity contracts from an object-oriented perspective

Encapsulation and abstraction In object-oriented programs, we group semantically related data in data structures called *objects*. This core concept of object-oriented programming is called encapsulation. A *class* is a blueprint used to construct objects. It describes instance members of the object: its instance variables and its methods. The program can send a message to an object asking the object to invoke one of its methods without knowing the details of the implementation. This concept is referred to as *abstraction*. Similar to a traditional software class, a Solidity contract contains members, i.e., data fields and functions that can either be publicly accessible or restricted to the contract in which they are defined and, eventually, derived contracts.

Inheritance and polymorphism Other core concepts of object-oriented programming are *inheritance* and *polymorphism*. Inheritance enables a class called *derived class* to reuse and extend components from a *base class*. We say that the derived class extends the base class. Depending on the language, a class can extend one (single inheritance) or several (multiple inheritance) base classes. Inheritance is transitive. Polymorphism supports inheritance by enabling objects created through derived classes to be treated as objects of the base class. Consequently, one can invoke a method of the base class in the source code, causing the version defined in a derived class to be executed at runtime. Indeed, a derived class extending a base class can exhibit multiple behaviors: the derived class may either override methods from the base class, redefining their implementation, or inherit methods from the closest base class that defines them without overriding them.
Solidity contracts support multiple inheritance and polymorphism. Contracts can inherit the behavior of multiple other contracts. Because of polymorphism, a contract will always execute the method defined in the most derived contract in the inheritance hierarchy. This behavior is similar to most object-oriented languages. The Solidity compiler requires the programmer to make this behavior explicit by mentioning in the function signature that this function overrides one or several functions defined in the parent contracts. Solidity inheritance is fully static and resolved at compilation time. When a contract inherits multiple contracts, only a single contract is created on the blockchain: the code of every base contract is compiled with the created contract code. It enables the EVM to use JUMP statements to invoke functions from base contracts instead of CALL statements used to invoke external contract code.

Interface Object-oriented programming also introduces the notion of an *interface*. An interface contains definitions for a set of related properties that a class must implement. It can be seen as a description of the actions an object must support, i.e., a contract that states what functionalities one can expect from a class implementing the interface. A class implementing an interface must provide an implementation for every method declared in the interface. Like classes, interfaces can also be extended by derived interfaces. Solidity also enables the definition of interfaces that are very similar to general object-oriented program interfaces. The only difference is that Solidity interfaces can only declare external functions i.e., functions that can be called by external accounts. Interfaces in Solidity are important as they are required by contracts from different accounts to interact with each other.

Thus, Solidity contracts are very comparable to classes of object-oriented programs. Solidity compiles every contract into a single bytecode executed on the EVM. When a contract is deployed, the EVM creates an instance of the contract and associates it with an address on the blockchain. This address can be used by EOAs or other contract accounts to interact with the contract. Any EOA or contract can deploy a new contract with a *contract deployment transaction*. A contract deployment transaction is a transaction whose recipient is the zero address, and the data contains the code needed to initialize the contract. This code, generated by the Solidity compiler, does two things: first, it runs the contract's constructor to initialize the storage of the contract account. Second, it copies the rest of the bytecode (i.e., without the constructor) in the bytecode storage of the account. From an external point of view, a contract looks similar to an immutable class whose instance members are all private but with public getter methods defined for each public storage variable. However, this analogy breaks when contracts interact using DELEGATECALL or CALLCODE opcodes. Assume contract A maps to class A and contract B maps to class B, a DELEGATECALL from contract A to contract B can be seen as an unchecked typecast of class A to class B. The EVM delegates the control of A's storage to B's code. This is possible because of the slot-based storage mechanism of the EVM.

2.4 Smart contract vulnerabilities

Like other computer programs, smart contracts are vulnerable to traditional software bugs such as arithmetic and control-flow bugs. However, thanks to the improvement of the Solidity compiler and smart-contract security tools, the frequency of these types of bugs decreased drastically to the point where they are now much rarer. Most of the bugs in production systems nowadays are bugs specific to blockchain and smart contracts or business logic bugs. Several works aim to classify smart-contract vulnerabilities [61][24][51]. The SWC registry[3] provides an overview and a general classification of common security issues discovered in smart contract systems. However, as the ecosystem grows, exploits are increasingly complex and rely on several nested first-order vulnerabilities or business-logic exploits. For instance, several DeFi protocols have been attacked because of so-called *oracle attacks*. These kinds of attacks consist of manipulating the price of one or several assets in a protocol to influence the logic of another protocol dependent on the price of these particular assets [43]. Because of DeFi primitives like *flashloans*, such economic attacks are accessible to anyone without needing large funds. They are the consequence of business-logic vulnerabilities present either in single protocols or emerging from the composition of several protocols.

2.5 Properties

Functional property checking is a discipline that focuses on detecting bugs using partial formal specifications, also known as properties. These properties are expressed in *specification languages*, for instance in the form of *invariants* and *function pre-conditions and post-conditions* (inspired by the *design by contract paradigm* [28]). This paradigm stipulates that programs

[3]Smart Contract Weakness Classification Registry (`https://swcregistry.io`)

should be designed according to formal and precise interface specifications, which extend the definition of abstract data types with *pre-conditions, post-conditions* and *invariants*. This defines the *contract*[4] of the software component, i.e., a set of specifications that should be satisfied by the component. For clarity, we refer to this contract as *specifications* of the smart contract. Specifications are made of *properties* that a correct implementation must satisfy. Properties allow describing what a correct function or contract should be doing.

[4]not to be confused with "contract" referring to smart contract

Chapter 3

Design

Our approach to specifying properties takes inspiration from the *design by contract* paradigm. We check smart-contract properties at runtime by replaying transactions and capturing violations. Any verification technique that checks business logic properties requires the programmer to add some kind of annotations to their code. However, if the effort it takes to annotate the program exceeds the benefits guaranteed by the verification techniques, or the effort required to write the code in the first place, then programmers will not invest time to write such specifications [29]. Thus, annotations should remain simple, e.g., in the form of simple function pre-conditions, post-conditions, and contract invariants. Such properties enable developers to make their intent and assumptions explicit and provide clear and effective documentation for the code.

In addition to developers, auditors also directly benefit from effective automated verification techniques. Auditors usually deal with different codebases on a daily basis, which is often complex. Thus the need for verification techniques that are easy to use with a low setup overhead. As protocol complexity increases, security tools need to handle such complexity effectively. Our approach to contract verification only relies on the contract's external interface. This introduces some constraints on the properties our tool can express. For instance, without complex EVM instrumentation and the contract's source code, we cannot express properties of internal functions. However, our approach also comes with benefits: as we do not require any instrumentation of the contract, the setup overhead is low. Our tool only requires the programmer or the auditor to provide a set of properties expressed over the contract ABI along with the address of the deployed contract to monitor. It can be useful both in a Forensic context, by replaying previous transactions from the chain, or as a just-in-time monitoring tool.

3.1 Specification Language

To express properties, we design *Jessy*, a *specification language* inspired by the Scribble language [48], an open-source specification language designed for smart contracts created by Consensys Diligence.

A common critique of formal methods is that they are hard to define and difficult to understand [25]. To address this critique, we focus on making our language as easy and straightforward to use as possible with existing Solidity experience. The key is to make our specification language accessible to users that are not trained in formal methods.
Each property expressed in Jessy consists of three components:

1. A keyword defining the semantic of the property.
2. A message identifying the property
3. A condition, i.e., a Solidity-like Boolean expression that defines correctness of the property terminated by a semicolon or alternatively, an arbitrary block of solidity code that returns a single Boolean value[1].

```
//single-line property
```

$$\underbrace{\texttt{if\_succeeds}}_{\texttt{keyword}}\ \underbrace{\texttt{"a is updated"}}_{\texttt{message}}\ \underbrace{\texttt{old(this.a()) != this.a();}}_{\texttt{condition}}$$

```
//multi-line property[1]
if_succeeds "merkle root is not zero" {
    bytes32 root = this.roots(keccak256(_message));
    return root != bytes32(0);
}
```

Listing 3.1: Property syntax

Currently, our tool supports three types of safety properties with different semantics: contract invariants, expressed using the keyword `invariant`, transaction post-conditions, expressed using the keyword `if_succeeds`, and reverted transaction pre-conditions, expressed using the keyword `if_aborts`.

To illustrate our point, we introduce a toy Solidity contract along with three properties, one of each kind, in Figure 3.2. For convenience, we introduce the properties as comments in

[1]We introduce multi-line properties as a workaround as we do not support defining arbitrary immutable variables inside of an annotation yet. This is useful to unpack results of arbitrary function calls and avoid code duplication.

the contract's source code. However, note that the whole purpose of our tool is that it is not mandatory to have access to the contract source code. The contract ABI is sufficient. The first property (l.2) is a contract invariant constraining the value of the public state variable `a`. The second property (l.17) is an if_succeeds property (a post-condition of the function `kind`) relating the pre-state and post-state values of the state variable a with the return value of the function. The third property (l.25) is an if_aborts property (a pre-condition of the function `mean`) in case the call to the function reverts.

The condition of a property p is an evaluation function

$$\begin{cases} f_{c_{pre}}^{c_{post},\mu,\sigma} : expr \longrightarrow \{true, false, \emptyset\} & \text{if } p \in \{invariant\} \\ f_{c_{pre},S_{pre}}^{c_{post},S_{post},\mu,\sigma} : expr \longrightarrow \{true, false, \emptyset\} & \text{if } p \in \{if\_succeeds, if\_aborts\} \end{cases} \tag{3.1}$$

where c_{pre} (resp. c_{post}) is the *publicly observable state* of the checked contract before (resp. after) the call, S_{pre} (resp. S_{post}) is the set of all publicly observable states of deployed contracts before (resp. after) the call, μ is the context of the current transaction *tx*: $\mu = (tx.from, tx.to, tx.value, tx.input, tx.output, tx.error)$, and σ is the context of the chain accessible from a Solidity contract [55]. We define the *publicly observable state* of a contract as the set of all its public view/pure functions that return at least one argument and its account state (address, account, balance).

The expression of a condition is a typed Boolean Solidity expression or a block of Solidity code that returns a Boolean expression which may, only in the case of a call post-condition, be augmented with the *old* operator. It allows referring to the pre-state value of any function that is part of the publicly observable state of the checked contract, or another arbitrary contract. A valid expression may refer to one or several *legal variables or functions* in the scope of the property. For an `invariant` property, it can be any function that is part of the publicly observable state of the checked contract along with a subset of global variables accessible from Solidity[2]. Any reference to the publicly accessible state of the checked contract must be prefixed with `this`. In addition to that, `if_succeeds` properties may contain references to function input and output values, pre-state values of functions part of the publicly accessible state of the checked contract (using the *old* operator), and transaction parameters. A legal function augmented with the `old` operator constitutes an *old* expression. `if_aborts` properties may reference function input values, pre-state values of functions part of the publicly accessible state of the checked contract, error message eventually thrown by a revert statement, and transaction parameters. Finally, both `if_succeeds` and `if_aborts` properties can refer to the publicly accessible state of arbitrary contracts, both in the pre and post states of the call, as long as the interface to call the referred contract is defined. In that case, as in a standard external Solidity call, one must prefix the function call with a cast of the contract address to the function interface. When

[2]Variables that refer to the chain context

```
contract Bar {
    // invariant "invariant: a is 51" this.a() == 51;
    uint256 public a;

    constructor() {
        a = 51;
    }

    function increment() private {
        a += 1;
    }

    function decrement() private {
        a -= 1;
    }

    // if_succeeds "kind: a is still the same" old(this.a()) == a_
       && this.a() == a_;
    function kind() public returns (uint256 a_) {
        //invariant not violated
        increment();
        decrement();
        a_ = a;
    }

    // if_aborts "mean: a is 53" this.a() == 53;
    function mean() public {
        // violate invariant
        decrement();
        decrement();
        require(a != 51, "a is not 51");
    }
}
```

Listing 3.2: Solidity contract annotated with three properties

referring to functions returning multiple arguments, one can access the nth argument returned by the function by adding the `._n` suffix to the function call.

As under the hood, conditions are typed Solidity expressions, it means that every logical operator, equality predicate, and numerical expression evaluates similarly as in Solidity. It also means that the ordering of sub-expressions inside a condition might alter the semantics of the condition as opposed to general first-order logic expressions. This is because of the short-circuiting rules of boolean operators in Solidity: in the expression `f(x) || g(y)`, if `f(x)` evaluates to true, `g(y)` will not be evaluated even if it may have side-effects. For instance, `(this.a() == 0) || (this.b()/this.a() == 1)` is valid if `this.a() == 0` whereas the commutative expression `(this.b()/this.a() == 1) || (this.a() == 0)` will error. A valid condition must be well-defined and must evaluate as true or false. Because we evaluate conditions directly inside the EVM, evaluating a condition that is not well-defined will throw an error. For instance, this is the case for a condition triggering a division by zero. In that case, the tool throws a property valuation error.

3.2 Checker contract

To evaluate the properties, we rely on a Solidity contract that we generate automatically. We refer to this contract as the *"checker contract"*. Relying on a solidity contract to evaluate properties removes the need to write an interpreter for the specification language. This brings several benefits: first, we can support most Solidity expression out of the box. Second, properties are evaluated by the EVM itself meaning we do not have to worry about mirroring its semantics. Third, we do not need to maintain our interpreter if the Solidity language evolves. We only need to upgrade the compiler used within our tool to support new expressions. It reduces implementation time and effort.

We generate the checker contract using the contract ABI annotated with the properties provided by the user. In Figure 3.3, we show the checker contract that our tool generates for the toy contract in Figure 3.2. For each invariant, we generate a single function that evaluates its condition. For if_succeeds properties, we generate two functions: the first one, *if_succeeds_pre*, evaluates the pre-state values of the old expressions before the function call while the second one, *if_succeeds_post*, evaluates the condition of the property after the call to the function returns. We also pass the transaction context along with the pre-state values of the old expressions as arguments to the "post" function as needed to evaluate the condition of the property. For if_aborts properties, we generate a single *if_aborts* function with transaction and error context as parameters that evaluates the condition. In the next section, we provide additional details about the semantics of the different properties. All the functions we generate in the checker contract are *view* functions meaning they are not allowed to modify the state of the checker contract. Additionally, we use static calls to invoke every function of the checker contract. This ensures it is not possible for a property to alter the state of the checked contract in which case checking the property throws an error.

3.3 Language semantics

In this section, we detail the semantics of contract invariants and call properties. Each property is associated with its own semantics that defines at which execution point the property is expected to hold.

3.3.1 Contract Invariants

The first type of property is *contract invariants*. Contract invariants are similar to *program invariants* [21]. Invariants specify properties that must hold at every point in time the contract is observable. They typically restrict and/or relate the values of some storage variable of the contract.

According to Leino and Müller [26], "It should be possible to reason about smaller portions of a program at a time, say a class and its imported classes, without having access to all pieces of code in the program that use or extend the class". Our design follows this advice and focuses on single-contract invariants. Dealing with multi-contract invariants brings a lot of design complexity. Moreover, most multi-contract invariants can be refactored and expressed using single-contract invariants.

Most useful invariants should not hold in every state of a contract. That is because instructions are executed sequentially; otherwise, no computation would be possible. Take the invariant presented in Listing 3.2 that constrains the value of the state variable a. Assume a call to the function $bar.kind()$. This first invokes the function $increment()$ (l.20) effectively incrementing the value of the state variable a to 52. But then, directly after the call to $increment()$ returns, the function $decrement()$ is invoked, decreasing the value of a by one and reestablishing the invariant before the first call to $bar.kind()$ returns. This means that the invariant constraining the value of a is temporarily violated in between the calls to the functions $increment$ and $decrement$. However, the invariant is restored before the top-most call into $kind()$ returns.

That being said, we need to precisely define when an invariant must hold and when it is allowed to be violated. A common object-oriented paradigm for checking invariants is the *visible states* semantics [31]. It enforces stronger requirements than other semantics such as the observable state semantic [28] but protects against invariants' violation in re-entrant calls. We

```
interface IChecked {
    function a() external view returns (uint256 ret_0);
    function kind() external returns (uint256 ret_0);
    function mean() external;
}

contract Checker {
     struct txParams {
        address sender;
        uint256 value;
    }

    IChecked private _checked;

    constructor(address checked) {
        _checked = IChecked(payable(checked));
    }

    function inv_0() external view returns (bool) {
        return _checked.a() == 51;
    }

    function if_succeeds_pre_0() external view returns (uint256 _a_0
       ) {
        (_a_0) = _checked.a();
    }

    function if_succeeds_post_0(
        uint256 old_a_0,
        uint256 ret_0,
        txParams calldata params
    ) external view returns (bool) {
        uint256 a_0 = _checked.a();
        return old_a_0 == ret_0 && a_0 == ret_0;
    }

    function if_aborts_1(
        string memory error_msg,
        txParams calldata params
    ) external view returns (bool) {
        uint256 a_0 = _checked.a();
        return a_0 == 53;
    }
}
```

Listing 3.3: Checker contract generated for contract in Listing 3.2

use this paradigm to check contract invariants. We define a contract in a *visible state* as follows:

Definition 3.3.1 (Visible state)**.** A contract is in a visible state when it can be called from any other account.

We allow invariants to be violated while the contract is not in a visible state but must ensure they hold whenever the contract is visible. It turns out that a sufficient condition to make sure invariants hold in every visible state is to check them before "exiting the contract". Indeed, "exiting the contract" triggers a context switch inside the EVM, meaning any other contract can call into the contract, expecting invariants to hold. We want to leave the contract in a clean state to prevent any call or reentrant call into the contract while an invariant does not hold. We exit the contract in two cases:

1. After returning from an external call into this contract (this includes "internal-external" calls and transactions)
2. When calling from this contract account to an external account

If we ensure every invariant holds in both cases, we can guarantee it holds in every visible state of the contract (provided it holds after the contract is created, i.e., after the constructor returns). Excepted balance invariants[3], we do not require invariants to hold before the constructor returns. Invariants are only required to hold from the point where the contract is fully created, i.e., after the constructor successfully returns. Because of the EVM semantics, it is impossible to have reentrant calls inside the constructor of a contract. That is because the EVM copies the code of the contract to the account storage only when the CREATE or CREATE2 opcode terminates and the constructor successfully returns. Thus, any reentrant call inside the constructor fails as the code of the account is not yet initialized.

In Solidity, internal function calls are mapped to JUMP instructions at compilation time. It means calling an internal function does not "exit" the contract: no CALL instruction is used and the EVM context is left unchanged. One can force an external re-entrant call from function `f` of the current contract into the same contract by invoking it using `this.f()`. The Solidity compiler will compile such expression to a *CALL* opcode which in turn, will trigger the corresponding instruction in the EVM. In our framework, this will trigger an invariant check. We say this is a *qualified call*. In practice, one should never trigger an external call to a public function of the same contract. However, it is not a problem for our tool. It will trigger a contract invariant check and enforce stricter conditions as an invariant might be temporarily violated.

[3]That is because of the EVM semantics, it is possible to deposit ETH on a contract account before deploying the contract. Thus we have to make sure balance invariants hold at creation time.

Definition 3.3.2 (Qualified call)**.** A call into a contract is qualified for a property check if (1.) There is at least one contract invariant or call property defined for the corresponding function selector or special function (receive/fallback). (2.) The call does not revert.

Theoretically, there should be no need to reestablish invariants when calling into the contract as long as they are reestablished before leaving the contract. When calling functions internally i.e., jumping, the contract is not in a visible state. Additionally, transaction execution is atomic: either it succeeds or reverts. No partial change to the world state is possible. The state is locked while the transaction is executing. This is similar to JML (Java Modeling Language) [54] way of handling helper functions: helper functions are authorized to violate invariants as long as they are re-established before leaving the contract in a visible state.

Unfortunately, the current EVM implementation breaks this semantic by enabling to transfer ETH into a contract without calling into it. It means that the balance of a contract account can change even if the account is not loaded inside the EVM context (i.e., without first calling into the contract). This is possible in the following case: one creates contract A, sends ETH to this contract, and makes this contract self-destruct. When self-destructing, the contract can be instructed to send all its funds to contract B without calling into B. Block creators can also directly reward ETH to any account without calling into the account. Consequently, we need to check every invariant referring to the balance of the account before every call into the contract. Otherwise, the caller may assume a violated invariant. This is also the reason we need to check balance invariants before deploying the contract.

3.3.2 Call specifications: functional specifications

We focus on checking *call specifications*, more precisely, *call post-conditions* and *call error pre-conditions* by adding *call properties.* This differs from classical Hoare pre and post-conditions as we allow for evaluation of properties in case the program does not successfully terminate (if the transaction reverts) by using error pre-conditions. This also slightly differs from usual *function properties* and *function pre/post-conditions* found in most specification languages (JML, Frama-C). That is because we focus on properties that can be expressed over the external interface of the contract. Call properties can only be specified for external and public functions of a contract that we generalize as the *public functions* of the contract. These are the only functions of the contract that can be called from external accounts. We provide two types of call property clauses to specify conditions (1.) if the call into the function successfully terminates (`if_succeeds`) or (2.) if it reverts (`if_aborts`). Note that thanks to the EVM semantic, a call will always successfully return, or revert with an error.

```
contract Bar {
    Foo private _foo;
    bool public guard;

    /// if_succeeds "p1" guess <
         10;
    /// if_aborts "p2" guess ==
       10;
    /// if_aborts "p3" !this.
       guard();
    function doSomething(uint256
        guess) public {
       // ...
        guard = true;
        if (guess < 10) {
            _foo.reenter(guess);
        }
        require(guess != 10, "
           revert");
        guard = false;
    }
    //...
}
```

```
contract Foo {
    Bar private _bar;

    function reenter(uint256 v)
       public {
        //...
        _bar.doSomething(v + 10)
            ;
        //...
    }
    //...
}
```

Figure 3.1: Solidity contracts with a re-entrant call

If_succeeds properties If_succeeds properties specify post-conditions that should hold on normal termination of calls. If_succeeds properties might refer to *pre-state* expressions. However, the property is checked in the *post-state* of the call.

Definition 3.3.3 (Pre/post-state)**.** Consider a qualified call into a function f of a contract. We call the state of the contract after passing all the arguments to f but before executing any code of f the *pre-state* of the call. We call the state when the execution of this call returns the *post-state* of the call.

If_succeeds properties might contain pre-state expressions and can refer to function arguments and return values. We evaluate every pre-state expression before executing the function and use those values to evaluate properties when the call returns. Note that every contract invariant should also hold in the pre-state and post-states of a call. A call (or transaction if initiated by an EOA) might invoke other functions internally and initiate external calls that might trigger re-entrant calls. We only require if_succeeds properties to hold when the call that initiated the property check returns. In the context of a re-entrant call, this means that two instances of the property with different contexts and values might evaluate differently.

Assume the two Solidity contracts shown in Figure 3.1 with a single if_succeeds property whose condition depends on the argument of `doSomething` call. Assume a transaction calling `bar.doSomething(5)`. The call triggers a property check when the transaction returns. Because the function is called with argument $guess = 5$, the property condition evaluates to $(guess = 5) < 10$. However, this function call triggers another call into foo.reenter(guess) which in turn, triggers a re-entrant call into `bar.doSomething(15)`. This causes another property check. This time, the condition evaluates to $(guess = 15) < 10$. The tool evaluates this condition first, as the inner re-entrant call is the first to return. It marks the property as being violated. Then, it evaluates the property of the top-most call as soon as it returns, which holds.
`If_succeeds` properties can also refer to the *pre* and *post* publicly observable state of external contracts by calling external view functions. These calls are evaluated in the pre and post-state of the function call. We provide an example of such a property in Listing 3.4.

```
if_succeeds "withdraw at most MAX_TOKEN from the treasury" old(IERC20(this.
   TOKEN_ADDRESS()).balanceOf(this)) - IERC20(this.TOKEN_ADDRESS()).
   balanceOf(this) < this.MAX_TOKEN();
```

Listing 3.4: if_succeeds property with external view call

`If_aborts` **properties** An EVM call might revert, either because of a contract instruction or because of an execution error (out of gas, wrong opcode...). If_aborts properties specify *error pre-conditions* that should hold if a call into a function throws an error and the transaction reverts. Similarly to `if_succeeds` properties, `if_aborts` properties can refer to the *pre* publicly observable state of external contracts. We evaluate if_aborts properties in the pre-state of the call and only check the valuation if it reverts. Otherwise, if the call does not revert, if_aborts properties are skipped.

For a transaction with nested internal calls, when at least one of the internal calls reverts, the complete transaction reverts but each internal call is responsible for reverting the changes it made to the state. We mirror this semantic when evaluating if_aborts properties: modifications made by the previous internal calls are not yet reverted, and if_abort properties are evaluated in the pre-state of the call i.e., on the dirty state still modified by previous calls. From a practical point of view, take the contracts presented in Figure 3.1. Assume a transaction calling `bar.doSomething(0)`, this in-turn calls `foo.reenter(0)` followed by `bar.doSomething(10)` which reverts. If_aborts properties $p2$ and $p3$ are evaluated after the internal call to `bar.doSomething(10)` reverts. $p2$ holds but $p3$ does not. That is because the modifications made by the call to `bar.doSomething(0)` to the contract state are not reverted yet meaning the boolean variable $guard$ still equals $true$. It is gonna revert to $false$ when the top-most call to `bar.doSomething(0)` reverts.

3.3.3 Correctness of a contract

In our framework, we can define the correctness of a contract as the consistency of the contract with respect to its specifications. By extension, we can define the correctness of a call into a contract.

In our framework, a contract is correct if:

1. Its constructor satisfies:

$$\{[\![INV_c]\!]_{c.balance}\}\ body_c\ \{INV_c\} \tag{3.2}$$

   i.e. assuming the balance invariants $[\![INV_c]\!]_{c.balance}$ are correct before the constructor is invoked, a correct contract, once constructed successfully, satisfies all its contract invariants INV_c.

2. All its public functions preserve the contract invariants and satisfy their post-conditions $post_c$ and error pre-conditions pre_c^{err}.

$$\{INV_c \cap pre_c^{err}\}\ body_f\ \{INV_c \cap post_c\} \tag{3.3}$$

We say that a contract is *partially-correct* with respect to its call specifications if every CALL (i.e. transaction) into this contract seen so-far[4] satisfies the correctness requirements expressed by the contract invariants and the call post-conditions and error pre-conditions. Proving a contract is correct with respect to its call specifications means proving every possible transaction to this contract satisfies the previous rule. As there is an infinite number of transactions, it is impossible to provide such guarantees by fuzzing or just monitoring. Thus, our definition of *partial correctness.* We focus on runtime monitoring the partial-correctness of the contract by monitoring every CALL inside the contract.

A call into function f of a contract c is correct with respect to the contract specifications if it satisfies the following rules:

$$\begin{cases} \{[\![INV_c]\!]_{c.balance}\}\ r, err \leftarrow CALL(c, f(x))\ \{INV_c \cap post_f(x,r)\} & \text{if } err = 0 \\ \{[\![INV_c]\!]_{c.balance}\}\ r, err \leftarrow CALL(c, f(x))\ \{pre_{err}^c(x,r)\} & \text{if } err = 1 \end{cases} \tag{3.4}$$

where f is a function of contract c taking formal arguments x and returning values r (or an error message r in case the call reverts), err indicates if the call reverts, INV_c is the set of contract invariants of contract c and $[\![INV_c]\!]_{c.balance}$ is the set of all invariants containing a reference to the balance of contract c. This summarizes the semantics we described above.

[4]from the genesis block to the tip of the blockchain

Finally, our framework also enables us to define the notion of *weak invariants*, and by extension, *weak call properties*. We summarize them as *weak properties*.

Definition 3.3.4 (Weak property)**.** A *weak property*, i.e., *weak invariant*, or *weak call property* is a property that holds only for part of the lifetime of the contract, i.e., from block number α to block number β, with $\alpha \in \mathbb{N}$, $\beta \in \mathbb{N} \cup \{\infty\}$ and $\beta > \alpha$.

This notion is only useful for purely monitoring purposes. For instance, when dealing with proxy contracts, where the implementation code can be arbitrarily upgraded.

Chapter 4

Implementation

Our tool works with every contract deployed on Ethereum. It requires a contract address along with a list of properties to monitor. It leverages the EVM tracing feature to monitor property violations at runtime.
We design our tool such that we do not need to sync a complete Ethereum archive node to replay transactions and monitor properties. Instead, the tool fetches the data needed to execute the transactions to replay depending on the monitored contract. Given an annotated ABI with user-provided properties, we automatically generate the *checker contract* used to evaluate the properties.

4.1 High-level design

The tool is composed of four components: a *fetcher/analyzer* in charge of fetching and analyzing past blocks and transactions from Etherscan API and Infura archive nodes to determine which blocks are relevant to replay, a *simulator* in charge of replaying the transactions and monitoring the properties on a custom EVM implementation that lazily fetches Ethereum state as needed, a *parser* in charge of parsing the properties provided by the user in a YAML file and a *generator* that creates the Solidity checker contract depending on the properties. We also include a simulator mode that enables us to deploy arbitrary contracts provided in a plaintext file along with its annotated ABI, and send arbitrary transactions to the contract. This is useful for testing and debugging property checks. We show the architecture of the tool in Figure 4.1.

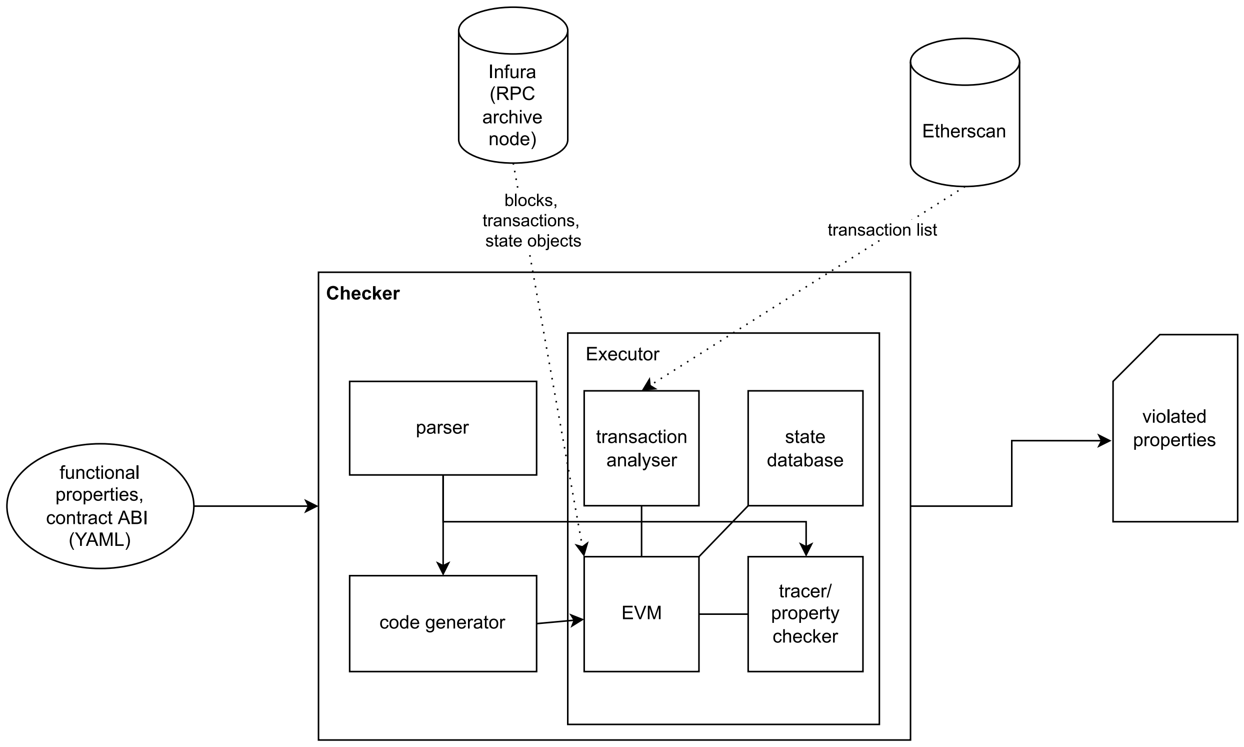

Figure 4.1: Architecture of the tool

4.2 Parser and checker contract generator

First, the checker tool parses the YAML file containing the Jessy properties and if required, external contract interfaces, and generates the associated checker contract. As an example, we provide the YAML file associated with the toy contract *bar* in Figure 4.2. The parser is simple and relies mostly on regular expressions. This creates limitations such as the impossibility of writing properties with nested arbitrary expressions in old operator. In practice, this limitation can be easily bypassed by manually propagating the old operator so that it only contains legal variables to evaluate before the function call. Parsing invariants is straightforward as invariants cannot contain old variables or transaction-specific variables. Parsing function property is more complex as they might contain old expressions, transaction-specific variables, function arguments, function return values, and external view function calls. First, we parse the function declaration to retrieve the function selector, the arguments, and eventually, the return values. Then, we parse the properties, extract the old expressions, the transaction-specific variables, the function arguments and return values and match them with the checked contract JSON ABI as needed. We also extract the view calls to external contracts and match them with the ABIs provided in the YAML file. Then, we generate the checker contract as described in the previous section and the interfaces required to interact with the checked contract and the external contracts. We use regular expressions to prepare properties and generate a plain Solidity file.

```yaml
contract_address: "0xd9145CCE52D386f254917e481eB44e9943F39138"
start_block: 1
invariants:
  - invariant "a is 51" this.a() == 51;
functions:
  - function: a() external view returns (uint256 ret_0)
    properties:
      -
  - function: kind() external returns (uint256 ret_0)
    properties:
      - if_succeeds "a is still the same" old(this.a()) == ret_0 && this.
        a() == ret_0;
  - function: mean() external
    properties:
      - if_aborts "a is 53" old(this.a()) == 53;
raw_json_abi: "..." # checked contract ABI as a JSON string
external_interfaces:
    # (optional) external contract ABIs as JSON strings
    # the YAML key defines the name of the interface used in the
       properties
    # - IERC20: "..."
```

Figure 4.2: YAML file corresponding to contract Bar

We compile this file using solc, the official Solidity compiler, and store the generated bytecode along with the JSON ABI for future interaction with this checker contract. We deploy the checker contract on our EVM so that it can interact with both the checked contract and external contracts, and evaluate the properties.

4.3 Transaction analyzer and simulator

The tool replays past transactions from the blockchain to check properties. There are several ways to retrieve previous Ethereum transactions to replay them. The straightforward way is to spin up a full Ethereum node so that it syncs from the genesis block. Unfortunately, syncing a full Ethereum nodes can take days and roughly requires a terabyte of storage. Additionally, it is impossible to ask the node to replay transactions starting from a given block. Thus, we build our own transaction simulator on top of go-ethereum, one of the three original implementations of the Ethereum protocol [18].

We fetch previous blocks, transactions, and blockchain states from Infura archive nodes to replay them. We customize go-ethereum EVM implementation so that it can replay transactions on top of the Ethereum state at a given block without having to sync the whole blockchain state. We fetch the state required to execute transactions at runtime from Infura HTTP API [22]. The main bottleneck of this approach is performance. To mitigate this issue, we try to batch calls to the API and pre-fetch as much data as possible in parallel while other transactions are executing. Unfortunately, it is not possible to know every slot required to execute every transaction before actually executing them. Similarly, it is impossible to fetch the blockchain state in-between transactions from a given block. It is only possible to fetch the state at the end of a block. Thus, if we avoid doing complex transaction dependency analysis that requires access to additional data not available from archive nodes[1], we have to replay all the transactions of a block to make sure our local state matches the real blockchain state. Additionally, `if_succeeds` properties might also refer to the state of arbitrary contracts. Thus, we need to ensure that the state of our local fork matches the remote state at all time meaning we cannot only replay transactions that we want to monitor. To increase the performance of the tool, we rely on Etherscan API [13] to fetch all external and internal calls to a given contract. We have to rely on an external service as unfortunately, it is not possible to access such information from an archive node or from Infura API. We use this information to discard irrelevant blocks by executing the following algorithm, which returns a set of blocks to execute along with the block to start executing transactions from. That way, we avoid executing blocks that contain only irrelevant transactions. Note that we either execute a complete block (i.e., apply all the transactions in the block) or discard it. We never execute partial blocks.

Algorithm 1: Fetch relevant blocks

```
Input: $from, to$: the block numbers to start/stop fetching transactions
       $address$: the address of the contract to monitor
Output: $executorStartBlock$: blocks number to start the executor from
        $blocksToExecute$: set of blocks to execute
1  $blocksToExecute \leftarrow \{\}$   // set of block numbers
2  $to \leftarrow \max(to, etherscan.lastBlockNumber())$
3  $iTx \leftarrow etherscan.fetchInternalCalls(address, from, to)$
4  $eTx \leftarrow etherscan.fetchExternalTransactions(address, from, to)$
5  // skip blocks before contract creation
6  $executorStartBlock \leftarrow$
     $\max(startBlock, \min(iTx[0].blockNumber, eTx[0].blockNumber)$
7  for $tx \leftarrow iTx \cup eTx$ do
8      $txBlock \leftarrow tx.blockNumber$
9      if $!blocksToExecute.contains(txBlock)$ then
10         $blocksToExecute.add(txBlock)$
11     end if
12 end for
13 return $executorStartBlock, blocksToExecute$
```

[1]We include an algorithm in Appendix B for PoC

Caching stragey

To avoid fetching the same value multiple times from remote archive nodes, our tool relies on a local in-memory cache. It stores state objects that are lazily fetched from the archive node when required. Each state object stores the balance, the nonce, the bytecode, and the storage trie of an account. This approach of selectively pruning blocks to execute requires using a better-caching strategy than naively caching all the values we fetch from the archive node. When executing consecutive blocks, we want to leverage the local cache as much as possible. However, when skipping blocks, we want to re-fetch all the state values from the archive node as they might have been modified inside the transactions we skipped. We need to evict the balance, the nonce, and the storage trie of the account from the cache. Only the bytecode of an account, which is immutable, can be kept in memory. This approach is sub-optimal as we might evict values from the cache that were not touched. However, to our knowledge, there is no simple way of knowing all the storage slots that have been touched in a given block. This makes this caching strategy optimal with regard to the information we can access.

To implement this strategy, we rely on sequential timestamps that we assign to every object in the cache. For each state object, we keep a timestamp that indicates for which block the object was touched (read or modified) for the last time. In addition to that, we keep a timestamp that corresponds to the last block we skipped. Then, we apply the logic described in Algorithm 2 to discard dirty state objects from the cache and re-fetch the values that might have been updated from the archive node.

As we either execute or skip entire blocks, it guarantees that the state of the cache matches the state of the chain at all times. We do not uselessly evict objects from the cache when executing successive blocks. Still, it is sub-optimal as when skipping blocks, we might evict objects from the cache that were not updated and could avoid re-fetching them in an optimal strategy.

Algorithm 2: Caching strategy

```
Input: $O$: state object,
$lastBlockSkipped$: timestamp of the last block skipped
$current$: timestamp of the current block
1 if $O.lastUpdated < lastBlockSkipped$ then
2     $O.invalidate()$  // invalidates the storage trie of the object
3     $O.updateFromRPC()$  // update the object from node
4     $O.lastUpdated = current$
5 end if
6 return $O$
```

A simple solution to drastically increase the performance of the tool would be to sync a local

Ethereum archive node to avoid fetching the state from a remote API.

4.4 Property checking

As detailed in chapter 3, checking properties is a multi-step process. Depending on the type and the semantic of the property, we need to invoke functions of the checker contract and evaluate properties at different execution points of the contract. We rely on go-ethereum EVM tracing capability to trace each transaction [14]. We implement a tracer that creates a callstack, intercepts every call and return from/to the checked contract. Depending on the callstack, it calls back to other routines in charge of calling into the checker contract to evaluate expressions or properties.

Given the semantic detailed in chapter 3, checking invariant properties is straightforward. For each invariant, the tool makes a static call to the corresponding evaluation function in the checker contract. It returns a boolean that indicates if the invariant holds. Violated invariants are stored in the tracer. Checking function properties is more complex. It requires decoding the input of the transaction, evaluating pre-state expressions before the external call, storing their values, and evaluating the properties after the call. As transaction encoding is not self-describing, we rely on the contract ABI provided by the user as part of the YAML file to decode the inputs and outputs of every transaction into the checked contract. Then, we pass them along with transaction parameters like transaction value and sender as arguments to the checker contract method. Pre-state values are stored as part of the callframe and fed back to the checker contract as required to evaluate properties. Violated properties are stored in the tracer.

When the transaction terminates, the tracer output is marshalled into a JSON object and logged. It contains information about the transaction, the callstack, along with all the invariant and function properties violated during the transaction. This is useful to investigate the transaction further.

Chapter 5

Evaluation

To evaluate our work, we select and study past attacks that impacted different protocols deployed on Ethereum. We try to write functional properties using our language, Jessy, that captures the attack. The main goal is to see if we manage to capture interesting properties given our language design decisions. We run our properties for a couple of blocks before and after the attack to check that our properties are violated when the attack occurs. In this evaluation, we focus on answering two questions: first, is our specification language expressive enough to catch hacks that happened in the past? Second, what extensions of the language could be useful?

5.1 Experimental Setup

First, we start by evaluating our language and tool on a set of toy contracts that we construct. We write a few simple properties for these contracts and generate a bunch of transactions both obeying and violating the properties and invariants. Then, we run our tool on those toy contracts to ensure we get the expected results. We proceed by evaluating the language on real-world examples. We compile a list of major Ethereum exploits that occurred in the last two years and led to high monetary losses. We discard all the attacks that are not the consequence of smart contract vulnerabilities but other attack vectors such as compromised private keys, compromised front-end, or user phishing. We ignore any attack that is not due to a vulnerability in a smart contract deployed on-chain. Then, we select a list of eight exploits whose root cause seems related to a business-logic issue in the Solidity code, focusing on recent attacks. For each of these attacks, we start by understanding the protocol and the main specifications by reading their documentation. Then, we carefully study each exploit by reading exploits post-mortems and analyzing attackers' transactions and the code of the vulnerable contracts. Following this, we write functional properties to capture the root vulnerabilities. We set up our tool for each attack with the annotated YAML file containing the properties, the contract ABI, and the external interfaces. We run the tool on transactions before the attack to check that properties hold. We also check that when the attacker transaction occurs, the properties are reported as being violated.

5.2 Omni protocol

Omni protocol [39] is a lending and borrowing protocol deployed on Ethereum. The beta version of the protocol suffered an attack on the 10th of July 2022, where the attacker stole 1300 ETH [56]. The protocol was renamed as *Para Space* shortly after the attack. The attacker used multiple transactions of 150ETH to drain one of the liquidity pools of the protocol. The protocol enables users to lend their NFTs and ERC-20 tokens and use them as collateral to borrow other tokens. Like most lending and borrowing protocols in decentralized finance, the Omni protocol requires loans to be over-collateralized. It means that the value of the collateral provided by the user has to be strictly greater than the total value of the assets borrowed. The ratio between the value of the borrowed assets and the collateral value is known as the *liquidation threshold*. If this ratio increases above a pre-defined threshold, the account is known as *unhealthy* and the position can be liquidated. It ensures that the overall balance sheet of the protocol stays healthy even if the collateral value drops rapidly.

To understand the attack, we analyze one of the transaction[1] used by the attacker. The attacker exploits a *reentrancy vulnerability*. It is one of the most well-known vulnerabilities in smart contracts. It enables the attacker to hijack the control flow of the contract and perform unexpected actions. In Omni's hack, the attacker exploits the reentrancy vulnerability to borrow funds without providing collateral. The blockchain enables new financial instruments: flashloans and flashmints. These primitives enable a contract to borrow many tokens (resp. non-fungible tokens) as long as it returns them before the transaction commits. This makes attacks requiring a lot of funds possible without possessing the capital. The attacker combines the reentrancy vulnerability in Omni's contracts with a flashloan, a flashmint, and an oracle price manipulation to perform the attack. The reentrancy vulnerability is located in the borrowing logic of the protocol, which does not perform any check for reentrant calls. Combined with clever use of the liquidation logic, the attacker exploits the protocol. First, he takes a flashloan of 1000 ETH and 20 DOODLE. Then, he deposits some DOODLE as collateral in the protocol. Next, he borrows funds and removes some of its collateral from the protocol, making the account unhealthy. As the position is unhealthy, one can liquidate it. With a second account, he liquidates his position, recovering his collateral. However, while liquidating his position, he exploits the reentrancy vulnerability enabling him to re-deposit the same DOODLEs (already used as

[1]Transaction hash: 0x264e16f4862d182a6a0b74977df28a85747b6f237b5e229c9a5bbacdf499ccb4

collateral earlier) in the protocol and take a second loan. Because of the reentrancy vulnerability, the attacker tricks the protocol and is allowed to withdraw all its collateral while keeping the funds borrowed. Finally, he returns the DOODLEs to the treasury and reimburses his flashloan, keeping the ≈150 ETH profit. The attacker violates a key security property of the protocol: a user should not be able to withdraw its collateral if it collateralizes a borrowing position. However, this is possible because of the reentrancy vulnerability in the contract.

We demonstrate that our language enable us to express a simple property that captures the bug. We focus on writing properties for the liquidity pool contract [2]. Two core external functions from the pool contract are involved: `borrow` and `liquidateERC721`. We write two simple functional properties, one for the `borrow` function and one for `liquidateERC721`. We present the properties and function signatures in Listing 5.1. The first property states that borrowing should not be allowed without depositing collateral beforehand. The second property states that liquidating a position (either partially or completely) should decrease the size of the position borrowed by the user being liquidated. In the case of this attack, this property fails as when the function `liquidateERC721` terminates, the size of the position borrowed by the attacker increases. The property captures the reentrancy vulnerability.

```
// borrow(address asset, uint256 amount, uint256 interestRateMode, uint16
   referralCode, address onBehalfOf) external
if_succeeds "borrowing requires adding collateral" IERC20(asset).balanceOf(
   onBehalfOf) - old(IERC20(asset).balanceOf(onBehalfOf)) > 0;
// liquidationERC721(address collateralAsset, address liquidationAsset,
   address user, uint256 collateralTokenId, uint256 liquidationAmount, bool
    receiveNToken) external
if_succeeds "liquidation decreases funds borrowed" (liquidationAmount > 0)
   && IERC20(liquidationAsset).balanceOf(user) < old(IERC20(
   liquidationAsset).balanceOf(user));
```

Listing 5.1: Properties for the omniPool contract

5.3 Rari Fuse

Fuse is a product of Rari Capital [46]; a lending, borrowing and yield farming protocol. Fuse enables "pool creators" to create isolated liquidity pools with custom tokens for lending and borrowing assets. Pool creators can decide on every pool parameter: oracle, interest rate curve, collateral factor, etc... Under the hood, Fuse uses forked Compound code. This code has a known reentrancy problem in the `borrow` function which does not follow the *check-effect-interaction*

[2]At address: 0xEBe72CDafEbc1abF26517dd64b28762DF77912a9

pattern [49]. However, this vulnerability cannot be exploited as long as the transferred token does not have a transfer hook. Rari used a version of Compound cToken contracts where ETH payments are sent by calling `to.call.value(amount)`. This makes it possible to exploit the reentrancy vulnerability. The attacker flashloans assets to use as collateral, deposits them into the Fuse pool, borrows ETH, and exploits the reentrancy vulnerability to call the `exitMarket` function, withdrawing all its collateral and bypassing the health factor check as the loan is not yet registered in the contract state. The attacker can repay the flashloan and keep all the borrowed ETH. The attacker drained several Fuse pools by repeating the same process, stealing $\approx \$80M$ worth of assets. We provide snippets of the vulnerable code in Appendix A. Reentrancy attacks are known and well-documented. However, we see that they are still widely used as of now and lead to disastrous consequences for protocols. Fuse attack occurred on April 30th, 2022 [47] in several transactions. The main logic of Fuse is split between two contracts: the *Comptroller* and the *cERC20* (the borrowing pool) contracts. We write a property for the *cERC20* contract of the pool *fUSDC-127* [3] which was drained in the attack. We write a property for the borrow function in Listing 5.2. This property states that if the `borrow` function succeeds, the collateral value deposited before and after the call should remain constant. In the attack, this property is violated by the attacker by exploiting the reentrancy vulnerability. Additionally, we also experiment and write a contract weak invariant (in Listing 5.3). The invariant states that the value of the collateral deposited in the pool is always greater or equal to the total value of the borrowed asset. Otherwise, the pool is under-collateralized. This weak invariant holds before the attack but is violated when the attack occurs at block 14684814. Note that this is a weak invariant that only makes sense from a monitoring perspective. Indeed, this economic invariant should hold in a properly functioning system, but the contract code does not guarantee it will hold. That is because it is the responsibility of arbitrageurs to liquidate under-collateralized positions to prevent the pool from being undercollateralized. Also, such invariant only holds if the oracles that return the prices of assets functions properly. Still, it is interesting to mention from a purely monitoring perspective.

```
if_succeeds "collateral value of the account stays the same" old(ERC20(this.
    underlying()).balanceOf(msg.sender)) == ERC20(this.underlying()).
    balanceOf(msg.sender);
```

Listing 5.2: Property for the `borrow` function

```
invariant "pool is healthy" this.getCash() * this.exchangeRateStored()
    /(1*10^18) >= this.totalBorrows();
```

Listing 5.3: Invariant for the Fuse cERC20 contract

[3] At address: 0xEbE0d1cb6A0b8569929e062d67bfbC07608f0A47

5.4 Audius

Audius is a protocol deployed on Ethereum that provides an alternative to traditional music streaming services by enabling artists to publish and monetize their work by distributing it directly to fans [3]. The protocol has its own token: AUDIO, which enables its owners to vote in the DAO and participate in the protocol's governance. Recently, on July 24th, 2022, the protocol suffered a governance attack. The attacker managed to propose and vote on a malicious proposal that passed and was executed. The proposal sent all the funds from the community treasury to his wallet.

The attacker exploits a bug inside the Proxy pattern used by Audius. Despite being a source of critiques, the proxy pattern is heavily used to make smart-contracts upgradeable. In the case of Audius, a governance contract is used to control upgrades of the protocol. Once a sufficient number of AUDIO tokens is locked to vote on a proposal, the proposal is executed and the contracts are upgraded. For controlling upgrades of the protocol, Audius uses the *AudiusAdmin-UpgradeabilityContract*. It is based on a modified version of OpenZeppelin *UpgradeabilityProxy* contract. Audius developers modify the base contract by adding a "proxyAdmin" variable that stores the address of the governance contract which manages the upgrades. Unfortunately, this leads to a storage slot clash with OpenZeppelin *Initializable* contract that Audius' implementation contracts use. This is a known vulnerability of the transparent proxy pattern [36]. In this case, this enables the attacker to initialize contracts a second time, passing new parameters. This is impossible in a correct implementation as two boolean fields prevent the initializer function from being called multiple times. Because of the storage collision, the value of these two boolean fields was wrong [4]. The attacker modifies the parameters of the staking and the governance contracts, delegating governance of the protocol to himself and abusing the DAO. He delegates 10B AUDIO tokens to himself to vote on his malicious governance proposal. The proposal is accepted thanks to the new erroneous contract parameters.

It is interesting to observe that the first transaction of the attacker failed (at 22:54:37 UTC). The attacker had to submit a second transaction (at 23:10:12), which succeeded. Transfer of the stolen tokens occurred a few blocks later. There is potential to mitigate such attacks with proper monitoring and incident response. This attack is interesting as it shows that it is insufficient to test the system on a per-contract basis. Indeed, in this case, the system failed because of a specification mismatch between the implementation and the proxy contract. The only way to detect such vulnerability is to test the system while fully deployed. We focus on writing properties for the *Governance* contract deployed behind the vulnerable proxy managed by *AudiusAdminUpgradeabilityContract* [4]. We catch this bug using the property P1 in Listing 5.4. This property mirrors a Natspec comment of the `initialize` function. The property states that the staking address should be set separately after a successful initialization, meaning its address

[4]At address: 0x4DEcA517D6817B6510798b7328F2314d3003AbAC

should still be Solidity default value after initialization. The first time the contract is initialized, the property holds. However, when the attacker initializes the contract a second time, the staking address has already been initialized and the property fails. We cannot directly write a property on the boolean fields $initialzing$ and $initialized$ guarding the `intialize` function as they are internal. This attack also demonstrates that adding language and tool support for linear temporal logic is useful. Indeed, we can capture this vulnerability with a temporal property. Assume a `once` operator taking a boolean condition and returning true as long as the expression is true and the function has been called at most once since the last time the condition became true. Then, the property P2 in Listing 5.5 on the initializer function captures the vulnerability in a concise manner.

```
if_succeeds "staking address must be set separately after initialization"
   this.getStakingAddress() == address(0x0);
```

Listing 5.4: Audius: P1

```
if_succeeds "initialize function is called at most once" once(this.
   implementation() == "0x35dD16dFA4ea1522c29DdD087E8F076Cad0AE5E8");
```

Listing 5.5: Audius: P2

5.5 Qubit Finance

Qubit Finance is a lending and borrowing platform originally deployed on Binance Smart Chain (BSC). However, one of the features of the protocol is the X-Bridge. The X-Bridge allows users to deposit and lock ETH or wETH on Ethereum, and mint an equivalent amount of qXETH on BSC to borrow other assets. This is process is known as *cross-chain collateralization*. Unfortunately, there was a bug in the bridge contract deployed on Ethereum. An attacker exploited this logic bug using 16 transactions[5] to mint xETH on BSC without locking ETH on Ethereum [42]. The bug is introduced in a protocol upgrade. The contract *QBridgeHandler* [6] deployed on Ethereum has two functions to deposit assets into the contract: `deposit` and `depositETH`. We provide the code of both functions in Appendix A. Before the upgrade, users deposit ETH in the contract by calling *depositETH*. The resourceID corresponding to ETH resolves to the `ETH` address stored in the contract state which is the contract address of WETH (Wrapped ETH). The X-Bridge upgrade introduces a new function `depositETH` for directly depositing ETH instead of WETH in the contract. The intent is to replace `deposit` with `depositETH` for all ETH transfers. Thus, the upgrade also replaces the address `ETH` in the contract state with the zero address (intended to represent deposits of ETH). Unfortunately, this introduces side effects in the `deposit` function.

[5]tx hash 0x501f8541dc7cc594cd0ee245c8710ca12084529b596a3fc0d702f4a77f70b848 for instance
[6]At address: 0x20E5E35ba29dC3B540a1aee781D0814D5c77Bce6

The deposit function uses a low-level call to call transferFrom on ERC-20 compliant contract located at *tokenAddress*. In case users do not possess enough tokens, the deposit function fails. After the upgrade, calling the deposit function with the ETH address is still permitted but the *tokenAddress* resolves to the zero address meaning the low-level call always succeeds as there is no contract code stored in the account of the zero address. This enables users to trick the bridge and deposit an arbitrary amount of ETH without having to own any ETH. The attacker exploits this bug to illegally mint 77162 qXETH and uses them to borrow more than $80M worth of assets on BSC without any collateral.

We try capturing this bug by writing a property for the deposit function of the *QBridgeHandler* contract. It is interesting to see that several properties can capture the bug. A first approach would be to write a property stating that if the user is depositing ETH, the transaction should be associated with a value that matches the amount passed as a parameter. We express this property using our language in Listing 5.6. However, the property is quite complex and does not work. Indeed, the function deposit is not payable. Thus, the property trivially holds. Additionally, after the upgrade, as opposed to depositETH, the function deposit is not supposed to receive ether. We introduce property P2 in Listing 5.7. P2 states that depositing ETH with the deposit function is not permitted. This property captures the bug. However, note that the property is only valid after the protocol upgrade as the specification changed. Before the protocol upgrade, the deposit function is allowed to receive ETH which is not the case after the upgrade. This is already good, but we can do better by introducing property P3 in Listing 5.8. P3 states that on a successful deposit, the balance of the user depositing token decreases by 'amount'. The property holds on all valid deposits but the property check errors on the attacker transaction. It fails when trying to query the balance of the 'depositer'. The reason is that the contract at address this.resourceIDToTokenContractAddress(resourceID) is not ERC20 compliant (because it is the zero address) and do not implement a balanceOf function. It is interesting to note that in that case, the property catches the bug by triggering an error and not by being violated. Nevertheless, it successfully captures the vulnerability. Such cases could be false positives in case of checker or EVM error. However, it should not occur and should raise concerns.

```
if_succeeds "ETH Deposit should be associated with a msg value" {
    uint option;
    uint amount;
    (option, amount) = abi.decode(data, (uint, uint));
    if (this.tokenContractAddressToResourceID(this.ETH()) == resourceID) {
      return msg.value == amount;
    } else {
      return true;
    }
}
```

Listing 5.6: Qubit Finance: P1

```
if_succeeds "ETH deposits are not permitted" this.
   resourceIDToTokenContractAddress(resourceID) != this.ETH();
```

Listing 5.7: Qubit Finance: P2

```
if_succeeds "User balance decreases by 'amount'" {
        uint option;
        uint amount;
        (option, amount) = abi.decode(data, (uint, uint));
        return old(ERC20(this.resourceIDToTokenContractAddress(resourceID)).
   balanceOf(depositer)) - ERC20(this.resourceIDToTokenContractAddress(
   resourceID)).balanceOf(depositer) == amount;
}
```

Listing 5.8: Qubit Finance: P3

5.6 Nomad bridge

Nomad bridge is a bridge protocol [33]. It enables cross-chain communication through a variety of features. Nomad works by enabling dapps to send messages to pre-defined contracts on an origin chain, selecting a destination chain. Messages are relayed by off-chain agents to the destination chain. The key difference between Nomad and other bridge protocols is that it leverages *Optimistic Verification*. In most bridge protocols, off-chain agents verify that messages are correct through different proof systems before relaying them. In contrast, Nomad verification mechanism optimistically signs messages on the origin chain and enforces a timeout period on the destination chain, during which any user can veto it if they notice something wrong. This mechanism enables good security while minimizing trust assumptions.

The bridge was attacked on August 2, 2022, and $\approx \$190M$ were stolen [34]. This attack is particular because several users were able to replay the initial attack and steal funds by copying the attacker's transaction and replacing the attacker's address with their own address. The attack exploits a logic bug in the *Replica* contract [7]. This contract is in charge of tracking the Merkle Tree roots coming from origin chains and processing cross-chain messages if they are valid, i.e., included in a Merkle Tree with a valid Merkle Tree root. The contract stores a mapping from Merkle Tree roots to block timestamps from which the tree root is valid, i.e., after the optimistic

[7]At address: 0x5D94309E5a0090b165FA4181519701637B6DAEBA

verification time has elapsed. A second mapping maps a message hash to the Merkle Tree roots under which it is proven. When a message hash is not in the mapping (i.e., the message is not proven), the getter for the Merkle root returns Solidity default value: `bytes32(0)` in this case. When proven, the message is processed in the `process` function (we provide the code in Appendix A). This function verifies that the message is proven and that the Merkle Root is valid using the function `acceptableRoot`. Unfortunately, it turns out that `acceptableRoot` returns true for messages that are not proven! That is because `confirmAt[0x0] = 1`. This value is initialized at contract creation time and allows retro-compatibility with messages proven in previous versions of the protocol. The bug is introduced in a protocol upgrade[8] that changes the semantic of the process function.

We write a property for the `process` function to captures the logic bug (in Listing 5.9). We also annotate the `proveAndProcess` function with the same property as we also expect it to hold. The property verifies that the `process` function terminates successfully only if the message is proven, i.e., its hash is not null. For completeness, we also introduce a second property that asserts that the message has not already been processed, i.e., that the message is not replayed. We run our tool on the *Replica* contract with both properties. The tool reports a violation for the first property when the first attacker transaction executes at block 15259689.

```
if_succeeds "the message is proven" old(this.messages(keccak256(_message)))
    != this.LEGACY_STATUS_NONE();
if_succeeds "the message is not replayed" old(this.messages(keccak256(
   _message))) != this.LEGACY_STATUS_PROCESSED();
```

Listing 5.9: Nomad bridge: `process` properties

5.7 Poly Network Hack

As of today, the Poly Network Hack is the second biggest hack in Ethereum history [40]. Poly Network is a "bridge" protocol. It aims to provide blockchain interoperability by enabling cross-chain contract interactions and asset transfers. To perform that, it relies on a set of contracts deployed on every chain supported by the protocol. One of these contracts is *EthCrossChainManager*. This contract deployed on Ethereum is responsible for relaying transactions on Ethereum. It can call any Ethereum contract. It owns another contract: *EthCrossChainData* that keeps the list of public keys allowed to submit transactions to be relayed (called *keepers*). It also exposes a protected function to update this list. The attacker figured out that *EthCrossChainManager* could be used to update the list of keepers. He attacks the protocol by

[8]https://github.com/nomad-xyz/monorepo/commit/46d14571f3eada6ecf9ffce8fdc3adea11e73031#diff-45141e17961c00f6e486db12a2155e88d1c5e6dbc66410c62e37f9eb1e312e32

forging a transaction on Ontology chain that invokes the `putCurEpochConPubKeyBytes` function of *EthCrossChainData* to set his public key as the sole keeper. This transaction is relayed by Poly Network and broadcasted on Ethereum through *EthCrossChainManager* contract. As *EthCrossChainManager* has the permission to update the relayers, the transaction completes. The attacker becomes the network's sole keeper, granting him the permission to perform any operation such as transferring all the funds to his wallet. The attack was made possible because of the combination of two vulnerabilities: *EthCrossChainData* contract was owned by *EthCrossChainManager* which is a wrong design decision per the separation of duties and least privilege principles. It enabled to use the Manager contract to call any privileged method of other contracts it owns. Additionally, the team made an assumption that only methods having `(bytes,bytes,uint64)` signature could be called whereas one only needs to fork the right payload to call a function with an arbitrary signature. This is known as `function selector clashing` and occurs because of the way Solidity function selectors work. Poly Network now whitelists contracts and methods that can be called through cross-chain transactions to prevent a similar attack.

This attack is interesting as we fail to write a functional property in our language that captures the root cause of the hack. One can write a weak invariant on the *EthCrossChainData* contract expressing that the list of keepers is fixed, as long as it is not updated by the team. However, this would merely detect the attack but not the root vulnerability. This is where *information flow* properties are useful. One can encode a security policy in a functional property stating that there should be no way to call functions of *EthCrossChainData* from the function `verifyHeaderAndExecuteTx` of *EthCrossChainManager*. Unfortunately, for now, our tool does not support such properties. Additionally, the contract address to be called from `verifyHeaderAndExecuteTx` is destructured at runtime. This address is not an argument of the function. Unfortunately, it means we cannot easily access the address in the scope of a property. Consequently, we cannot write an `if_succeeds` property that asserts that the called address is not *EthCrossChainData* address. Also, we cannot write a property on `putCurEpochConPubKeyBytes` asserting that the sender of the call is not *EthCrossChainManager* because there exists some legitimate path between *EthCrossChainManager* and *EthCrossChainManager*. The function `changeBookKeeper` from *EthCrossChainManager* is authorized to call `putCurEpochConPubKeyBytes`.

5.8 Compound

Compound [8] is a lending and borrowing protocol deployed on Ethereum. Users supply Ether or ERC20 tokens and receive cTokens. Then, they can use cTokens to borrow other assets. In September 2021, the protocol introduced a bug while upgrading the protocol [9].

Users started to exploit it a few hours later. Compound's core team fixed it a week later [10]. To incentivize liquidity and usage of the protocol, Compound rewards active users by distributing its native COMP token when lending or borrowing assets on certain liquidity pools. These tokens have monetary value and can be used to vote for proposals of the Compound DAO. The bug enables users that provided liquidity on pools before the COMP distribution was enabled to claim COMP tokens that they should not have been rewarded. This "bank error" cost the protocol $147M worth of COMP tokens. The bug occurred because of a wrong check in the logic of the `distributeSupplierComp` and `distributeBorrowerComp` functions of the *Comptroller* contract[9]. The code of `distributeSupplierComp` is provided in Appendix A. The bug in `distributeBorrowerComp` is analogous. The bug is located on line 9. Instead of `supplyIndex > compInitialIndex`, the second condition in the if statement should be `supplyIndex >= compInitialIndex`. That is because when rewards for a pool are not activated, `compInitialIndex == 1e36`. For users that supplied liquidity in those pools, `supplyIndex == 1e36`. Thus, the contract execution does not branch into the if check leading to an overflow when computing the number of tokens to be distributed. As a result, users are awarded many tokens instead of zero.

We do not manage to write a functional property that captures this bug using our framework. This is because the *Comptroller* contract does not expose enough public information to precisely compute the number of COMP tokens to be distributed to a single address. This number of tokens is calculated internally and aggregated over all markets where the user borrows or supplies liquidity. This makes it impossible to write a simple property stating that users should not receive rewards for markets where rewards are not active. However, this is a case where having a language and a tool supporting hyper-properties would make a lot of sense. This enables to write an intuitive property that catches the bug.
Such a property would describe the following: assume a liquidity pool P and a user u providing liquidity n to the pool. Assume the two pseudo traces consisting of two sets of transactions in Figure 5.1. The only addition in execution trace 2 compared to trace 1 is that we deactivate the rewards before distributing the COMP rewards to the user. Starting from the same initial state and assuming $\alpha, \beta >= 0$, the number of COMP tokens distributed in the second execution should be less or equal to the number of tokens distributed in the first execution. This can be expressed with a hyper-property. We leave defining the language to express hyper-properties for future work.

[9]Contract at address: 0x374abb8ce19a73f2c4efad642bda76c797f19233

$$\begin{tikzcd}
\texttt{P.activateRewards()} \arrow[d] & \texttt{P.activateRewards()} \arrow[d] \\
\texttt{P.addLiquidity(u, n)} \arrow[d, "block.number \mathrel{+}= \alpha + \beta"] & \texttt{P.addLiquidity(u, n)} \arrow[d, "block.number \mathrel{+}= \alpha"] \\
\texttt{P.distributeRewards(u)} & \texttt{P.disableRewards()} \arrow[d, "block.number \mathrel{+}= \beta"] \\
 & \texttt{P.distributeRewards(u)} \\
\text{Trace 1} & \text{Trace 2}
\end{tikzcd}$$

Figure 5.1: Compound hyper-property execution traces

5.9 Harvest Finance

Harvest Finance is a yield-farming protocol. It enables users to deposit tokens and earn profits in a click by leveraging various DeFi economic strategies. Harvest finance lost ≈ \$33M in a hack in October 2020 [20]. The protocol pools the assets of users in liquidity pools. Each user is awarded some shares of the liquidity pool. The price of a share and the number of shares a user receives depends on the value of the assets in the pool. The price of these assets is calculated thanks to oracles. In that particular case, the price of the assets is sourced from Curve. The attacker manipulates the price of the assets (USDC/USDT) in the Curve pool to trick Harvest share computations. We must note that this attack sparked a lot of debate in the security community for some time as people debated whether it really was an attack or just a clever arbitrage. Today, this is considered to be an *oracle manipulation attack*. We fail to express any useful property that captures the attack with our language. It is not clear how to detect such vulnerabilities with traditional property-based tests. However, other methods such as model testing or agent-based testing could be useful.

5.10 Summary and discussion

We evaluate our tool on eight recent attacks that impacted different Ethereum protocols. Each of these attacks led to the loss of millions of dollars, impacting tens of thousands of users. We manage to capture five out of the eight attacks by writing a functional property using Jessy, our specification language. This supports our language design decisions and shows that it is possible to express interesting functional properties using our language, relying only on a contract ABI. For two of the three remaining attacks that we don't manage to capture, we suggest new language constructs for improving Jessy that would help capturing similar bugs. Finally, for the last attack, we explain why our approach fails to capture such economic vulnerabilities and suggest that other software verification approaches such as agent-based testing or model testing might be more suited for detecting them.

Our evaluation validates the idea that a few carefully designed functional properties might help improving protocols' security, both in the development phase by leveraging tools such as fuzzers and test networks, and in production by actively monitoring the contracts. We also notice that most exploits are executed using multiple transactions, often included in different blocks. This is either because the exploit requires preliminary preparation in a sequence of non-atomic actions, or because the attacker repeats its transaction several times to avoid gas limit and drain more funds. In most cases, it seems difficult to completely prevent an attack by simply monitoring the chain. However, assuming a way to monitor transactions in near real-time, and depending on the exploit, it suggests there is a chance to lower the impact of an attack by using an incident response strategy combined with appropriate mitigation techniques.

Chapter 6

Related Work

6.1 Smart contract verification

Software verification is a discipline whose goal is to make software more secure and robust and ensure a given software satisfies its specifications. It leverages several techniques ranging from manual code inspection to automated tools that we can split into four broad categories: static code analysis, dynamic code analysis, formal specification and verification, and other toolchains integrating several techniques [45]. Tools specific to smart contracts from all categories have been developed and reviewed several times [1][5]. They have different strengths and weaknesses. Static code analysis tools aim to prove the absence of bugs in software by checking its code before it is run. They face issues such as a high false-positive rate and difficulty identifying unexpected issues that may arise during runtime [23]. Dynamic verification (or runtime verification) tools check the proper behavior of software while it is running. The most common tools in this category are fuzzers which try to find bugs in a program by providing it inputs that are mutated in a more or less smart way. Depending on the fuzzing method, these tools perform very differently. The main issue they face is the coverage wall, i.e., the difficulty of achieving full code coverage. Formal verification tools such as symbolic execution engines, theorem provers, or other formal verification tools face the path explosion problem or the need to write complex specifications/model of the system.
Only some of these tools can detect business logic vulnerabilities. They require either partial or complete program specifications. Complete program specifications are usually time-consuming and hard to write [57]. Often, simple functional properties are useful as they enable developers to add partial specifications to the code and test their software against those specifications.

6.2 Property based testing

Property-based testing has become popular in the past few years thanks to the development of easy-to-use libraries for different programming languages. The most well-known is Hypothesis [27], a property-based testing library for Python inspired by the Haskell library Quickcheck [44]. These libraries enable to write property-based tests instead of traditional unit tests. Naturally, security engineers have extended property-based testing to Solidity contracts by adding support for property-based tests in several tools such as Brownie [41], Echidna [19] or Foundry [17]. These tools do not rely on specification languages to describe properties. Instead, they require the properties to be described in functions written in Solidity or in the tool's programming language. This can add friction and introduce boilerplate code to express some properties. Also, it makes it harder to interface with different checkers. The goal of these tools is to make writing property-based tests as straightforward as possible for developers that are used to writing unit tests.

6.3 Specification languages and runtime property checking

Specification languages designed for property checking have been developed for several programming languages. The most notable ones are JML [54] for Java and ACSL (Frama-C) [2] for C. Similarly to our specification language, Jessy, these languages enable to annotate programs with specifications: function pre/post-conditions and invariants. Similarly to our tool, properties can then be checked at runtime with specific tools such as OpenJML [35] or E-ACSL [12]. People have also developed static analysis tools such as ESC for JML [15] or WP for ACSL [59] that rely on different techniques, mainly SMT solvers like Z3 [30] or proof assistants like Coq [52], aiming to formally prove that the code is valid with respect to the specifications. The main drawback of these tools is that they do not scale well as the complexity of the code increases. This is where fuzzing and runtime monitoring help. Our specification language Jessy is inspired by Scribble [48], an open-source tool developed and maintained by Consensys Diligence. Scribble encompasses a specification language, the Scribble language, along with a tool that instruments Solidity files so that they can be checked using different tools such as fuzzers like Harvey [60] or symbolic execution engines like Mythril [32]. The key difference between Scribble and Jessy is that Scribble is made to check properties by instrumenting the contract source code, whereas Jessy relies on the contract ABI and does not require code instrumentation. This means Jessy supports different language constructs than Scribble and does not support static analysis tools. However, it can check properties dynamically on deployed systems.

6.4 Runtime monitoring

There exist few services that monitor Ethereum transactions and trigger custom alerts to developers when pre-defined or custom conditions are met. OpenZeppelin Defender [37] is the most popular. It allows developers to monitor transactions to a contract and write Javascript-based rules to filter the transactions and define conditions to trigger alerts. These conditions refer to transaction parameters, emitted events, and function arguments. The service can also trigger automatic mitigation, such as sending a transaction to pause the contract. Tenderly [53] provides a similar service. It can monitor contracts and send alerts based on different conditions, e.g., the value returned by a view function of the contract or change of a contract state variable. More recently, Forta Network [16] was released. Forta is a decentralized network composed of independent *scan nodes* constantly monitoring different blockchains (including Ethereum) to detect threats in real-time. Scan nodes process blockchain data and execute *detection bots*, i.e., a set of code scripts running in a Docker container. The detection bots emit alerts when specific threat conditions occur. For instance, if a flash loan attack occurred or when an account balance fell below some pre-defined value. Users can write their own bots to monitor their projects, and deploy them on the network. Bots are scripts developed in Javascript or Python that operate on transaction logs. The goal is to alert projects so that they can provide an appropriate response in time if something strange occurs. Compared to our approach, Forta does not provide a custom specification language and does not enable to check arbitrary functional properties. However, Jessy could perhaps be used by detection bots.

Chapter 7

Conclusion

Securing smart contracts is critical. The total value controlled by smart contracts drastically increased in the last two years. Several tools, based on different techniques, have been developed for finding bugs in smart contracts. These tools face different challenges with different objectives and strengths/weaknesses. Yet, they all have the same end goal: making smart contracts more secure.
To be efficient in practice, an automated software security technique has to require a proportionate amount of effort compared to the guarantees it provides. It must also be relatively easy to use so that developers can adopt it. In fact, most recent exploits involve business-logic vulnerabilities. This makes it difficult to detect them using simple "pattern-based" analyses. In contrast, property-based testing provides a good cost-efficiency trade-off and allows detecting business-logic issues.

In this work, we introduce Jessy, a specification language tailored for EVM smart contracts. Jessy enables expressing functional properties that can be checked at runtime without code instrumentation. Compared to other approaches that require code instrumentation, our approach reduces the effort needed to check properties. It also enables checking properties on deployed contracts using only their ABI. This approach trades language expressive power for ease of use and setup time. We develop a prototype tool that checks Jessy properties at runtime by replaying Ethereum transactions. The tool relies on the EVM tracing capabilities and a remote Ethereum RPC node, meaning it does not require syncing a local archive node.

We evaluate our tool by studying 8 recent exploits that occurred on different systems deployed on Ethereum. We try writing Jessy properties that capture the vulnerabilities. In 5 out of 8 cases, we manage to write a functional property that captures the bug, validating our approach. In the 3 other cases, we analyze why we do not manage to capture the vulnerability with simple properties. It provides us insights for future extensions. Our evaluation also shows that several attacks are carried-out in multiple transactions, leaving room for proper incident response and mitigation systems.

Future work is directed towards improving the language by adding support for additional properties such as information-flow properties, temporal properties such as "once" or "forall" properties, and eventually, hyper-properties. One could also focus on improving existing language constructs, e.g., by adding support for reading private contract variables, i.e., arbitrary EVM slots.

While developing and evaluating our tool, we faced issues such as the lack of a proper way to fetch all the internal transactions to a contract, difficulties knowing which EVM slots are accessed by a given transaction or the difficulty of fetching the state of the chain in between transactions. We also faced challenges when analyzing previous exploits as existing tools to analyze and debug transactions are still limited and do not always provide clear information. This supports the idea that there is still room to improve blockchain security and analysis tools. Considering the strong economic incentives to increase the security of the blockchain and its everyday-increasing usage, we are confident that tooling will improve and new techniques will emerge.

Bibliography

[1] Sefa Akca, Chao Peng, and Ajitha Rajan. “Testing Smart Contracts: Which Technique Performs Best?” In: *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2021, pp. 1–11.

[2] *ANSI/ISO C Specification Language*. 2022. URL: `https://frama-c.com/html/acsl.html`.

[3] *Audius Documentation*. Accessed 07/2022. URL: `https://docs.audius.org/protocol/overview`.

[4] *Audius Governance Takeover Post-Mortem*. July 2022. URL: `https://blog.audius.co/article/audius-governance-takeover-post-mortem-7-23-22`.

[5] Chaïmaa Benabbou and Önder Gürcan. “A Survey of Verification, Validation and Testing Solutions for Smart Contracts”. In: *2021 Third International Conference on Blockchain Computing and Applications (BCCA)*. IEEE. 2021, pp. 57–64.

[6] BillWagner. *Classes, structs, and records*. docs.microsoft.com, Mar. 2022. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/`.

[7] Vitalik Buterin et al. “A next-generation Smart Contract and decentralized application platform Ethereum. White Paper”. In: *Ethereum Project White Paper* (2014).

[8] *Compound*. Accessed 07/2022. URL: `https://compound.finance/`.

[9] *Compound governance proposal 62*. Accessed 07/2022. URL: `https://compound.finance/governance/proposals/62`.

[10] *Compound governance proposal 64*. Accessed 07/2022. URL: `https://compound.finance/governance/proposals/64`.

[11] *Contracts - Solidity documentation*. URL: `https://docs.soliditylang.org/en/v0.8.13/contracts.html`.

[12] *E-ACSL - Frama C*. 2022. URL: `https://frama-c.com/fc-plugins/e-acsl.html`.

[13] *Etherscan documentation*. Accessed 06/2022. URL: `https://docs.etherscan.io`.

[14] *EVM Tracing - Go Ethereum*. Accessed 05/2022. URL: `https://geth.ethereum.org/docs/dapp/tracing`.

[15] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James Saxe, and Raymie Stata. “Extended Static Checking for Java”. In: vol. 37. May 2002. DOI: `10.1145/512529.512558`.

[16] *Forta Network*. Accessed 08/2022. URL: `https://forta.org/`.

[17] *Fuzz Testing - Foundry Book*. 2022. URL: `https://book.getfoundry.sh/forge/fuzz-testing.html`.

[18] *Go Ethereum (GETH)*. Accessed 05/2022. URL: `https://geth.ethereum.org`.

[19] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. “Echidna: effective, usable, and fast fuzzing for smart contracts”. In: July 2020, pp. 557–560. DOI: `10.1145/3395363.3404366`.

[20] *Harvest Finance - REKT*. Oct. 2020. URL: `https://rekt.news/harvest-finance-rekt/`.

[21] Charles Antony Richard Hoare. “Proof of correctness of data representations”. In: *Programming methodology*. Springer, 1978, pp. 269–281.

[22] *Infura documentation*. Accessed 06/2022. URL: `https://docs.infura.io/infura/networks/ethereum`.

[23] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. “Why don’t software developers use static analysis tools to find bugs?” In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681.

[24] Tomas Krupa, Michal Ries, Ivan Kotuliak, Rastislav Bencel, et al. “Security issues of smart contracts in ethereum platforms”. In: *2021 28th Conference of Open Innovations Association (FRUCT)*. IEEE. 2021, pp. 208–214.

[25] Baudouin Le Charlier and Pierre Flener. “Specifications are necessarily informal or: Some more myths of formal methods”. In: *Journal of Systems and Software* 40.3 (1998), pp. 275–296.

[26] K Rustan M Leino and Peter Müller. “Modular verification of static class invariants”. In: *International Symposium on Formal Methods*. Springer. 2005, pp. 26–42.

[27] David R. MacIver. *Welcome to Hypothesis! — Hypothesis 4.50.8 documentation*. 2022. URL: `https://hypothesis.readthedocs.io/en/latest/`.

[28] Bertrand Meyer. “Applying’design by contract’”. In: *Computer* 25.10 (1992), pp. 40–51.

[29] Bertrand Meyer. “Class invariants: concepts, problems, solutions”. In: *arXiv preprint arXiv:1608.07637* (2016).

[30] 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. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[31] Peter Müller, Arnd Poetzsch-Heffter, and Gary T Leavens. “Modular invariants for layered object structures”. In: *Science of Computer Programming* 62.3 (2006), pp. 253–286.

[32] *Mythril - Consensys.* Sept. 2021. URL: https://github.com/ConsenSys/mythril.

[33] *Nomad Bridge Documentation.* Accessed 07/2022. URL: https://docs.nomad.xyz/nomad-101/introduction.

[34] *Nomad Bridge Hack: Root Cause Analysis.* Aug. 2022. URL: https://medium.com/nomad-xyz-blog/nomad-bridge-hack-root-cause-analysis-875ad2e5aacd.

[35] *Open JML.* 2022. URL: https://www.openjml.org/.

[36] *OpenZeppelin - Unstructured Storage Proxies.* Accessed 07/2022. URL: https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies#unstructured-storage-proxies.

[37] *OpenZeppelin Defender.* URL: https://www.openzeppelin.com/defender.

[38] Packard and Alan Snyder. *Wl3 HEWLETT The Essence of Objects: Common Concepts and Terminology.* 1991. URL: https://www.hpl.hp.com/techreports/91/HPL-91-50.pdf (visited on 04/12/2022).

[39] *Para Space (OMNI) protocol.* Accessed 07/2021. URL: https://docs.para.space/para-space/para-space/readme.

[40] *Poly Network Hack - REKT.* Accessed 07/2022. URL: https://rekt.news/polynetwork-rekt/.

[41] *Property-Based Testing - Brownie.* 2022. URL: https://eth-brownie.readthedocs.io/en/stable/tests-hypothesis-property.html.

[42] *Protocol Exploit Report - Qubit Finance.* Jan. 2022. URL: https://medium.com/@QubitFin/protocol-exploit-report-305c34540fa3.

[43] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. “Attacking the defi ecosystem with flash loans for fun and profit”. In: *International Conference on Financial Cryptography and Data Security.* Springer. 2021, pp. 3–32.

[44] *QuickCheck: Automatic testing of Haskell programs.* Accessed 07/2022. URL: https://hackage.haskell.org/package/QuickCheck.

[45] Heidelinde Rameder, Monika Di Angelo, and Gernot Salzer. “Review of automated vulnerability analysis of smart contracts on Ethereum”. In: *Front. Blockchain* 5 (2022).

[46] *Rari Capital.* Accessed 07/2022. URL: https://www.rari.capital.

[47] *Rari Fuse Hack - REKT.* Accessed 07/2022. URL: https://rekt.news/fei-rari-rekt/.

[48] *Scribble Documentation.* URL: https://docs.scribble.codes/.

[49] *Security considerations - Solidity documentation.* Accessed 08/2022. URL: https://docs.soliditylang.org/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern.

[50] Gary Shute. *Object-Oriented Terminology.* Umn.edu, 2019. URL: https://www.d.umn.edu/~gshute/softeng/object-oriented.html.

[51] Mirko Staderini, Caterina Palli, and Andrea Bondavalli. "Classification of Ethereum Vulnerabilities and their Propagations". In: *2020 Second International Conference on Blockchain Computing and Applications (BCCA).* 2020, pp. 44–51. DOI: 10.1109/BCCA50787.2020.9274458.

[52] The Coq development team. *The Coq proof assistant reference manual.* Version 8.15.0. 2022. URL: http://coq.inria.fr.

[53] *Tenderly.* URL: https://tenderly.co.

[54] *The Java Modeling Language (JML).* 2022. URL: https://www.cs.ucf.edu/~leavens/JML/index.shtml.

[55] *Units and Globally Available Variables.* URL: https://docs.soliditylang.org/en/v0.8.13/units-and-global-variables.html#special-variables-and-functions.

[56] Lisa Walters. *OMNI Protocol Loses $1.43 Million Worth of Testing Funds in Reentrancy Attack.* July 2022. URL: https://coinquora.com/omni-protocol-loses-1-43-million-worth-of-testing-funds-in-reentrancy-attack/.

[57] Hillel Wayne. *Why don't people use formal methods? • Hillel Wayne.* Jan. 2019. URL: https://www.hillelwayne.com/post/why-dont-people-use-formal-methods/.

[58] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[59] *WP - FramaC.* 2022. URL: https://frama-c.com/fc-plugins/wp.html.

[60] Valentin Wüstholz and Maria Christakis. "Harvey: a greybox fuzzer for smart contracts". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Oct. 2020). DOI: 10.1145/3368089.3417064. URL: https://mariachris.github.io/Pubs/FSE-2020-Harvey.pdf.

[61] Pengcheng Zhang, Feng Xiao, and Xiapu Luo. "A framework and dataset for bugs in ethereum smart contracts". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2020, pp. 139–150.

Appendix A

Code of vulnerable functions

A.1 Compound: `distributeSupplierComp`

```solidity
function distributeSupplierComp(address cToken, address supplier) internal {
    // TODO: Don't distribute supplier COMP if the user is not in the supplier market.
    // This check should be as gas efficient as possible as distributeSupplierComp is called in many places.
    CompMarketState storage supplyState = compSupplyState[cToken];
    uint256 supplyIndex = supplyState.index;
    uint256 supplierIndex = compSupplierIndex[cToken][supplier];
    // Update supplier's index to the current index since we are distributing accrued COMP
    compSupplierIndex[cToken][supplier] = supplyIndex;
    if (supplierIndex == 0 && supplyIndex > compInitialIndex) {
        // Covers the case where users supplied tokens before the market's supply state index was set.
        // Rewards the user with COMP accrued from the start of when supplier rewards were first set for the market.
        supplierIndex = compInitialIndex;
    }
    // Calculate change in the cumulative sum of the COMP per cToken accrued
    Double memory deltaIndex = Double({
        mantissa: sub_(supplyIndex, supplierIndex)
    });
    uint256 supplierTokens = CToken(cToken).balanceOf(supplier);
    // Calculate COMP accrued: cTokenAmount * accruedPerCToken
    uint256 supplierDelta = mul_(supplierTokens, deltaIndex);
    uint256 supplierAccrued = add_(compAccrued[supplier], supplierDelta);
    compAccrued[supplier] = supplierAccrued;
    emit DistributedSupplierComp(
        CToken(cToken),
        supplier,
        supplierDelta,
        supplyIndex
    );
}
```

A.2 Rari Fuse: `borrow`

```
function borrowFresh(address payable borrower, uint256 borrowAmount)
    internal
    returns (uint256)
{
    /* Fail if borrow not allowed */
    uint256 allowed = comptroller.borrowAllowed(
        address(this),
        borrower,
        borrowAmount
    );
    if (allowed != 0) {
        return
            failOpaque(
                Error.COMPTROLLER_REJECTION,
                FailureInfo.BORROW_COMPTROLLER_REJECTION,
                allowed
            );
    }

    /* Verify market's block number equals current block number */
    if (accrualBlockNumber != getBlockNumber()) {
        return fail(Error.MARKET_NOT_FRESH, FailureInfo.
            BORROW_FRESHNESS_CHECK);
    }

    /* Fail gracefully if protocol has insufficient underlying cash */
    uint256 cashPrior = getCashPrior();

    if (cashPrior < borrowAmount) {
        return
            fail(
                Error.TOKEN_INSUFFICIENT_CASH,
                FailureInfo.BORROW_CASH_NOT_AVAILABLE
            );
    }

    /*
     * We calculate the new borrower and total borrow balances, failing
        on overflow:
     *  accountBorrowsNew = accountBorrows + borrowAmount
     *  totalBorrowsNew = totalBorrows + borrowAmount
     */

    //code omitted for concision

    /////////////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)
    /*
     * We invoke doTransferOut for the borrower and the borrowAmount.
     *  Note: The cToken must handle variations between ERC-20 and ETH
        underlying.
     *  On success, the cToken borrowAmount less of cash.
     *  doTransferOut reverts if anything goes wrong, since we can't be
        sure if side effects occurred.
     */
    doTransferOut(borrower, borrowAmount); //REENTRANCY Vulneraility

    /* We write the previously calculated values into storage */
    accountBorrows[borrower].principal = vars.accountBorrowsNew;
    accountBorrows[borrower].interestIndex = borrowIndex;
    totalBorrows = vars.totalBorrowsNew;

    /* We emit a Borrow event */
    emit Borrow(
        borrower,
        borrowAmount,
        vars.accountBorrowsNew,
        vars.totalBorrowsNew
    );

    /* We call the defense hook */
    // unused function
    // comptroller.borrowVerify(address(this), borrower, borrowAmount);

    return uint256(Error.NO_ERROR);
}
```

A.3 Audius: `initialize`

```
function initialize(
    address _registryAddress,
    uint256 _votingPeriod,
    uint256 _executionDelay,
    uint256 _votingQuorumPercent,
    uint16 _maxInProgressProposals,
    address _guardianAddress
) public initializer {
    require(_registryAddress != address(0x00), ERROR_INVALID_REGISTRY);
    registry = Registry(_registryAddress);

    require(_votingPeriod > 0, ERROR_INVALID_VOTING_PERIOD);
    votingPeriod = _votingPeriod;

    // executionDelay does not have to be non-zero
    executionDelay = _executionDelay;

    require(
        _maxInProgressProposals > 0,
        "Governance: Requires non-zero _maxInProgressProposals"
    );
    maxInProgressProposals = _maxInProgressProposals;

    require(
        _votingQuorumPercent > 0 && _votingQuorumPercent <= 100,
        ERROR_INVALID_VOTING_QUORUM
    );
    votingQuorumPercent = _votingQuorumPercent;

    require(
        _guardianAddress != address(0x00),
        "Governance: Requires non-zero _guardianAddress"
    );
    guardianAddress = _guardianAddress;  //Guardian address becomes the
       only party

    InitializableV2.initialize();
}


contract InitializableV2 is Initializable {
    bool private isInitialized;

    function initialize() public initializer {
        isInitialized = true;
    }

    //...
```

```
}

contract Initializable {
  address private proxyAdmin;

  uint256 private filler1;
  uint256 private filler2;

  // Indicates that the contract has been initialized.
  bool private initialized;

  //@dev Indicates that the contract is in the process of being
     initialized.
  bool private initializing;

  modifier initializer() {
    require(msg.sender == proxyAdmin, "Only proxy admin can initialize");
    require(initializing || isConstructor() || !initialized, "Contract
       instance has already been initialized");

    bool isTopLevelCall = !initializing;
    if (isTopLevelCall) {
      initializing = true;
      initialized = true;
    }

    _;

    if (isTopLevelCall) {
      initializing = false;
    }
  }
```

A.4 Qubit Finance: `deposit`, `depositETH`

```
/**
        @notice A deposit is initiated by making a deposit in the Bridge
           contract.
        @param resourceID ResourceID used to find address of token to be
           used for deposit.
        @param depositer Address of account making the deposit in the
           Bridge contract.
        @param data passed into the function should be constructed as
           follows:
        option                                uint256     bytes  0 - 32
        amount                                      uint256      bytes   32 - 64
     */
function deposit(
    bytes32 resourceID,
    address depositer,
    bytes calldata data
) external override onlyBridge {
    uint256 option;
    uint256 amount;
    (option, amount) = abi.decode(data, (uint256, uint256));

    address tokenAddress = resourceIDToTokenContractAddress[resourceID];
    //WETH tokenAddress = 0x0
    require(
        contractWhitelist[tokenAddress],
        "provided tokenAddress is not whitelisted"
    );

    if (burnList[tokenAddress]) {
        require(
            amount >= withdrawalFees[resourceID],
            "less than withdrawal fee"
        );
        QBridgeToken(tokenAddress).burnFrom(depositer, amount);
    } else {
        require(
            amount >= minAmounts[resourceID][option],
            "less than minimum amount"
        );
        tokenAddress.safeTransferFrom(depositer, address(this), amount);
           //code included below
    }
}

function depositETH(
    bytes32 resourceID,
    address depositer,
    bytes calldata data
) external payable override onlyBridge {
    uint256 option;
    uint256 amount;
    (option, amount) = abi.decode(data, (uint256, uint256));
    require(amount == msg.value);

    address tokenAddress = resourceIDToTokenContractAddress[resourceID];
    require(
        contractWhitelist[tokenAddress],
        "provided tokenAddress is not whitelisted"
    );

    require(
        amount >= minAmounts[resourceID][option],
        "less than minimum amount"
    );
}

function safeTransferFrom(
    address token,
    address from,
    address to,
    uint256 value
) internal {
    // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
    (bool success, bytes memory data) = token.call(
        abi.encodeWithSelector(0x23b872dd, from, to, value)
    );
    require(
        success && (data.length == 0 || abi.decode(data, (bool))),
        "!safeTransferFrom"
    );
}
```

A.5 Nomad Bridge: process, acceptableRoot

```
function process(bytes memory _message) public returns  (bool _success) {
    // ensure message was meant for this domain
    bytes29 _m = _message.ref(0);
    require(_m.destination() == localDomain, "!destination");
    // ensure message has been proven
    bytes32 _messageHash = _m.keccak();
    require(acceptableRoot(messages[_messageHash]), "!proven");
    // check re-entrancy guard
    require(entered == 1, "!reentrant");
    entered = 0;
    // update message status as processed
    messages[_messageHash] = LEGACY_STATUS_PROCESSED;
    // call handle function
    IMessageRecipient(_m.recipientAddress()).handle(
        _m.origin(),
        _m.nonce(),
        _m.sender(),
        _m.body().clone()
    );
    // emit process results
    emit Process(_messageHash, true, "");
    // reset re-entrancy guard
    entered = 1;
    // return true
    return true;
}

function acceptableRoot(bytes32 _root) public view returns (bool) {
    // this is backwards-compatibility for messages proven/processed
    // under previous versions
    if (_root == LEGACY_STATUS_PROVEN) return true;
    if (_root == LEGACY_STATUS_PROCESSED) return false;

    // BUG: _root == 0x0 ==> _time = confirmAt[0x0] = 1
    // (intialized at contract creation)
    uint256 _time = confirmAt[_root];
    if (_time == 0) {
        return false;
    }
    return block.timestamp >= _time;
}
```

Appendix B

Additional Material

Algorithm 3: Find transactions to execute

```
Input: $iTx, eTx$: set of internal and external transactions of a block
       $dependencies$: set of addresses: address of the contract to monitor + addresses
       of all accounts called in properties
Output: $txToExecute$: set of transaction hashes to execute
Procedure BuildAndPruneTransactionGraph(iTx,eTx)
    $componentsToExecute \leftarrow \emptyset$
    $txToExecute \leftarrow \emptyset$
    $G(v,e) \leftarrow BuildTxGraph(iTx, eTx)$
    $C \leftarrow findConnectedComponents(G)$
    for every component $c \in C$ do
        /* Look at every connected components */
        for every vertex $v \in c$ do
            if $v \in dependencies$ then
                /* if the address in the vertex is part of the dependency-list,
                   mark the connected component as a component to execute */
                $componentsToExecute.add(c)$
            end if
        end for
    end for
    for every component $c$ in $componentsToExecute$ do
        for every edge $e$ in $c$ do
            /* add transactions hashes to the list of tx to execute */
            $txToExecute.add(e.txHash)$
        end for
    end for
    return $txToExecute$
```

```
Procedure BuildTxGraph(iTx,eTx)
    $V \leftarrow \emptyset$
    $E \leftarrow \emptyset$
    for $tx$ in $iTx, eTx$ do
        if there is no vertex $tx.from$ in $V$ then
            add new vertex $tx.from$ to $V$
        end if
        if there is no vertex $tx.to$ in $V$ then
            add new vertex $tx.to$ to $V$
        end if
        if there is an edge $e$ in $E$ between $tx.from$ and $tx.to$ then
            /* $e.txHash$ is a list of transaction hashes                    */
            /* add this transaction hash $tx.hash$ to the list               */
            $e.txHash.add(tx.hash)$
        else
            create edge $e$ between $tx.from$ and $tx.to$ in $E$
            $e.txHash.add(tx.hash)$
        end if
    end for
return $V, E$
```

Appendix C

Extension Ideas

In this appendix, we list a few ideas and motivations for adding support for more complex properties to the language and the tool.

The notion of time is very important in blockchains. Blockchains are made of consecutive blocks emitted at a regular time. Blocks are batches of transactions with a hash of the previous block in the chain. Miners can choose the ordering of transactions within a block, the only constraint being that transactions of a given account must be ordered by non-decreasing nonce. Miners usually order transactions in a way that maximizes their profit. This is called Miner Extractable Value (MEV). However, reordering transactions can sometimes lead to nondesirable side effects for the user. Specifying and verifying transaction ordering properties may help prevent undesired consequences of transaction reordering. Similarly, temporal logic properties would be other interesting extensions to consider. Smart contract protocols are usually made of a sequence of actions/transactions that should be ordered in a given way. Being able to express temporal specifications in properties and checking them could help discover vulnerabilities. In our case, as we check properties at runtime, support would be limited to past temporal properties.

By contrast to trace properties, hyperproperties are properties of computational systems that require more than one trace to evaluate. In the case of smart contracts, we can designate properties that require more than one transaction to be checked "hyperproperties". Hyperproperties are interesting because they can express system properties that cannot be expressed in trace properties. However, these properties are challenging to check because one cannot check them during a single program execution. It requires running the system multiple times with the same state on different inputs and comparing the outputs. Many information flow and robustness properties are hyperproperties.

Information flow properties consider how data should flow through the program. These properties can catch vulnerabilities related to improper use of untrusted data. For example, if untrusted information from external sources is allowed to spread to critical program points, it can alter key aspects of program behavior. These hyperproperties are challenging to express and verify. Checking information flow properties at the ABI level is interesting to explore.

Robustness testing designates any quality assurance methodology focused on testing the robustness of software. Robustness is defined by the IEEE standard glossary of software engineering terminology as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions". The goal of robustness testing is to develop test cases and test environments where a system's robustness can be assessed. By definition, most smart contract systems live on public blockchains. Everyone can access, interact and influence the behavior of the system. This makes public chains a very adversarial environment. Software needs to be very robust and resilient to changing states and inputs. Thus, assessing robustness of functions/systems by verifying properties taking robustness into account would be interesting.