Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Ethereum smart contracts verification: a survey and a prototype tool

Tesis de Licenciatura en Ciencias de la Computación

Vera Bogdanich Espina

Director: Diego Garbervetsky

CABA, Argentina, 2019

ABSTRACT

Smart contracts are programs that can be consistently executed by a network of mutually distrusting nodes, without the arbitration of a trusted authority. Because of their resistance to tampering, smart contracts are appealing in many scenarios, especially in those which require transfers of money to respect of certain agreed rules. Unfortunately, programming smart contracts is a delicate task that requires strong expertise: the rich applications and semantics of decentralized applications inevitably introduce many security vulnerabilities. Therefore, methods and tools have emerged to support the development of secure smart contracts. Assessing the quality of such tools turns out to be difficult.

This thesis is meant as a guide for those who intend to analyze smart contracts, either deployed ones or during development. In particular, for OpenZeppelin auditors, who showed interest in automated analysis and want to apply it in their day to day work. Besides doing this survey on automatic analysis techniques for smart contracts, we also developed a prototype tool that combines two existing approaches, and adds a specification language to create a monitor for safety checks. This monitor was implemented as an instrumentation of the provided contract, so it can be fed to any analysis program to take advantage of its capabilities while also expressing contract invariants in a language inspired in temporal logics.

Keywords: Ethereum, software verification, survey, Solidity, temporal logic, monitoring

CONTENTS

1. INTRODUCTION

Since the release of Bitcoin [1] in 2009, the idea of taking advantage of its technology to develop applications beyond currency has been receiving increasing attention. In particular, the append only keeping book for transactions (the blockchain), and the decentralized consensus protocol that Bitcoin nodes use to extend it (relying on a peer-to-peer, or p2p, network), have revived Nick Szabo's idea of *smart contracts* [2]: programs whose correct execution is automatically enforced without relying on a trusted authority. The archetypal implementation of smart contracts is the Ethereum framework [3]. Ethereum has become the de facto standard platform for smart contract development, for this reason, we focus exclusively on it. The consensus protocol of Ethereum ensures that only the valid updates to the contract states are recorded in the blockchain, so ensuring their correct execution. However, it only ensures correct execution of source code written in an assembly-like language, or its high level alternatives that compile to it; it's not equivalent to ensuring correctness with respect to the intended behavior by the developer. Being confident about smart contracts behavior is another important problem to tackle.

Errors in smart contracts present a serious issue for multiple reasons. Firstly, smart contracts usually handle financial assets of significant value. Secondly its bugs cannot be patched. By design, once a contract is deployed, its functionality cannot be altered even by its creator. Finally, once a faulty or malicious transaction is recorded, it cannot be removed from the blockchain.

Consequently there are many companies that offer audit services for developers that want to create an application that relies on the Ethereum blockchain. Some services consist in systems that can be used by the client to check their contracts, but most provide manual audits by security specialists. During these audits, security specialists usually try to understand the client's system to manually find issues before deployment. Some companies offer to also use automated tools to analyze the client's code, with the intention of providing even greater guarantees. Among these companies we can find OpenZeppelin [4]. Right now, they perform mostly manual analysis to check contracts, but they understand the importance of formal guarantees in the context of the Ethereum blockchain. Because of that, they partially funded this work, where we try to understand the state of the art in analysis techniques for Ethereum smart contracts, and to develop a tool that could be useful to them in their day to day work auditing contracts.

Our contributions are:

- An extensive survey on the state of the art of automated analysis techniques for Ethereum smart contracts. It is presented as a guide for developers or auditors that might not have a background on system verification but want to take advantage of its benefits.

- Considering the lack of standardized benchmarks we proposed a first example to compare qualitative aspects of the tools. We only compared a selection from the ones discussed in the survey, we only selected techniques that have a publicly available mature implementation.

- As a means of evaluation we also performed a case study with the same tool selection. We choose a popular contract used currently in production, and attempted to use the tools as we expect an auditor would.
- A prototype to integrate two tools with very promising results in the survey to create a new one with the benefits of both.
- An extension to our first prototype to monitor safety properties expressed with a Solidity-like specification language that supports temporal operators.

2. PRELIMINARY

This chapter provides a brief introduction to Ethereum, Solidity (a high level programming language widely used for Ethereum smart contracts), and automated system verification.

2.1 Ethereum and Solidity

Ethereum [3] as a whole can be viewed as a transaction based state machine; transactions represent valid arcs between states. These transactions are collected into *blocks*, that are chained together using a cryptographic hash as a means of reference: each block contains in one of its fields a hash of the previous block, as shown in figure 2.1.

A *transaction* is a single cryptographically-signed instruction constructed by an actor externally to the scope of Ethereum. While a *state* can be though of as a mapping between *addresses* (160 bit identifiers) and account states. Likewise, an Ethereum state can be thought of as an object comprised of many small objects called *accounts* that are able to interact with one another through a message-passing framework. There are two types of accounts: *externally owned accounts* which are controlled by private keys and have no code associated with them, and *contract accounts* which are controlled by their associated code. This code associated is usually referred to as a *smart contract*, whereas a whole system that interacts with such code is called a *dapp*, as in "decentralized application". A contract account is created by one of the two types of transactions, the other type sends messages to code already associated with an account. Each account also has storage space, that can be used for this code, and even for data. Once code is placed in an account's storage, it cannot be replaced or modified.

Since the system is decentralized and all parties have an opportunity to create a new block on some older preexisting block, the resultant structure is a tree of blocks. In order to form a consensus as to which path, from the root (a blocked referred to as "genesis block") to a leaf (the block containing the most recent transactions) through this tree structure, known as the *blockchain*, there must be an agreed-upon scheme. If there is a disagreement

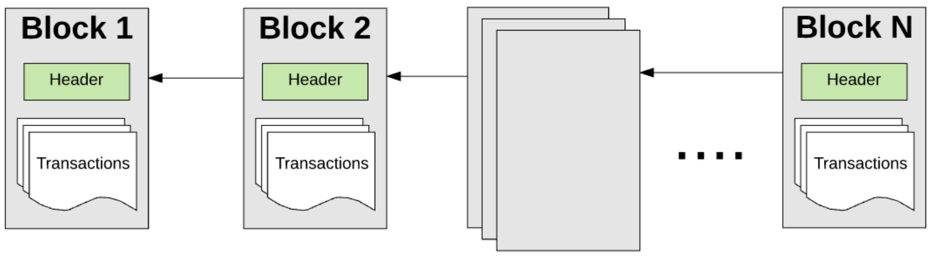

Fig. 2.1: Representation of a blockchain that stores transactions, taken from https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369

between nodes as to which path down the block tree is the "best" blockchain, then a *fork* occurs. In order to have consensus over which path it is, conceptually we identify the path that has had the most computation done upon it, or, the "heaviest path". Forking is the only way to roll back a transaction, which requires consensus among several parties, and undermines the trustworthiness of the platform. Besides choosing between forks, nodes can also decide to which blockchain connect: they can start their own local blockchain for development or testing; most cryptocurrencies also have public blockchains dedicated to testing. We will refer to the main blockchain that holds currency with real life value as "mainnet", and we will refer to the rest as "testnet".

Transaction series are punctuated with incentives for nodes to mine, *mining* is the process of dedicating effort to bolster one series of transactions (a block) over any other potential competitor block. It is achieved thanks to a cryptographically secure proof. This scheme is known as *proof of work* or PoW. It proves beyond reasonable doubt that a particular amount of computation has been done. The proof of work not only functions as a method of securing confidence that the blockchain will remain canonical into the future, but also as a wealth distribution mechanism. It should be as accessible as possible to as many people as possible. The requirement of, or reward from, specialized and uncommon hardware should be minimized, to avoid like that having parties with enough power to fork the blockchain whenever they want.

A miner who validates a new block is rewarded with a certain amount of value for doing this work; Ethereum uses an intrinsic digital token called *ether*, every time a miner proves a block, new ether is generated and awarded. Each Ethereum account has a *balance* field, it represents the amount of ether owned by this address, that's where the ether awarded to miners is "deposited".

In order to avoid issues of network abuse and the inevitable questions stemming from Turing completeness, all programmable computation in Ethereum is subject to fees. The fee schedule is specified in units of *gas*. Memory storage is considerably more expensive than to perform other instructions, and it is more costly the larger it grows, it scales quadratically [5]. Miners, in general, will choose to advertise the minimum gas price for which they will execute transactions and transactors will be free to ponder these prices to determine what gas price to offer.

The execution model specifies how the system state is altered given a series of bytecode instructions and a small tuple of environmental data. This is specified through a formal model of a virtual state machine, known as the *Ethereum Virtual Machine* (EVM). The EVM is a simple stack based architecture. These bytecode instructions correspond to the message calls we already mentioned that are included in transactions by external actors. For these purposes, transactions contain properties such as:

- *gas price*: a scalar value equal to the number of wei[1] to be paid per unit of gas for all computation costs incurred as a result of the execution of this transaction.

- *gas limit*: a scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up front, before any computation is done

[1] The minimum amount of ether that can be handled is a *wei*, there are 10^{18} weis per ether.

and may not be increased later.

- *to*: the 160 bit address of the message call's recipient. In a contract creation transaction this field is empty.
- *value*: a scalar value equal to the number of wei to be transferred to the message call's recipient. In the case of contract creation, as an endowment to the newly created account.
- *init*: only contained in contract creation transactions. It's an unlimited size byte array specifying the EVM code for the account initialization procedure.
- *data*: only contained in message call transactions. It is an unlimited size byte array specifying the input data of the message call.

Ethereum smart contracts are mostly developed in an ad-hoc language, called Solidity [6], an object oriented high level language that features non-standard semantic behaviors and transaction-oriented mechanisms, which complicate smart contract development and verification. For example, the context of each function contains information about the block where the transaction that triggered that function call is included, information on the transaction, etc. Calls also have a different semantic than regular calls in imperative programming languages: there's a distinction between transactions triggered by externally owned accounts (*external transactions*), and those triggered by contract accounts (*internal transactions*). Solidity also provides almost direct access to these interesting EVM instructions:

- `call`: executes any public function on the contract specified.
- `callcode`: calls a function in a contract as it was part of the caller code.
- `delegatecall`: it's similar to `callcode`, but instead of creating a new internal transaction it modifies the current transaction to delegate handling to another contract's code.
- `selfdestruct`: destroys the current contract, and receives an address as a parameter for sending the current funds to.

Solidity is statically typed, supports inheritance, libraries and complex user defined types among other features. The keyword `public` is used when declaring state variables and functions. In case of state variables, it automatically generates a function that allows you to access the current value of the state variable from outside of the contract. In case of functions, it means that anybody can execute it by encoding its message call in a transaction; `external` behaves in a similar way.

The task of analyzing or verifying an Ethereum smart contract is difficulted by the facts that EVM bytecode features in general very little static information, which makes it extremely difficult to analyze; and that Solidity lacks formal semantics, so a Solidity program cannot be proven correct [7].

2.2 System verification

System verification is used to establish that the design or product under consideration possesses certain properties, that are mostly obtained from the system's specification [8]. A defect is found once the system does not fulfill one of the specification's properties. The system is considered to be "correct" whenever it satisfies all properties obtained from its specification.

Some verification approaches work with models that represent the targeted system, while others directly check the source code. Traditional techniques that target source code can be classified into static analysis and dynamic analysis [9]. *Static analysis* is the analysis of programs that is performed without actually executing the programs. The advantage of static analysis is that the complexity of the analysis only depends on the size of the source code and the analysis technique chosen, but it has a high false positive rate in practice [9]. In *dynamic analysis* of programs, an analyst needs to execute the target program in real systems or emulators. The advantage of dynamic analysis is the high accuracy, and among its disadvantages is that the complexity of the analysis depends on the execution complexity of the each target program.

To make a rigorous verification possible, properties should be described in a precise and unambiguous manner. This is typically done using a property specification language. Through this work we focus in particular on the use of a *temporal logic*. Temporal logic is basically an extension of traditional propositional logic with operators that refer to the behavior of systems over time. It allows for the specification of a broad range of relevant system properties such as:

- *functional correctness*: does the system do what it is supposed to do?;
- *reachability*: is it possible to end up in a deadlock state?;
- *safety*: "something bad never happens";
- *liveness*: "something good will eventually happen";
- *fairness*: does, under certain conditions, an event occur repeatedly?; and
- *real-time properties*: is the system acting in time?

We will specially focus in a type of temporal logic called Past Linear Temporal Logic (PLTL), both in our survey and to discuss our prototype.

Using temporal logic in the context of smart contracts is very natural. With classical logic we talk about relations between values; temporal logic uses classical logic to talk about relations across states. A state is a valuation of symbols, while a trace is a sequence of states. Not only smart contracts have well defined states: it's only possible to change state after successfully executing a whole interface function; but the sequence of states through which the contract went can be represented by each transaction placed in a specific order in the blockchain. PLTL is particularly useful because its counterexamples are finite traces, just like we would want if we think about counterexamples as exploits an auditor could find. We will introduce its syntax and semantics in the following section, and we will continue discussing PLTL in the following chapters.

Past Linear Temporal Logic

A formula in Past Linear Temporal Logic [10], or Past LTL, is inductively defined as either an atomic formula A or a logical or temporal connective applied to one or more formulae $\varphi,\ \psi$:

$$\varphi, \psi ::= A \mid \varphi \vee \psi \mid \neg\varphi \mid\ \bullet\varphi \mid \varphi\ \mathcal{S}\ \psi$$

Boolean connectives are interpreted in the standard way, while $\bullet\varphi$, read as "previously φ", evaluates the formula in the previous state if such exists and returns false otherwise; and $\varphi\ \mathcal{S}\ \psi$, read as "$\varphi$ since ψ", looks for a past moment k, $0 \leq k \leq \mathit{present}$, when ψ holds so that φ holds in all j moments, with $k < j \leq \mathit{present}$.

The since connective can be used to define other useful connectives like $\blacklozenge$, read as "once", meaning that the formula was or is true; and $\square$, read as "always" or "so far", to state that the formula must always hold.

3. A SURVEY ON THE STATE-OF-THE-ART OF AUTOMATED AUDIT TECHNIQUES

One of our main goals is provide to smart contract auditors a comprehensive report on their options to leverage program analysis during their day to day work. In the following we will discuss tools and techniques available so far aiming at Ethereum smart contracts analysis.

We decided to focus in approaches that can be used to find errors in already developed contracts; specially those with published or comprehensive papers, open source code, or mature projects. These selected approaches will be presented according to a classification inspired in a work by Grishchenko et al. [7]. Each category will be divided according to the main techniques employed; we provide an introduction to each general technique, and finally expand on the works oriented to smart contract that use it. Some works are just mentioned briefly, while we develop others in their own subsection, because we consider those specially interesting or mature.

Outside this selection there is a plethora of tools and techniques worth mentioning, because some of them constitute the building blocks of heavy weighted analysis approaches, they can complement them, or are related in some way to them.

Out of scope

Some programs attempt to extract information out of Solidity contracts like Slither [11], Surya [12], SolGraph [13], SolMet [14], RemixIDE [15], and EthersPlay [16]; Slither even attempts to find generic security issues and provides a flexible Python API to extend its analysis. Guth et al. developed an algorithm to generate an automaton that represents the contract for visual analysis out of its interactions in the blockchain [17].

There are also many works aiming at developing decompilers for EVM bytecode, not only to obtain a high level or intermediate representation but also to directly get a Solidity contract out of it. Among these are Rattle [18], Gigahorse [19], Porosity [5] (deprecated), JEB [20], EthIR [21], Octopus [22], and Erays [23].

New programming languages for smart contracts, focused on security and formal verification, are also emerging, such as Flint [24], Bamboo [25], Obsidian [26], and Simplicity [27]. Bamboo makes state transitions explicit, and its oriented to program verification; the authors plan to write an interpreter of Bamboo in Coq or Isabelle/HOL. Obsidian is also state oriented. Moreover Simplicity exhibits larger expressiveness yet allowing easy static analysis compared to EVM code, and comes with formal denotational semantics defined in Coq.

Additionally, interest was shown in dynamic monitoring or runtime verification in general, like Sereum [28], and ContractLarva [29]. Sereum is a technique designed to be included in Ethereum clients, while ContractLarva, given a contract and its specification, generates a new one that checks runtime behavior against the specification.

```
void aFunction(int a, int b) {
  int x = 1, y = 0;

  if(a != 0) {
    y = 3 + x;

    if(b == 0) {
      x = 2 * (a + b);
    }
  }

  assert(x - y != 0);
}
```

Fig. 3.1: Code example to illustrate symbolic execution trees.

Finally, there are some novel approaches worth mentioning: S-gram [30], and Raziel [31]. S-gram is a semantic-aware security auditing technique for smart contracts, and Raziel's authors propose a system for securely computing contracts guaranteeing their privacy, correctness and verifiability.

3.1 Automated analysis for bug finding

Most of the work in the literature belonging to this category is based on symbolic execution or fuzzing. So, the following sections will discuss the approaches relying on each of those methods, while tools that exploit mixed or different techniques are going to be mentioned at the end.

3.1.1 Symbolic execution

In a concrete execution, a program is run on a specific input and a single control flow path is explored. In contrast, symbolic execution [32] [33] can explore multiple paths that a program could take under different inputs. The key idea is to allow a program to take *symbolic values* as inputs. Execution is performed maintaining for each explored control path a formula that describes the conditions satisfied by the branches taken along that path, and a symbolic memory store that maps variables to symbolic expressions or values. Figure 3.1.1 presents a code example, and 3.2 shows its symbolic execution tree, along with its symbolic stores and path constraints.

The main goal of this technique, in the context of software testing, is to explore as many different program paths as possible in a given amount of time; then with each path we can generate a set of concrete input values that execute it, to finally check for the presence of errors. From a test generation perspective, it allows the creation of high coverage test suites, while from a bug finding perspective, it provides developers with a concrete input that triggers the bug, which can be used to confirm and debug the error independently of the symbolic execution tool that generated it.

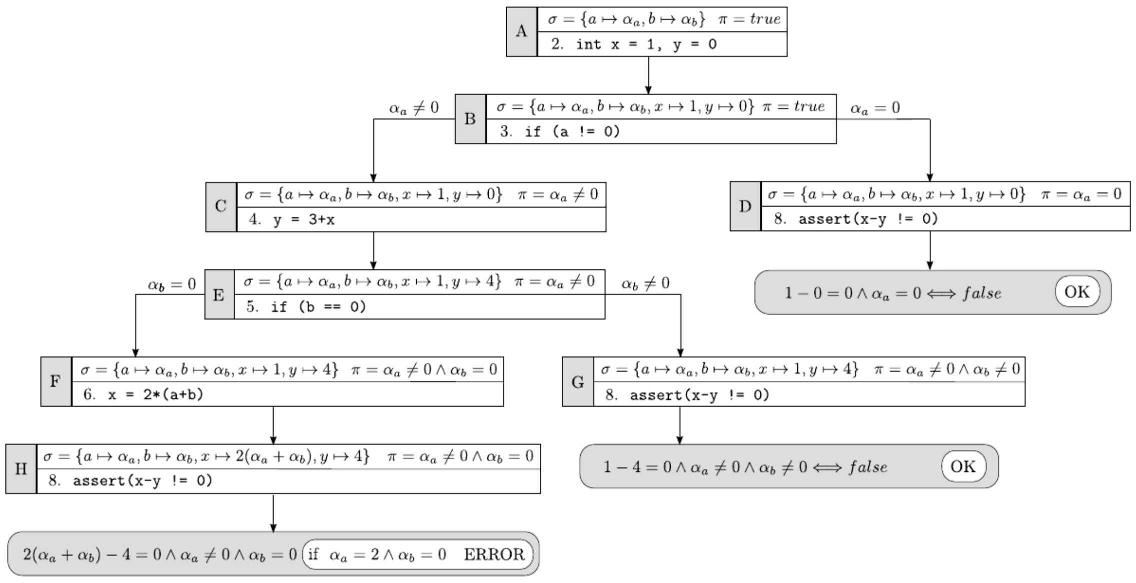
A
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$ $\pi = true$
2. int x = 1, y = 0
$\alpha_a \neq 0$
B
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = true$
3. if (a != 0)
$\alpha_a = 0$
C
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a \neq 0$
4. y = 3+x
D
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 0\}$ $\pi = \alpha_a = 0$
8. assert(x-y != 0)
$1 - 0 = 0 \wedge \alpha_a = 0 \Longleftrightarrow false$ OK
$\alpha_b = 0$
E
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0$
5. if (b == 0)
$\alpha_b \neq 0$
F
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
6. x = 2*(a+b)
G
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 1, y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b \neq 0$
8. assert(x-y != 0)
H
$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4\}$ $\pi = \alpha_a \neq 0 \wedge \alpha_b = 0$
8. assert(x-y != 0)
$1 - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b \neq 0 \Longleftrightarrow false$ OK
$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$ if $\alpha_a = 2 \wedge \alpha_b = 0$ ERROR

Fig. 3.2: Symbolic execution tree of function `aFunction` given in figure 3.1.1. Each execution state, labeled with an upper case letter, shows the statement to be executed, the symbolic store σ, and the path constraints π. Leaves are evaluated against the condition in the `assert` statement. This example was taken from [32].

Given a *path constraint*, meaning the boolean formula mentioned, the generation of concrete input values that satisfy it is typically based on an off-the-shelf satisfiability modulo theories solver, or *SMT solver*. Constraint solving is one of the main challenges faced by symbolic execution, SMT solvers can scale to complex combinations of constraints over hundreds of variables, however, constructs such as non linear arithmetic pose a major obstacle to efficiency and decidability.

Another challenge of this technique is state space explosion. Language constructs such as loops might exponentially increase the number of execution paths. It is thus unlikely that a symbolic execution engine can exhaustively explore all the possible paths within a reasonable amount of time. There are two key approaches that have been used to address this problem: heuristically prioritizing the exploration of the most promising paths, and using sound program analysis techniques to reduce the complexity of the path exploration [33].

Classical symbolic execution cannot deal with external code not traceable by the executor, nor with complex constraints solving. To cope with these issues concrete and symbolic execution can be mixed into *concolic execution*.

One popular concolic execution approach, known as *dynamic symbolic execution* [32], is to have concrete execution drive symbolic execution. After choosing an arbitrary input to begin with, it executes the program both concretely and symbolically. Whenever the concrete execution takes a branch, the symbolic execution is directed toward the same branch and the constraints extracted from the branch condition are added to the current set of path constraints. As a consequence, the symbolic engine does not need to invoke the constraint solver to decide whether a branch condition is satisfiable. In order to explore different paths, the path conditions given by one or more branches can be negated to generate a new input.

There are several approaches for bug finding in smart contracts that rely on symbolic execution. Oyente [34] was one of the first published approaches, it focuses on known security vulnerabilities. It was later extended as Gasper [35] to detect gas costly patterns, Osiris [36] to focus on integer errors, and SASC [37] to generate a diagram of invocation relationship and to find potential logic risks. Pakala [38] aims at efficiently handling multiple transaction counterexamples, it symbolically executes once each public function that produces a state change, to then join resulting states in all the possible orders up to a bound. It only looks for vulnerabilities related to sending ether from the contract under study to an address in control of an attacker.

Another tool that relies on symbolic execution is Mythril [39], it wasn't formally published but it's open source, it was informally introduced at HITBSecConf 2018 [40]. Later on, MythX [41] integrated Mythril with Harvey (a fuzzer described in section 3.1.2), and a new static analysis tool: Maru [42], along with Maestro, a component that syncs them all. Maestro also estimates the confidence of each vulnerability found according to what each tool reported. MythX is a private tool accessible through a paid subscription. Both Mythril and MythX only search for predefined security properties.

teEther

teEther [43] takes advantage of a common static analysis technique called *slicing* to reduce the state space explosion. A program *slice* is a subset of the original program statements relevant to a particular computation.

teEther's goal is check whether a contract is vulnerable to arbitrary ether transfers, thus it only cares about possible execution paths that end up in specific instructions executed with parameters under an attacker's control. These instructions, referred to as "critical instructions", are: `call`, `callcode`, `delegatecall`, and `selfdestruct`.

To take advantage of an unprotected `delegatecall` or `callcode`, for example, a `selfdestruct` statement could be placed on the callee.

This tool first explores the contract to find these critical instructions, then calculates slices that reach them, and finally performs symbolic execution on the slices to potentially generate transaction sequences that allow arbitrary ether transfers.

Another approach that implements a similar technique is sCompile [44], but we won't expand on it because it's neither published nor has a public implementation.

Manticore

Manticore [45] is an open source program developed by Trail of Bits [46], a company that specializes in software security audits. It's a dynamic symbolic execution framework for analyzing binaries and an arbitrary number of interacting Ethereum smart contracts.

Its core engine is the source of its flexibility; it implements a generic platform-agnostic symbolic execution engine that makes few assumptions about the underlying execution model. State exploration also contributes to its flexibility, because it implements a variety of heuristics to select the next state to explore and to determine when a state must be concretized.

Users are encouraged to create modular, event-based analyses using Manticore's plugin system. Arbitrary points within the application can broadcast various symbolic execution events that can be handled by an event subscriber outside of the tool.

The authors affirm that Manticore achieves on average 66% code coverage with a default analysis. This analysis was performed on 100 contracts taken from mainnet, but they didn't specify how and why those specific contracts were chosen.

3.1.2 Fuzzing

A fuzzing [9] program starts generating massive inputs for target applications, and tries to detect exceptions by feeding the generated inputs to the target applications and monitoring the execution states. To do so, fuzzers automatically start and finish the target program process for each input. Usually, the process stops at a predefined timeout, program execution hangs, or crashes.

Though a simple concept, many improvements have been proposed to increase the effectiveness and efficiency of the classical fuzzing approach.

With respect to the dependence on program source code and the degree of program analysis, fuzzers could be classified as *white box*, *gray box* and *black box*. White box fuzzers have access to the targeted program's source code, thus more information could be collected analyzing it. Black box fuzzers do fuzzing without any knowledge about the targeted program's internals. While gray box fuzzers work without source code, but gain internal information about the target through program analysis.

Among smart contract analysis tools we found a few that perform fuzzing in their analysis, like ContractFuzzer [47], ReGuard [48] to target reentrancy errors, and IFL [49].

Harvey

Harvey [51] implements a grey box fuzzer that instruments the contracts to gain knowledge about each input's execution path, to maximize branch coverage. Each potential test case, determined by a sequence of transactions, is mutated to produce a new one. The mutations consist in changing each transaction arguments, inserting a new function call somewhere in the sequence, and replacing prior calls with a new sequence. Besides these mutations, to increase coverage, it analyses branch conditions in terms of distance, based on its operators. For example, if the branch condition is `a > 27` then its distance could be calculated as $|27 - a + 1|$. Exploring how distances behave according to certain inputs, they can find inputs to take both execution paths, avoiding heavy-weighted techniques.

As figure 3.3 illustrates, if we know $distance(0) = 42$ and $distance(7) = 35$ then we can look for p' such that $distance(p') = 0$ using well known methods. Harvey uses the secant method as the figure shows.

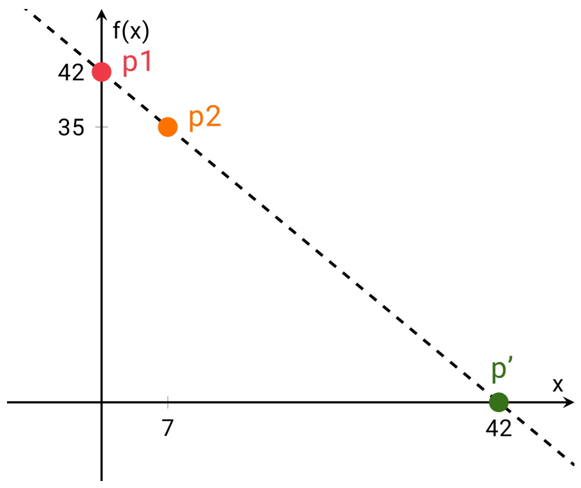

Fig. 3.3: Known points $p1$ and $p2$ let Harvey derive p' [50].

Echidna

Given a deployment specification, Echidna [52] performs white box property-based fuzzing. It is an open source implementation that provides an interface to indicate when an input found is interesting. At least one function with a special kind of name has to be added in the targeted contract returning a boolean, for Echidna to find a sequence of transactions that will generate a contract state where at least one of the added functions returns `false`.

3.1.3 Others

The company Synthetic Minds [53], based on previous work by them, developed a *program synthesis* tool that, given a specification that might refer to a target contract, generates a Solidity contract that satisfies it. Even though the authors did extensive work on that research area, their work related to Ethereum smart contracts is not public yet, neither its code or a binary. Feng et al. [54] also worked with synthesis, but in this case aiming at synthesizing transactions to exploit predefined vulnerabilities, with a tool called SmartScopy. It uses Vandal [55] as a decompiler (more on it in section 3.2.1) and extends its language to support symbolic variables and expressions. A naive algorithm to accomplish their goal is to enumerate all possible candidate programs and then symbolically evaluate each of them to check if it satisfies each query. But, because it doesn't scale, they introduce a technique called *summary-based symbolic evaluation*: each entry function is evaluated symbolically to generate a summary about how it could affect the contract state, meaning its storage and global properties. This summary is generated analyzing instructions like `sstore`, `call`, `selfdestruct`, and `delegatecall`, along with the path conditions that execute them, similarly to teEther. In this work they also explore some pruning and parallel algorithms to improve the overall efficiency, and over approximate the semantics of the `sha3` and `call` instructions to produce more tractable vulnerability queries.

Nikolić et al. presented Maian [56] a symbolic analysis tool that tries to find contracts that can be destroyed by anyone, that can send ether to anyone, or that have ether locked. Another tool out there, though behind a paywall, is Certora [57]. Their goal is to check common security errors and enable users to define new requirements with a code directive similar to an assertion. SmartCheck translates Solidity code to an intermediate representation based on XML and looks for XPath patterns [58].

3.2 Automated analysis for verification of properties

This line of research's goal is obtaining formal guarantees for the analysis results. As in the previous section, there are three techniques that cover most of the work in this category, so we will expand on those in the following sections.

3.2.1 Datalog

Datalog is a logic programming language that syntactically is a subset of Prolog [59]. A Datalog program [60] consists of an extensional database, which is defined by facts, and an intensional database, which is defined by rules. In the context of automated analysis, the program under study is represented with facts and its specification as rules.

It enables the development of program analyzers mostly independents of the program language details. If they change then only its translation to Datalog would have to be modified, its specification and related optimizations could remain the same.

Among the efforts to analyze smart contracts we found a few programs that rely on Datalog, like Vandal [55], which also provides a rich library to simplify the writing of specifications. Its authors also developed MadMax [61], an extension oriented to catch gas related vulnerabilities. MadMax consists of several analysis layers that progressively infer higher-level concepts about the smart contract under study; after the final layer, concepts like loops with unbounded mass storage can be expressed as undesired properties.

Another tool that aims at catching specific patterns on contracts using Datalog is Securify [62].

These mentioned approaches analyze bytecode, so its Datalog representation resembles assembly code, and both expose data and control-flow dependencies of the bytecode. The authors also focus their studies in known security vulnerabilities, like transaction order dependency and insecure value send.

3.2.2 Abstract interpretation

Abstract interpretation [63] of programs consists in using the program denotation to describe computations in a universe of abstract objects, so that the results of abstract execution give some informations on the actual computations. An intuitive example is the rule of signs. The program `-1515 * 17` may be understood to denote the computations on the abstract universe $\{(+), (-), (\pm)\}$, where the semantics of arithmetic operators is defined by the rule of signs. The abstract execution of the given program proves that the result is a negative number.

In order to perform abstract interpretation it's necessary to specify the abstract values, the flow function, and the initial state.

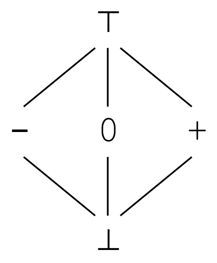

Fig. 3.4: Lattice representation of the abstract values defined for integer signs.

An *abstract value* represents a set of concrete values. To keep track of the sign of integer variables the abstract values can be defined as:

- $\top$, called "top";
- $-$ for negative numbers;
- 0 to represent the value of zero;
- $+$ for positive numbers; and
- $\bot$, called "bottom".

In figure 3.4 the set containment relationship is represented. The mathematical structure corresponding to that representation is called a *lattice*. A lattice consists of a partially ordered set in which every two elements have a unique supremum and a unique infimum.

Among other things, a lattice defines how to merge two abstract values. To merge the abstract values v_1 and v_2, we look for the lowest value in the lattice that covers both. This lowest value is called *least upper bound* and represented with the symbol $\sqcup$, like so: $v_1 \sqcup v_2$.

Abstract interpretation also requires us to maintain an abstract state and interpret each instruction. The *flow function* is in charge of the interpretation. The input is an abstract state and an instruction, and the output is the abstract state after executing the instruction. Thus, it describes the *abstract semantics* of every instruction type.

There is a mechanical procedure for deriving the flow function for a statement `x = a op b`, for any binary operator `op`. For each pair of possible abstract input values v_a and v_b:

- For every concrete value c_a covered by v_a and every concrete value c_b covered by v_b evaluate c_a `op` c_b: $C = \{c_a \texttt{ op } c_b \mid c_a \in v_a,\ c_b \in v_b\}$.
- Find the least upper bound that covers C, v_{lub}. Then, the flow function should have v_a `op` $v_b \rightarrow v_{lub}$.

For example, considering our lattice, to interpret `x = a - 1` with $a = +$ we evaluate c_a `- 1` for c_a in $+$. The result is "maybe zero", and the least upper bound is $\top$.

Finally, it's necessary to specify the initial abstract state. To determine the signs of integer variables we can define as an initial abstract state the state where the value of each variable is undefined, meaning that every variable is set to $\perp$.

The basic solving algorithm for this technique works on basic blocks; a *basic block* is a code sequence with no branches in except to the entry and no branches out except at the exit. In each step, we get the entry state of a basic block from its predecessors, then abstractly interpret the basic block, and finally set the new exit state of that block. If the exit state changes from the last interpretation of the block, then we need to rerun the successor blocks, because their state depends on the current block's exit state. Like this, we keep running blocks until no state changes from the last time. The resulting state is called a *fixed point*, because further abstract interpretation doesn't change it.

If the program under analysis has loops then we know that at least one of its basic blocks is going to be interpreted more than once. The first time the body of a loops is analyzed, it's interpreted as if it is executed only once, the next time as if it is executed either once or twice. A fixed point will be reached once further iterations produce no new behaviors.

Several works in the literature employed abstract interpretation to analyze smart contracts. Grishchenko et al. created EtherTrust [64], a tool that translates bytecode to Horn clauses, abstracting like so the small step semantic developed in their previous work (more in section 3.3), to later perform reachability analysis for two predefined security properties. *Horn clauses* here describe how an abstract configuration represented by a set of abstract state predicates evolves within the execution of the contract's instructions. This is the only work yet oriented to smart contract verification to have provided a formal proof for the soundness of their approach. The authors also claimed that EtherTrust can be easily extended to support other reachability properties.

Wüsdtholz et al. [65] extended Harvey to incorporate Bran, an abstract interpretation framework for EVM bytecode. Given a set of target code locations they attempt to discard paths that don't execute any of them, to finally use a fuzzer, Harvey, to generate test cases over the remaining paths. One of their goals is to avoid code instrumentation, therefore maintaining the contract semantic related to gas consumption. Bran choses some points in the program, referred to as "split points"; for each it analyzes its prefix using abstract interpretation to infer a postcondition. Using the postcondition it checks that the targeted locations are unreachable. Bran works directly on the bytecode, avoiding the expensive task of building a control flow graph[1] or decompiling. It leverages constant propagation[2] and integrates with Geth, the official implementation of the Ethereum protocol in Go, to execute instructions with constant arguments.

[1] A *control flow graph*, or CFG, is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

[2] *Constant propagation* is the use of control flow and data flow information to determine that a variable must have a particular constant value at a specific point in the program.

Predicate abstraction

Abstract interpretation maps a set of states S_C of a given domain to a state s_A in a different domain, where s_A represents a superset of S_C. When s_A consists of predicates it's called predicate abstraction [66].

For example, consider a formal specification of a program through logical formulae, where *pc* is a distinguished program variable that represents a program counter. To simplify the notation for transition relations we will define at_l as $pc = l$. Given the set of predicates $\{at\_l_1, ..., at\_l_5, y \geq z, x \geq y\}$ and the abstraction function α, we compute $\alpha(at\_l_2 \wedge y \geq z \wedge x \geq y)$ as follows. First, we check the logical consequence between the argument to the abstraction function and each of the predicates. The results are presented in the following table:

	at_l_1	at_l_2	at_l_3	at_l_4	at_l_5	$y \geq z$	$x \geq y$
$\alpha(at\_l_2 \wedge y \geq z \wedge x \geq y)$	$\not\models$	$\models$	$\not\models$	$\not\models$	$\not\models$	$\models$	$\not\models$

Then we take the conjunction of the entailed predicates as the result of the abstraction:

$$\alpha\left(at\_\ell_2 \wedge y \geq z \wedge x+1 \leq y\right) = at\_\ell_2 \wedge y \geq z$$

To improve the abstraction new predicates can be added, in a process called *abstraction refinement*. To select new predicates a counterexample guided procedure can be followed. It computes an over-approximation of the set of reachable states, and then checks if that set contains an error state. First, a postcondition is applied and then the result is over-approximated, like so $\alpha(post(state, predicates)) = post^{\#}$.

When a counterexample is found, for instance, over the path formed by p_1, p_3 and p_5, it's necessary to check that the path also reaches an error state without over-approximating, through the computation of $post(post(post(state_{init}, p_1), p_3), p_5)$. If the result value is false we need a refined abstraction function such that the execution along the counterexample path does not compute a set of states that contains some error states:

$$post^{\#}(post^{\#}(post^{\#}(\alpha(state_{init}), p_1), p_3), p_5) \ \wedge \ state_{error} \models false$$

To do so new predicates are added to the set, representing the intermediate steps of the condition:

$$\begin{aligned}
&state_{init} \models state_1\\
&post(state_1, p_1) \models state_2\\
&post(state_2, p_3) \models state_3\\
&post(state_3, p_5) \models state_4\\
&state_4 \wedge state_{error} \models false
\end{aligned}$$

The new set guarantees α and $post^{\#}$ are precise enough to avoid that erroneous counterexample.

VerX

VerX [67] [68] is a private tool that tries to prove a set of user defined Past LTL properties written in a Solidity-like syntax, given a set of Solidity contracts and a deployment script specifying the contracts constructors parameters.

When this tool tries to prove a given property φ on a contract C, C is instrumented with φ, it's inductively checked via symbolic execution, and in case of failure a fixed point is computed through a technique introduced by the authors called *delayed predicate abstraction*. The property is then checked on the fix point, ideally generating a proof or a counterexample. In case of failure more predicates are automatically added to the analysis, and the user is also requested to provide new ones.

For the time being only a fragment of Solidity is supported:

- Loops are bounded and no recursion is allowed.
- Gas exceptions propagate to the external transaction.
- No direct access to storage via assembly.
- No execution of external code (`callcode` nor `delegatecall`).
- No creation or destruction of contracts (`create` nor `suicide`).

They also assume contracts are *effectively external callback free* or EECF [69]. Informally, if an object o calls another object o', and this last one calls o, this last call is defined as a *callback*. The authors made this decision because they claim most real contracts are designed with this behavior as intended. Because of this decision the requirements formalization is simplified and a scalable and precise analysis is facilitated.

The target contract is instrumented with a representation of the specification predicates. For example, consider the following formula to illustrate how temporal operators are handled:

$$\square(\texttt{claimRefund}() \rightarrow \neg\blacklozenge(\mathit{sum}(\texttt{deposits}) \geq \texttt{goal}))$$

This term is transformed into the following, where p_ψ keeps track of whether the sum of the deposits has exceeded the goal in at least one of the previous states:

$$\square(\texttt{claimRefund}() \rightarrow \neg p_\psi)$$

The verification of this temporal property is then reduced to a reachability check of this new property we will call φ'; assertions with conditions like the last one are added to the original contract. Verifying that φ' holds in the instrumented contract is equivalent to show that φ holds in the original contract.

After this preprocessing the code is compiled to perform the symbolic execution and abstract interpretation on the resulting bytecode.

As I mentioned before, VerX first tries to verify the property inductively: assuming that it holds before a given function is called, with symbolic arguments and caller, it checks whether φ' holds after the function execution. If it does in the initial state and after all of the contract functions, then it can confirm $\Box\varphi'$ holds in the instrumented contract. If this verification fails, the property might still hold in C. For example, consider properties that require more than one transaction to exhibit its validity, like "after the crowdsale is started it's always going to end", in a classic crowdsale contract that requires a transaction to start it and another one to terminate it.

When performing delayed predicate abstraction, the set of abstract states reachable from the transaction boundaries, $reach(C_{\varphi'})$, is initialized with an abstraction of the initial concrete state defined by the deployment script. Then, it non deterministically choses a function, symbolically executes it, and computes the symbolic state at the end of the transaction: $p_1 \vee p_2 \vee \ ... \ \vee \ p_k$, where each p_i represents a possible function path. Each symbolic state p_i is transformed into an abstract state a_i, which results in a new set of states to include in $reach(C_{\varphi})$: $\{a_1, \ a_2, \ ..., \ a_m\}$. Many different symbolic states can be abstracted into the same abstract one, and thus $m \leq k$. A fixed point is computed over this new set of states, if the fix point implies the checked property then VerX can conclude that $\Box\varphi'$ holds in the contract. Otherwise, it tries to build a counterexample and requests the user to provide more predicates if it's unsuccessful again.

In the beginning of the analysis, the set of abstract predicates P is conformed by every atomic sub-formula in the specification. If it fails on the counterexample generation, P is automatically augmented extracting predicates from the program, requesting predicates to the user in case of a new failure. Predicates extracted from the code can be any of the following types:

- atomic formulae in the property;
- pointers to contract instances;
- enums/booleans, for instance, if an enum can hold any of two values, then $(anEnum == aValue \ || \ anEnum == anotherValue)$ will be included;
- constants; or
- predicates defined by users.

The authors also implemented their own symbolic execution engine following standard techniques, plus Solidity specific functions, like handling object allocation according to hash functions, contract calls and sound approximations of gas consumption. Object allocation using the hash function `sha3` is a challenge, it is a rather complex function and if modeled precisely with SMT constraints, it could cripple the symbolic execution. We will expand on VerX's modeling of `sha3` because, as we will see in section 3.4.4, it seems to work better than other models. VerX's authors also claimed that it handles hash collisions more efficiently that other symbolic engines.

In a smart contract the i-element in an array `a` of size `n` is going to be stored in the storage location with address `sha3(id(a)) + n * i`. So, to guarantee a valid location scheme the modeled `sha3` function must lack collisions, and its hashes have to be sufficiently spread apart. These requirements form the two principal new sets of restrictions for the solver. Avoiding collisions it's impossible because the domain of the function is greater that its codomain, so, injectivity in the whole domain is not enforced, they only consider each symbolic execution by its own; injectivity is guaranteed on the possibly symbolic arguments of the execution. These two groups of conditions hold in any Solidity contract, and that's why in-line assembly is not supported.

3.2.3 Model checking

Model-based verification techniques build upon models describing the possible system behavior in a precise and unambiguous manner. The system models are accompanied by algorithms that explore all states of the system model. When the exploration of all possible system scenarios is exhaustive the technique is usually called model checking [8]. In this way, it can be shown that a given system model truly satisfies a certain property, usually formalized in a temporal logic, like LTL. The property specification prescribes what the system should do, and what it shouldn't, whereas the model description addresses how the system behaves. If a state is encountered that falsifies the property under consideration, the model checker provides a counterexample.

Model checking is a general and sound approach but suffers from the state space explosion problem. It also verifies a system model, not the system itself, and only stated requirements; there's no guarantee of completeness, thus, it is often used for finding logical errors rather than for proving that they don't exist.

A technique to combat the state explosion problem is to represent the state space symbolically, and it is called *symbolic model checking* [70]. Another approach, *bounded model checking* [71], searches for a counterexample in executions whose length is bounded by some integer.

Zeus [72], for example, is a tool for smart contract verification that takes advantage of symbolic model checking. Desirable properties are expressed in a XML-like format, which are later used in a *taint analysis* [73] to efficiently place assertions throughout the code. To "taint an object" is to insert some kind of tag or label to it. The tag allows us to track the influence of the tainted object along the execution of the program. Thus, assertions could be place only where its truth value could have changed. Finally, the instrumented contract is translated to the LLVM framework [74] such that any verifier for that platform can be used.

There are also model checking approaches aiming at being included in the Solidity compiler, like the bounded model checker developed by Alt et al. [75] and solc-verify [76]. solc-verify's core is a translation from Solidity to the Boogie language, as the tool the next section is about. We won't expand on any of these two works because of their maturity.

Similarly, two other works also rely on a previously developed model checking framework, in this case called NuSMV [77]. In one of these works [78] the authors tried to translate contracts to NuSMV's language directly, concluding that a more precise model is needed, and claiming that such precision cannot be reached by a NuSMV model. Their proposed model consisted of three layers: one to capture the Ethereum blockchain behavior, another to model the smart contracts themselves, and a final one to model the environment for the application execution. While the authors of the second work, VeriSolid

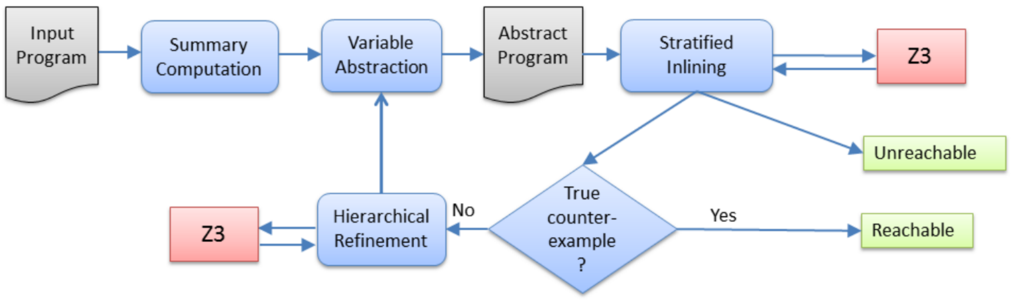

Fig. 3.5: Corral's architecture.

[79], claimed to have obtained positive results using the same framework. VeriSolid is an extension of a previous work by the same authors, a framework called FSolidM [80], which provides a graphical editor for specifying smart contracts as transitions systems and a Solidity code generator. VeriSolid introduces formal verification capabilities to the framework, thereby providing an approach for correct-by-design development of smart contracts. The dapp developer can use its web based graphical editor to design the contract and write down its specification in a natural language-like syntax, and the equivalent Solidity code of the contract can be automatically generated when all the specified properties are satisfied.

VeriSol

VeriSol's [81] technique consists mainly of three steps: it first translates Solidity contracts annotated with assertions to the Boogie language [82], then constructs a closed program, and lastly it uses bounded and unbounded verification techniques for Boogie programs. This language has the usual control flow constructs, unbounded types and operations such as arithmetic and uninterpreted functions. Moreover, this framework comes equipped with verification-condition, or VC, generation algorithms that can convert a call-free segment of code with an assertion to an SMT formula such that the latter is satisfiable if and only if there is an execution of the code that fails the assertion.

Corral [83] requires a closed program with a main method to obtain a well-defined verification problem. The process of closing an open Boogie program is called by the authors *harness construction*. In the case of a single contract C, they augment the Boogie program $\Delta(C)$ with a main procedure that first creates an instance of the contract by calling the constructor, and then repeatedly and non-deterministically invokes one of the public functions of C a bounded amount of times. While performing an analysis regarding multiple contracts requires creating first a meta-contract that represents the behavior encoded in all of them.

We can use VeriSol to look for a partial or full proof, or find a counterexample, where a counterexample corresponds to a sequence of transactions, where the first one creates an instance of $\Delta(C)$ and the last one is the first transaction in the sequence that reverts due to the failure of an assertion. To look for a proof it uses a type of predicate abstraction called *monomial predicate abstraction*, where a monomial over a set is defined as a conjunction of predicates belonging to it. It aims at automatically infer contract invariants satisfying the properties. Specifically, the contract invariant inference algorithm, Houdini [84], aims at constructing an inductive invariant over a set of candidate predicates, where an inductive invariant is a property each public function satisfies at the beginning and at the end of its execution, and that's also satisfied by the initial state defined by the constructor. The algorithm conjectures the conjunction of all candidate predicates as an inductive invariant and progressively weakens it based on failure to prove a candidate predicate inductive.

If VeriSol fails to verify full contract correctness, it employs Corral [83], a bounded model checker that also relies on Boogie. Corral uses a combination of abstraction refinement techniques, and requires a bound k for loop unrolling and recursive calls. Thus, it either finds a counterexample or verifies the lack of any on traces up to k transactions.

Corral has a two-level counterexample-guided abstraction refinement, or CEGAR, loop, illustrated in figure 3.5. The top level loop performs a localized abstraction over global variables. Next, it feeds the abstracted program to the module denoted *stratified inlining* to look for a counterexample. If stratified inlining finds a counterexample, it is checked against the entire set of global variables. If the path is feasible, then it is a true bug; otherwise, the module denoted *hierarchical refinement* is called to minimally increase the set of tracked variables, and then repeat the process.

For a single procedure program, stratified inlining simply generates its verification condition and feeds it to the SMT solver, Z3. In the presence of multiple procedures, instead of inlining all of them up to a given bound, only a few are inlined, its VC generated and Z3 is asked to decide its reachability. If it finds a counterexample then it's done; otherwise, it replaces every non-inlined call site with a summary of the called procedure, this over approximation of the program is then fed to Z3. A counterexample in this case, tells us which procedures to inline next. By default, Corral uses a summary that havocs all variables potentially modified by the procedure, but Houdini could also be used to compute summaries. A variable is said to be *havocked* if it's assigned a non-deterministic or unknown value. While hierarchical refinement takes a divide and conquer approach to the problem of discovering a minimal set of variables needed to refute an infeasible counterexample.

3.3 Frameworks for semi-automated proofs of properties

Program verification of smart contracts has been difficulted by the original EVM specification described by the Ethereum Foundation in their yellow paper [3], because it exhibits several under-specifications and does not follow any standard approach for the specification of program semantics. In order to provide a more precise characterization several works aimed at formalizing the EVM semantics to perform *deductive verification*, in some cases relying on proof assistants [7]. Ideally these formal semantics should constitute a sound over-approximation of the original one.

Deductive verification consists in expressing a program's correctness as a mathematical declaration and prove it. This approach is gaining popularity because of latests improvements on automatic theorem provers, specially from the SMT family.

Some of these mentioned approaches rely on *operational semantics*, a type of formal programming language semantic, in which desired program properties are verified building proofs from logic sentences about its execution. Others formalize programs using *small step semantics*, a type of operational semantics, that describes how individual calculation steps drive a system.

Grishchenko et al. defined the first small step semantics of the EVM [85] and formalized most of it in the functional programming language oriented to program verification: F*. Since F* code can be compiled to OCaml [86], they were able to validate their semantics against most of the official EVM test suite. Failed test cases, 320 out of 624, were related to yet unsupported functionalities.

Bhargavan et al. developed a tool to transform Solidity contracts into F*, and another one for a EVM bytecode to F* translation [87]. Both were implemented in OCaml and don't support loops nor many Solidity features. The authors claimed that even though their preliminary experiments suggest that F* is flexible enough to prove contract properties, static tools may be easier to use. Besides, their translation does not come with a justifying semantic argument, as Grishchenko et al.'s work.

Lem [88] is also being employed for smart contracts verification. Lem is a language for modeling semantic models using definitions that are translatable into OCaml for testing, and into popular interactive theorem provers, such as Coq, HOL4 and Isabelle/HOL. Hirai [89] proposed a formalization of the EVM with Lem and proved safety properties of some smart contracts in Isabelle/HOL. Lem's translation to OCaml enabled him to also test the executable part of his EVM definition against the official VM test suite. As a side effect, he discovered several problems in the specification; he requested eleven fixes to the yellow paper. Moreover, he found thirteen code paths in his model that the VM test suite did not touch. Building on top of this work, Amani et al. [90] enabled semi-automatically reasoning about correctness properties of EVM bytecode using Isabelle/HOL.

Another approach relies on Why3 to generate correct-by-construction smart contracts [91] and prove the quantity of gas used by the contract. Why3 is a tool to perform deductive verification, it has a logic language in which the specification can be expressed and proofs checked, and a programming language for smart contracts.

K Framework

K [92] is an executable semantic framework where programming languages, calculus, type systems and formal verification tools can be defined.

Based on the syntax and semantics of a language it generates a parser, an interpreter and formal methods' analysis tools, like a model checker and a deductive verifier.

K expresses correctness specifications with *reachability logic*, or RL, as reachability claims. RL is a sound logic tailored for reasoning about reachability. A reachability claim is a sentence of the form $p \implies q$ read as "p reaches q", where p and q are formulae in *matching logic*, or ML. This claim specifies that a set of states represented by p is going to reach a state of q if the execution completes, when executed with the given language semantics. Operational semantic rules and correctness specifications are treated by K's deductive verifier as reachability rules. Then, it uses RL's proof system to derive the proofs using semantic rules as axioms.

To briefly describe ML, and ML's formulae known as *patterns*, consider this rule example that aims at limiting the range of an integer variable:

```
rule chop(I:Int) => I
requires I >=Int 0 andBool I <Int pow256
```

This rule can be written as $a \wedge b \implies c$, where a, b, and c are patterns: `chop(I:Int)`, `I >=Int 0 andBool I <Int pow256`, and `I`, respectively. K's `require` introduces b as a condition, resulting in $a \wedge b$. This pattern can be thought of as the set of settings that satisfy b logically and match a structurally.

KEVM [92] is an EVM formal specification executable on the K framework. As any K specification, it includes three main components: the language syntax, a state or configuration description, and transition rules that drive the program's execution. Where configurations represent a state of the program's execution, as a unordered list of potentially nested cells, specified using an XML-like notation. These settings are divided into two parts: configurations related to the virtual machine containing information needed to execute transactions, and the ones related to the network state like information regarding accounts.

The authors mention two characteristics of the EVM that distinguish its assembler from a classical one, and difficult its formalization:

- Ethereum's yellow paper [3] specifies that no instruction can, during execution, cause an exceptional halt, as it was possible to detect exceptions before execution, and it is not. So, duplicate execution is necessary to avoid exceptions on runtime.
- Execution consumes an finite resource: gas.

Writing a program's specification at the EVM level can be tiresome and error prone so a domain specific language, or DSL, was developed, based on Ethereum's application binary interface, or ABI. Ethereum's *ABI* is a mechanism to simulate function calls at the VM level, it specifies encoding and decoding rules for the data sent on transactions, usually referred as *calldata*.

The procedure to verify a contract using KEVM consists mainly in:

1. Write down the high level logic of a contract as abstract state updates on the configuration.
2. Compile the code to EVM bytecode to use with KEVM.
3. Fill in the configuration from step 1 with the code corresponding to the specification, using a Hoare-like specification language.
4. If the claims cannot be proven, meaning the specification given in step 3 cannot be proven as true in the implementation defined in step 1, inspect the queries being sent to the SMT solver.

5. Fix issues found in the code or the claims in the previous step, so proofs can progress further.

6. Go back to step 4 if the proof fails again.

Among the downsides of this technique is worth mentioning that KEVM is 30 times slower than C++'s implementation of the EVM; as well as the multiple reasons why a successful contract proof could still hide a vulnerability. For example: theorem provers could be buggy as any other program, the VM's semantic might not match the implementations semantic, and domain information could be omitted in the specification of the contract.

3.4 Comparison of automated tools

Even though most surveys trying to assess the state of the art in smart contract verification focus in common security vulnerabilities, such as integer overflows and reentrancy, instead of functional correctness; its authors also reflect on interesting aspects of the topic. In this section we will expand on some these aspects discussed in previous work, on biases that affect many of these discussions, and finally we will share our thoughts and experiences on the subject.

3.4.1 Prior surveys

Dika [93] not only studies common bugs but also classifies issues found by OpenZeppelin auditors in publicly available audits, and compares tools according to its capacity to detect those. Their taxonomy of vulnerabilities splits errors into "blockchain", "EVM", and "Solidity". In "Solidity" they defined categories such as "reentrancy", "gas costly patterns", i.e. also common known security issues. For the tool comparison they selected Oyente, Remix, Security, SmartCheck, the F* framework, Mythril, and Gasper. The results from their experiments differentiate the security tools according to kinds of properties. In terms of effectiveness, SmartCheck outperformed the other tools with a score of 100% by successfully analyzing both vulnerable and audited contracts. Oyente, on the other hand, successfully analyzed the complete data set on bytecode analysis but performed slightly worse on Solidity analysis. When assessing accuracy, Oyente performed generally better than the rest of the tools although it did not perform well on the false negative rate. Finally, Securify performed worse compared to Oyente when assessing consistency.

A work by Parizi et al. [94] only considers contracts by Trail of Bits [46] and Ethernaut [95].[3] They don't use contracts included in evaluations from tool papers to make the evaluation fair minded; they only selected ten Solidity contracts with classic vulnerabilities and many instances of each type. The tools chosen for this propose were Oyente, Mythril, Securify, and SmartCheck. SmartCheck was shown statistically more effective than the other automated security testing tools at 95% significance level ($p < 0.05$). Concerning

[3] Ethernaut is a game developed by OpenZeppelin to learn about security in the context of smart contracts, the player has to hack a different contract in each level.

the accuracy, Mythril was found to be significantly ($p < 0.05$) accurate with issuing the lowest number of false alarms among peer tools.

In contrast with the empirical evaluation by Parizi et al., Di Angelo et al. [96] performed a completely qualitative comparison between: ContractLarva, E-EVM[4], Erays, EthIR, EtherTrust, FSolidM, KEVM, Maian, Manticore, Mythril, Osiris, Oyente, Porosity, Rattle, Remix, Securify, SmartCheck, Solgraph, SolMet, Vandal, S-gram, Gasper, ReGuard, SASC, sCompile, teEther, and Zeus. Their comparison is based in type of purpose (such as formal guarantees and security issues), level (bytecode versus Solidity), type (static versus dynamic), code transformations performed (disassembly, control flow graph, and more), analysis method (instrumentation, symbolic evaluation, model checking, etc.); similar to the aspects we mentioned about each tool in this chapter. Furthermore, they also analyze implementation details such as numbers of commits, active months, and affiliation. A tool is considered by them as a “publication tool” if the last noteworthy commit coincides with a date relevant to the conference where the accompanying paper was published. Their data suggests that academic tools tend to be publication tools. Functionality, usability and documentation are minimal, while the future of the tools remains uncertain. Notably, they relate the different empiric comparison presented by each tool’s creator in its paper; this is specially interesting considering that each author chooses a different subset of the remaining tools to evaluate their own. A distinction is also made between independent tool comparisons and those made by tool developers. These authors also use the capability to find common security issues as an evaluation criterion, but remark that standardized benchmark are desirable.

Likewise, Harz et al. [98] rely only on qualitative properties to compare both high level languages for smart contracts and verification methods, all of them similar to the ones considered in the surveys already mentioned.

3.4.2 Biases

The dramatic consequences of unsafe smart contracts are certainly a reality, however, many empiric results claimed by tool authors in their papers could be exaggerated by biases.

Perez et al. [99] studied ether distribution between contracts considered by several tools and confirmed that a very small amount of contracts hold most of the wealth; ether forms a Pareto distribution [61]. In this work they considered Oyente, Zeus, Maian, SmartCheck, Securify, ContractFuzzer, Vandal, and MadMax; after gathering data used by the authors of these tools, they noticed that the numbers of contracts analyzed usually did not match the data reported in the publications. Besides, contracts used by the different tools and vulnerabilities found rarely overlapped.

Another interesting point made by Perez et al. is that most papers only considered ether as an incentive to hack a contract, even though “real life value” could be represented in different ways.

A similar critique about ether distribution was pointed out by Trail of Bit’s CEO

[4] E-EVM is not included in our survey because it’s only a visualization tool. [97]

in a public communication channel[5], while commenting in particular about the paper that introduced Maian. He also remarked that the paper didn't discuss the fact that many contracts deployed to public blockchains are meant for testing; accounting for usage would yield better benchmarks and most papers don't do it. Solidity's changes can also cause issues, as he mentioned, in this case because all contracts compiled with solc versions smaller than 0.4.0 allowed any function to receive ethers. Since then, the behavior was changed, and only functions annotated with `payable` can receive ether. This semantic change affects Maian's results because one of its goals is to find ways to lock ether in contracts, and as we can see, this issue was way common before that Solidity change was introduced.

Grech et al. [61] discuss three threats to validity to their study to validate MadMax, that could probably apply to most works discussed here as well. Among them is the ether distribution issue already mentioned. Another common problem is the difficulty in establishing the base truth. Some techniques difficult checking the analysis results: each sample has to be manually inspected to establish whether a flagged vulnerability is indeed admissible. In addition to these two threats to validity or biases they mention that some vulnerabilities are actually not easy or cheap to exploit; sometimes exploiting a vulnerability even means blocking all ether in a contract, out of reach for the developer and the hacker.

To deal with ether distribution Brent et al. [55] recommend defining a contract's balance as the sum of the balances of all contracts with the same or similar bytecode. They also propose using number of basic blocks as a simple measure of complexity for smart contracts.

3.4.3 Qualitative analysis

For this comparison we choose a subset of the selection presented in our classification; we decided to ignore tools without publicly available implementations, those specialized in very specific vulnerabilities such as gas related issues, and semi-automatic approaches like theorem provers.

Considering there is no agreed upon benchmark to compare analysis tools specialized in Ethereum smart contracts, we decided to create a small set of contracts as an evaluation criterion[6]:

- **Loop**: forces the tool to iterate through a loop of one thousand iterations to find an error.
- **Trace**: it has to execute four transactions in a specific order to trigger an assertion failure.
- **Parameter**: if one of its functions receives an integer smaller than zero as an argument its assertion fails. The contract also has two almost identical functions but that don't present the same issue: one has the input parameter type changed to

[5] https://twitter.com/dguido/status/966795086704062465

[6] We used the last release version available for each program in July 15th 2019.

Tool	Loop example	Trace example	Parameter example	200 LOC example	2000 LOC example	Bytecode	Sound (FN)	Complete (FP)	Specific properties interface	Active development
Oyente	×	×	×	×	×	✓	×	×	×	×
teEther	×	✓	×	×	×	✓	×	✓	×	×
Pakala	✓	✓	✓	TO	TO	✓	×	✓	×	✓
Manticore	✓	✓	✓-	✓-	C	✓	×	✓	✓-	✓
Mythril	✓	✓	✓	×	✓	✓	×	✓	×	✓
MythX	✓	✓	✓	×	✓	✓	×	✓	×	✓
Echidna	✓	✓	×	×	C	✓	×	✓	✓-	✓
Maian	×	×	✓	×	×	✓	×	✓	×	×
VeriSol	TO	✓	✓	✓-	S	×	✓-	✓-	✓	✓
Securify	✓-	✓-	✓-	✓-	✓-	✓	✓-	×-	×	✓

Fig. 3.6: Qualitative comparison between automated tools with public implementations. "TO" indicates a timeout, "C" a crash, and "S" lack of support for a significant amount of Solidity statements used in the contract.

`uint` ("unsigned integer"), and the second includes a `require` statement to reject values smaller than zero. The idea behind this test is to asses parameter discovery and modeling issues.

- **200 LOC**: an adaptation of `PaymentSplitter` by OpenZeppelin.[7] If fails after any successful call to `release`.

- **2000 LOC**: an adaptation of `KittyCore` by CryptoKitties,[8] that fails after any successful call to `getKitty`.

Figure 3.4.3 shows these results, along with other quantitative aspects we consider desirables. Owing to Solidity's lack of formal semantics it's desirable for analysis approaches to work on EVM code instead of Solidity. It's also desirable for any analysis technique to be *sound* and *complete*, soundness avoids false negatives and completeness false positives. A lack of false negatives in this context is that all possible unsafe inputs are guaranteed to be found, while a lack of false positive means that all possible values deemed unsafe

[7] `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/payment/PaymentSplitter.sol`

[8] `shorturl.at/nyBMO`

are actually unsafe. Moreover, we ideally want programs to provide a friendly interface to write down formal specifications of contract specific properties. In the context of an auditor looking for programs to add to her tool set, it's also desirable to find active projects; we consider "active" a project that had any activity in the last six months. The table also splits tools horizontally according to the classifications of this chapter.

Some cells of the table, besides having a ✓ or × symbol, also include a + or a - symbol. In the case of Manticore it's because the tool was able to find a counterexample but had to be guided to do so; in `PaymentSplitter` we noticed it wasn't sending ether to the contract in any transaction so it always failed in a balance check, taking advantage of its flexible Python library we were able to add a constraint to make sure some ether was transfered. For this contract we also had to manually specify concrete arguments for the constructor to avoid a bug (that we reported[9]) at the analysis start. "Specific properties interface" also has a - because Manticore doesn't provide a language to write specifications, but it provides enough information during symbolic execution to nevertheless allow some contract specific analysis. For a similar reason Echidna has the same value in its properties cell: it doesn't provide a specification language but you can write any check you want inside the contract using Solidity code in an Echidna invariant function, and it will try to find a sequence of transactions that make it fail. Manticore and Echidna were also the only tools to crash with our examples; after spending several hours debugging Manticore we were able get closer to the bug that caused the crash but we were unable to fix it; we reported it[10] and even though they were able to reproduce it we haven't got any further information on it. To run `PaymentSplitter` in VeriSol we also had to make some small modifications in the contract because it's missing support for several Solidity statements. In the case of Securify we added - to all the code examples, even though we didn't have to do any special work to run them on it, because Securify also outputs a considerable amount of false positives, and it doesn't perform reachability analysis.[11]

Performing this assessment made us experience some of the annoyances that an auditor would face if trying to incorporate automated analysis to their work routine. A close relationship with the development team in charge of the program might be necessary to reduce the time overhead added by the primitive state of most tools. We also came to think that there's no clear "winner" among existing approaches.

3.4.4 Case study

To avoid some of the support and scalability issues of the tools, but still get a better idea of how they perform with production code, we decided to simplify a contract used in production and analyze it with the same programs we ran in the previous section. Considering the recent vulnerability[12] found by OpenZeppelin's auditors in a MakerDAO

[9] https://github.com/trailofbits/manticore/issues/1484
[10] https://github.com/trailofbits/manticore/issues/1477
[11] https://github.com/eth-sri/securify/issues/107
[12] https://blog.zeppelin.solutions/technical-description-of-makerdao-governance-critical-vulnerability-facce6bf5d5e

contract[13], we tried them on a simplified version of it.

Our first simplified version only had the original logic related to voting, because that's the main goal of the contract, and where the vulnerability was found, but we had to simplify it even further to reduce the work that had to be done to debug the tools we were trying to use: our final contract only allows users to vote for a single option, as figure 3.4.4 shows. SimpleDSChief does heavy use of a Solidity built-in type called `mapping`. Mappings can be seen as hash tables which are virtually initialized such that every possible key exists from the start and is mapped to a value whose byte representation is all zeros. However, it is neither possible to obtain a list of all keys of a mapping, nor a list of all values [6].

Our main goal was to find a way to easily express a property that represents how the voting system is supposed to work, assuming that an auditor would attempt to do the same, because that is the main feature of the contract and its correct behavior is critical. The best one that we could come up with using the contract's Solidity state variables is shown in figure 3.4.4.

Ideally, we would want to check that `approvals[anOption]` only has the sum of the valid votes for that option, but at this stage we didn't want to change the original contract even more, and we don't have a way to check that fact using the state variables available. The only mapping that we can access in every transaction is `votes` using `msg.sender`, because not all the public functions receive another address as argument to use as a mapping key; that's why we only check that pretty weak property. It's "weak" because it can hold and `approvals` contain an invalid value anyways, but if it's false we are certain there's an issue with the voting system.

We ran all the tools selected on the simplified contract extended with the invariant function, unsuccessfully in most cases:

- **Manticore**: It run out of memory quickly.
- **Oyente**: It reports that the assertion can fail, but to make sure that it isn't a false positive, we added `require(wad == 0)` in `lock` and it reports the same issue, so we confirmed it's just a false positive.
- **Securify**: We modified the contract to check for the property with the function shown in figure 3.4.4, being `aVar` a public state variable of the contract. This change was made because Securify tries to find some predefined patterns in the code, and one of those is unrestricted write to state variables. It reports the assignment as a warning, instead of a violation, so it can't guarantee that it's possible, and it doesn't give any further information.
- **Echidna**: We replaced the assertion with an Echidna function similar to what we did for Securify, and the tool couldn't find the issue.
- **Maian**: We replaced the assertion with a `suicide` statement, like in Securify's case, because that's what the tools searches for; it couldn't find the issue either.

[13] https://etherscan.io/address/0x8e2a84d6ade1e7fffee039a35ef5f19f13057152

```
contract SimpleDSChief {
    mapping(bytes32=>address) public slates;
    mapping(address=>bytes32) public votes;
    mapping(address=>uint256) public approvals;
    mapping(address=>uint256) public deposits;

    function lock(uint wad) public {
        deposits[msg.sender] = add(deposits[msg.sender], wad);
        addWeight(wad, votes[msg.sender]);
    }

    function free(uint wad) public {
        deposits[msg.sender] = sub(deposits[msg.sender], wad);
        subWeight(wad, votes[msg.sender]);
    }

    function voteYays(address yay) public returns (bytes32){
        bytes32 slate = etch(yay);
        voteSlate(slate);

        return slate;
    }

    function etch(address yay) public returns (bytes32 slate) {
        bytes32 hash = keccak256(abi.encodePacked(yay));

        slates[hash] = yay;

        return hash;
    }

    function voteSlate(bytes32 slate) public {
        uint weight = deposits[msg.sender];
        subWeight(weight, votes[msg.sender]);
        votes[msg.sender] = slate;
        addWeight(weight, votes[msg.sender]);
    }

    function addWeight(uint weight, bytes32 slate) internal {
        address yay = slates[slate];
        approvals[yay] = add(approvals[yay], weight);
    }

    function subWeight(uint weight, bytes32 slate) internal {
        address yay = slates[slate];
        approvals[yay] = sub(approvals[yay], weight);
    }

    function add(uint x, uint y) internal pure returns (uint z) {
        require((z = x + y) >= x);
    }

    function sub(uint x, uint y) internal pure returns (uint z) {
        require((z = x - y) <= x);
    }
}
```

Fig. 3.7: Simplified MakerDAO's DSChief smart contract.

```
    function checkAnInvariant() public {
        bytes32 senderSlate = votes[msg.sender];
        address option = slates[senderSlate];
        uint256 senderDeposit = deposits[msg.sender];

        assert(approvals[option] >= senderDeposit);
    }
```

Fig. 3.8: Invariant function for smart contract in figure 3.4.4.

```
    function checkAnInvariant(int aParam) public {
        bytes32 senderSlate = votes[msg.sender];
        address option = slates[senderSlate];
        uint256 senderDeposit = deposits[msg.sender];

        if (approvals[option] < senderDeposit) {
          aVar = aParam;
        }
    }
```

Fig. 3.9: Invariant function for smart contract in figure 3.4.4, adapted for a Securify analysis.

- **Pakala**: Likewise, we replaced the assertion with a `msg.sender.transfer(address(this).balance)` statement, but we were still unsuccessful.
- **Mythril**: It only found false positives.
- **MythX**: We had no luck here either.
- **teEther**: It run until the process crashed.

In finding this error we were luckier with VeriSol. We got a wrong trace at first (`free` → `checkAnInvariant`), and looking more into it we realized it was modeling uints as ints, so we added `require(anUint >= 0)` in most public functions of the contract and tried again. This time we got a strange trace, and we realized in this case it wasn't properly initializing the mappings: the votes mapping wasn't initialized with zeros, hence we added that into the Boogie representation of the contract that VeriSol generates. Similarly, we got another unexpected trace as output, and we noticed it was assuming that it could find a value such its hash would be a byte array with all zeros. In some cases that could be an issue but in this it wasn't, so we changed the Boogie representation again to avoid this and tried again. Finally it worked, we got: victim.`lock` → victim.`voteSlate` → attacker.`voteYays` → victim.`free`. It represents the issue found in the contract, even though this particular case would be very hard to exploit by an attacker, because the victim has to use the contract in a way that isn't intended to be used, and the attacker should know which option corresponds to the hash the victim used. After another minor change to the Boogie representation we got a different interesting trace: victim.`lock` → victim.`voteSlate` → victim.`etch`, that also shows the problem.

This case study is not meant to be a fair general comparison between our tool selection, because some of them are probably not intended to be used like this, but we think this is also a good illustration of the issues faced when trying to employ them.

After getting these results we posted a thread[14] in OpenZeppelin's forum to share our experience, and we received a lot of feedback from the Ethereum blockchain community; including replies from Echidna, Manticore, Pakala, and Mythril developers. Echidna developers not only commented that adding an extra function to the contract makes it possible to detect the bug, but they improved the tool so now it's not even necessary to do so. They achieved it by mining constant values from return values. In the case of Manticore, it was an issue related to how they model hashes as queries for their SMT solver. They told us through a private communication channel that a fix is almost ready and will be included in a release version soon. Mythril also failed because of how they modeled hashes, however after three months of work they finished making the changes needed, following a technique similar to VerX's, and redesigning a major project module. As of Pakala, its developer also mention that it was hard for Pakala to find bugs like that because of its hash modeling. He also found a minor bug related to this example, and was able to find the counterexample by running a version of DSChief simplified even further to use a single mapping.

[14] `https://forum.zeppelin.solutions/t/using-automatic-analysis-tools-with-makerdao-contracts/1021`

4. VERIMAN: POC OF YET ANOTHER TOOL FOR AUDITORS

This chapter focuses on the prototypes developed trying to benefit from the insights gained during the research described in the previous chapter: a hybrid tool that combines two existing ones, and an extension to write specifications our hybrid can analyze.

4.1 PoC I: VeriSol plus Manticore

First, we aimed at connecting a "trace generator" with a tool capable of concolic execution. Therefore, we could benefit from the scalable heavyweight analysis provided by the first program, while also avoiding false positives and obtaining a counterexample along with a concrete transaction sequence, thanks to the second tool.

Because of our experience researching the state of the art, we choose VeriSol and Manticore for this purpose, and so we called our program "VeriMan". Hence, given a contract containing assertions as input, VeriMan can analyze it with a standard VeriSol execution, and then parse the output files to obtain the counterexample trace, in case one was found. Then, a contract is created with Manticore, and each function in the trace is executed with symbolic parameters. Lastly, the final symbolic contract states are explored to find at least one error state, so it can be concretized and outputted with Manticore's style. That is, in a text file which shows each of the public function names and arguments conforming the trace, along with the transactions bytecode.

Even though VeriSol provides a trace with instantiated parameters this approach is still useful because VeriSol doesn't generates "blockchain ready" parameters. In its model all the Solidity variables are modeled as Boogie integers or references, it doesn't axiomatize types like `address` or `bytes`. Hence, it doesn't provide a mapping the other way round, meaning, it doesn't generate, for example, a concrete Ethereum address from the Boogie reference that represents it.

This prototype program is available in the history of VeriMan's GitHub repository. [1]

4.2 PoC II: Specification support

As already discussed, our focus is on providing guarantees about the functional correctness of contracts, like auditors aim to. To do so, a specification stating the intended behavior of the implementation is needed. Among the existing approaches to formalize requirements we choose temporal logic because of how contracts are executed in a sequence of transactions; properties can be interpreted over states that appear between transactions, while a contract state can refer to the values stored in the contract. This was also observed by some approaches discussed in the survey: VerX [68] and Scilla [100]. VerX is a private tool so we weren't able to compare our work with theirs, and Scilla is oriented for a different use, it is a new programming language for smart contracts.

[1] `https://github.com/VeraBE/VeriMan/tree/4188e7a86a40abf15b747fb649b67870d23f989a`

We can reduce the problem of verifying a temporal property to reachability checking by instrumenting the given contract with a representation of the property [101]. Previous work [102] [68] showed that Past LTL formulae (defined in section 2.2) are equivalent to assertions, and sufficiently expressive to capture any safety property; so our specification language will only support the temporal operators of PLTL. In conclusion, we aimed at providing a friendly interface with temporal operators for developers to define their project's requirements, or for auditors to test hunches or validate critical features. With this information is then possible to instrument the given smart contract with reachability checks equivalent to the temporal predicates conforming the specification; *instrumentation* here refers to adding extra instructions to the program source files. Therefore, the developer can take advantage of all the verification tools out there, and to come, that include assertion failures in their analysis. For instance, an engineer could analyze the instrumented contract with a fuzzer or a symbolic executer during development to aim at instant feedback about "easy to find" bugs, and later run heavyweight analysis programs like abstract interpreters or model checkers to get a full or partial proof, and more complex counterexamples.

Because of the analysis from the previous chapter, and the little work needed to do so, we decided to also add support for Echidna invariants; in addition to our previous integration between VeriSol and Manticore, that is still available in the tool. This approach is open source, as the last version in VeriMan's GitHub repository. [2]

Although we could have worked at the bytecode level, outputting an instrumented binary, we decided to produce a Solidity contract, because our first goal was to test it against VeriSol, and it requires a Solidity contract. Besides, we estimated this would be a better approach for a prototype, considering development times.

Before focusing in the implementation, is worth mentioning that this approach is based on the field of *runtime verification*. Runtime verification is a formal method that complements classical verification techniques with a more practical approach that analyses a single execution trace of a system, by extracting information from a running system and using it to detect and possibly react to observed behaviors satisfying or falsifying certain properties [103]. Even though our monitor checks properties in one trace at a time, if the resulting contract, for example, is analyzed by a static analysis tool then the instrumented properties will be checked in all the traces studied by the program.

The most common family of specification languages used for runtime verification is temporal logic; but other approaches employ regular expressions, state machines or other formalisms. For example, ContractLarva [29], as already mentioned in the previous chapter, is a runtime verification tool to monitor smart contracts on the Ethereum blockchain; it supports a state machine based formalism to specify behaviors. Given a Solidity contract and a specification file, it generates a new Solidity contract with the same functionality but augmented as specified in the specification file. Even though it is similar to our approach, we consider ours allows developers to think about contracts in a more intuitive way, by using temporal operators.

[2] https://github.com/VeraBE/VeriMan

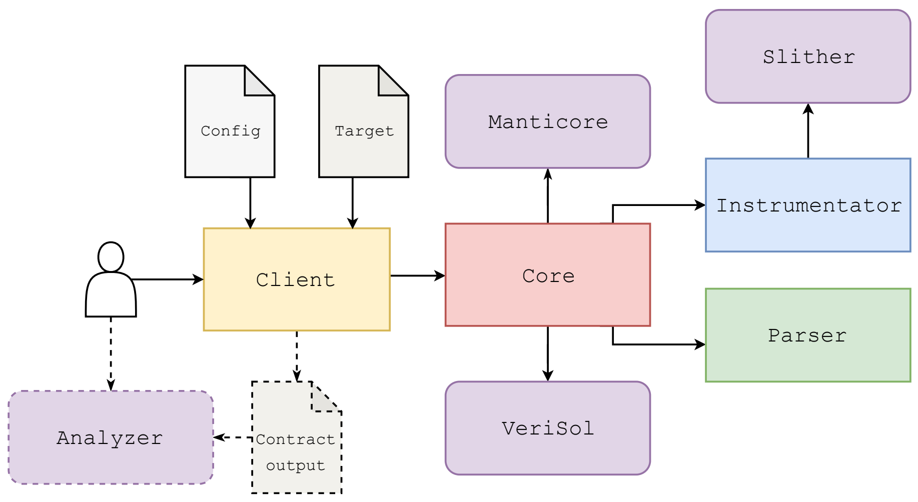

Fig. 4.1: VeriMan's architecture. Purple elements represent third party programs, while elements with dashed borders represent an alternative usage of the tool.

4.2.1 Implementation

VeriMan's implementation consists of four Python modules, as illustrated by figure 4.1 a client for console interaction, its core, a module in charge of the instrumentation, and a parser for the specification language. These modules interact with a JSON configuration file. There the users specify contract settings, meaning the path to the file containing the contract, the name of the contract of interest, and, if Manticore execution is desired, is also necessary to provide the concrete constructor parameters that would be used in its deployment. It is also the place for the users to write down the specification, along with the instrumentation and verification settings: to disable instrumentation in case only the VeriSol and Manticore integration is wanted, request Echidna invariants, disable VeriSol and/or Manticore, change the maximum counterexample length considered by VeriSol, and a few more implementation specific parameters.

Client

The client reads the configuration file, if the specification consists of several different predicates, and the verification and implementation features are enabled, it requests the core to pre processes the contract, and then requests a different analysis for each predicate; otherwise it requests only one analysis. Predicates are analyzed independently because VeriSol stops after finding a trace that makes an assertion fail, even when others can also be falsified. The same pre processing is used for all the predicates because is needed for the instrumentation, and it's very expensive computational-wise. We will expand on this step when the next module is described.

```
[.] Pre processing InOrder
[-] Pre processing time: 0.30290651321411133 seconds
[-] Will instrument to check: ['a_called -> num_calls > 0']
[-] Instrumentation time: 0.41573500633239746 seconds
[.] Running VeriSol
[!] Counterexample found: ['Constructor', 'a', 'b', 'b']
[.] Running Manticore
[...] Creating user accounts
[...] Creating a contract and its library dependencies
[...] Calling functions in trace
[...] Processing output
[-] Look for full output in: /home/user/veriman/output/1573524049_InOrder
[-] Analysis time: 5.792704343795776 seconds
[.] Cleaning up
```

Fig. 4.2: VeriMan's output in the console when analyzing the same contract as the original from figure 4.2.1.

Core

Analysis requests are handled by the core, its main goal is to merge the Solidity files needed to compile the contract under analysis, and coordinate the interaction between the instrumentation module, the VeriSol process and the Manticore API. It also records instrumentation and analysis times, and outputs progress information in the console if the `verbose` option is enabled. Figure 4.2.1 shows an output example.

Parser

Before to explain the instrumentation, we need to focus on the specification language and its parser. Our specification language supports most of the Solidity boolean and integer operators (`&&`, `||`, `==`, `!=`, `!`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, , `%`, and `**`), the basic PLTL temporal operators (`previously` and `since`), and two more temporal operators along with an implication symbol to simplify the writing (`always`, `once`, and `->`). The semantics of the temporal operators is as already discussed in section 2.2, while the syntax is as expected: `previously(p)`, `since(p, q)`, `always(p)`, and `once(p)`, where `p` and `q` are also predicates. Whereas variables can be formed by alphanumeric characters, dots and brackets, supporting like so Solidity's block and transaction properties (`this`, `block`, `msg`, `tx`, and `now`), enums (similar to enumeration types in C) and mappings.

While processing a predicate's abstract syntax three, the parser reduces implications into a new predicate, considering `p -> q` is equivalent to `!p || q`. It also reduces `once(p)` to `since(true, p)`, and `always(p)` to `!since(true, !p)`. Intuitively it can be seen that `always(p)` is equivalent to `!once(!p)`, which, according to the previous reduction, is equivalent to the final predicate. Finally, each predicate in the specification becomes an instance of the `Predicate` class, that has an attribute for the operator, and another for the nested predicates it has. For instance, if the specification is `aVar > 3 && anotherVar < 10` then the predicate instance operator will correspond to the conjunction operation, and the nested predicates will represent `aVar > 3` and `anotherVar < 10` respectively.

Instrumentator

If the specification predicates don't have temporal operators, then adding an assertion at the end of each public function with a conjunction of the conditions would be enough to detect a specification counterexample during execution. We could further improve the performance of this approach by adding a different assertion for each predicate; each only in the public functions that alter the value of its variables; including the constructor, because it creates the initial state.[3] Instead of changing function bodies, modifiers could also be employed; helping to avoid repeated code among other benefits. Modifiers are a Solidity feature that can be used to easily change the behavior of functions. For example, they can automatically check a condition prior to executing a function, or after it [6]. Despite this we will still alter each function's body, because some aspects of modifiers are not yet supported by VeriSol.[4]

As we already mentioned with classical logic we talk about relations between values; temporal logic uses classical logic to talk about relations across states. A state is a valuation of symbols, while a trace is a sequence of states. Because of this, to handle temporal operators, we need to keep track of states; thanks to using PLTL we only need to record information about previous states to evaluate an expression on the "present". Thus we will introduce new contract variables to keep track of the "past". As mentioned previously, every predicate will be reduced to a new one that only uses `previously` and `since` as temporal operators, so we only need to figure out how to handle those cases. It is worth mentioning that every predicate in the specification can be thought of as having an implicit `always` at the top, because we will want to check it holds in every state of the execution trace.

"Previously" operator During execution, the current state that a temporal logic term refers to will correspond to the final state after executing the current transaction, so the previous one corresponds to the contract state after the last transaction, which is equivalent to the state at the beginning of the new transaction. Hence, if there's a `previously(p)` somewhere in the predicate it suffices to declare a local variable `prev_p` at the start of each function to instrument, meaning where we consider an assertion must be added. The new variable will be initialized with the value of `p`, and the predicate term corresponding to `previously(p)` will be replaced with `prev_p` in the assertion at the end of the function.

"Since" operator As discussed before, `since(p, q)` holds when there was a moment in the past where `q` was true, and from the next moment included onwards `p` was true. Therefore we can keep track of two values to encode this formula. We will add two contract state variables for each `since` found in the specification under analysis: `q_held` initialized with `false`, and `p_since_q` with `true`. These values will be updated at the

[3] Consider what would happen if we tried to check `previously(!a_called) && a_called` in the original contract from 4.2.1 without considering the constructor as a function to instrument.

[4] A version using modifiers is implemented in a different branch of the repository for other uses.

```
contract InOrder {

    bool public a_called = false;
    bool public b_called = false;
    bool public c_called = false;
    int public num_calls = 0;

    constructor() public {
        bool VERIMAN_previously_nhixfelm=a_called;
        assert((!(VERIMAN_previously_nhixfelm))||(a_called));
    }

    function a() public {
        bool VERIMAN_previously_nhixfelm=a_called;

        a_called = true;
        num_calls++;

        assert((!(VERIMAN_previously_nhixfelm))||(a_called));
    }

    function b() public {
        bool VERIMAN_previously_nhixfelm=a_called;

        require(a_called);

        if(b_called) {
            num_calls = 0;

            assert((!(VERIMAN_previously_nhixfelm))||(a_called));
            return;
        }

        b_called = true;
        num_calls++;

        assert((!(VERIMAN_previously_nhixfelm))||(a_called));
    }

    function c() public returns(int) {
        require(a_called);
        require(b_called);

        c_called = true;
        num_calls++;

        return 3;
    }
}
```

Fig. 4.3: Instrumentation for `previously(a_called) -> a_called`.

```
    bool public a_called = false;
    bool public b_called = false;
    bool public c_called = false;
    int public num_calls = 0;

    bool VERIMAN_q_klagwfrt=false;
    bool VERIMAN_p_since_q_esidgsvh=true;

    constructor() public {
        if(VERIMAN_q_klagwfrt){
            VERIMAN_p_since_q_esidgsvh=b_called&&VERIMAN_p_since_q_esidgsvh;
        }
        VERIMAN_q_klagwfrt=a_called||VERIMAN_q_klagwfrt;
        assert((VERIMAN_q_klagwfrt&&VERIMAN_p_since_q_esidgsvh));
    }
```

Fig. 4.4: Fragment of the instrumentation for `since(b_called, a_called)` over the same contract as figure 4.2.1.

end of the functions to instrument, before the assertions. First, if `q_held` was true in the previous state (we haven't updated its value yet) then we update `p_since_q` by a conjunction between `p` and the current value of `p_since_q`. Then, we update the value of `q_held` by a disjunction between `q` and the current value of `q_held`. In this case, the term in the predicate corresponding to the temporal operator will be replaced by `q_held && p_since_q`; both conditions are required to represent its semantics.

Figures 4.2.1 and 4.2.1 show fragments of the resulting instrumented contract, when checking a predicate that has a `previously` operator, and another with a `since` operator.

Slither The instrumentator relies on Slither to parse the contract and get information in a malleable way, and interacts with the parser module to work with each specification predicate as an instance of the `Predicate` class, that could also have nested `Predicate` instances. Ideally, we should instrument by transforming the contract representation provided by Slither, generating the new Solidity contract from it at the end, but, because Slither doesn't provide the option to generate a Solidity contract from its representation, we decided to directly work on the string representing the file. Though not a desirable approach, it is enough to develop a prototype that could be improved later on. So, in the pre processing stage the file is read by Slither to create its representation, then its read into a string variable, that's splitted according to line breaks. When, for example, a new variable is needed in the contract, we use Slither's information to get the line and column numbers where the contract starts, to then add the declaration in the correct place of the corresponding element in the string array.

New variables Variables added by the tool start with `VERIMAN` and end with a random string of eight characters, to avoid keeping track of already created variables.

```
    function callsC() public returns (bool) {
        bool VERIMAN_return_value_jnmieetk=super.callsC();
        assert(a_var+b_var+c_var+d_var<10);
        return VERIMAN_return_value_jnmieetk;
    }

    function withTheSameName(uint aParam, bool anotherParam) public {
        super.withTheSameName(aParam,anotherParam);
        assert(a_var+b_var+c_var+d_var<10);
    }

    function withTheSameName(uint aParam) public {
        super.withTheSameName(aParam);
        assert(a_var+b_var+c_var+d_var<10);
    }
```

Fig. 4.5: Fragment of the instrumentation for `a_var + b_var + c_var + d_var < 10` in a smart contract with inheritance.

Inheritance The described recursive algorithm on the predicate is executed only over the main contract under analysis; inheritance is handled by first declaring in the main contract all the inherited functions with the same signature, but calling in its body to the same function with the same parameters but belonging to `super`. A similar approach is taken to handle implicit constructors: an empty constructor is added to the main contract if none is found. In compilation Solidity inlines all the declared constructors, in order according to inheritance, so this addition doesn't change the contract's behavior. Figure 4.2.1 illustrates part of an instrumentation over a contract with inheritance. Handling inheritance like this lets us ignore how Solidity's method dispatch works, it simplifies the implementation.

More on function selection Another aspect worth mentioning is that we also instrument at least one of the functions that end up outside of our criterion, because we can think of all of them as equivalent. The motivation behind this is that besides the case where the variables that affect an operator's value change during the transaction, we also want to make sure it holds when they are "stables". A special case among predicate variables are Solidity's block and transaction properties, these variables can potentially contain different values in each transaction. Because of this we will assume all of them can change value in every transaction; if a predicate contains one of them its assertion will be added to all the public functions.

4.2.2 Case study

Coming back to the MakerDAO contract we studied in section 3.4.4, it's worth noting that we can express the same weak invariant discussed using VeriMan's specification language. In our previous experience with this contract we had to make a few changes in the Boogie representation of the code, because of some bugs in VeriSol. Since

```
[.] Pre processing SimpleDSChief
[-] Pre processing time: 0.34923887252807617 seconds
[-] Will instrument to check: ['votes[msg.sender] != 0
    x00000000000000000000000000000000 -> approvals[slates[votes[msg.sender
    ]]] >= deposits[msg.sender]']
[-] Instrumentation time: 0.4664449691772461 seconds
[.] Running VeriSol
[!] Counterexample found: ['Constructor', 'voteSlate', 'lock', 'voteYays']
[-] Analysis time: 3.1317927837371826 seconds
[.] Cleaning up
```

Fig. 4.6: VeriMan's output in the console when analyzing the contract in figure 3.4.4.

then, all those bugs have been fixed, so we were be able to express the weak invariant with VeriMan and run it as figure 4.2.2 shows. We had to add `votes[msg.sender] != 0x00000000000000000000000000000000` as a precondition because, as we already mentioned, VeriSol's model assumes that it could find a value such its hash would be a byte array with all zeros. We thought it would be interesting to take a look at an example closer to real life usage, unlike 4.2.1.

5. CONCLUSIONS AND FUTURE WORK

Throughout this thesis we introduced Ethereum, along with an intuition about some difficulties faced when developing distributed applications that rely on it. In particular, those related to functional correctness of its smart contracts.

We studied theoretic approaches and tools to analyze smart contracts in various formats, and performed two qualitative comparisons, with the intention of understand how effective those approaches are. Firstly, we compared a selection of the available implementations, using some examples created by us, and considering its underlying algorithms. Secondly, we selected a production contract audited by OpenZeppelin and tried to take advantage of the previous selection of publicly available implementations to leverage their efficiency and understand the difficulties that automatic analysis carries with it.

We developed a prototype tool called VeriMan in two stages. In the first one we integrated VeriSol with Manticore; both tools showed promising results in our survey, so we tried to combine the benefits of both. In the second stage we created a specification language to check temporal properties with VeriSol and Manticore, or any program that detects assertion failures.

Future work includes creating a proper benchmark to compare analysis programs oriented to Ethereum smart contracts. There's also plenty of work pending related to VeriMan: upgrade the specification language to talk about function calls in its predicates (for example: "after each call to some function a condition holds"), allow properties that refer to interactions between multiple contracts, support invariants over aggregates (for example to write about the sum of the values of a mapping), support triggers (to say for instance that "a condition must be true before each call to `selfdestruct` in the contract"), and move instrumentation to a separate contract to avoid issues like loosing information about gas consumption in the original contracts. Finally, it would be interesting to compare performance with ContractLarva, VerX, and any tool that creates monitors for contracts. Gas consumption of VeriMan's output contracts could also be measured to ponder on the possibility of deploying contracts with these types of safety checks included.

BIBLIOGRAPHY

[1] S. Nakamoto *et al.*, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[3] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger,"

[4] "Openzeppelin's official website." https://openzeppelin.com/. Accessed: 2019-11-24.

[5] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," *DEF con*, vol. 25, p. 11, 2017.

[6] "Solidity documentation." https://solidity.readthedocs.io/. Accessed: 2019-11-21.

[7] I. Grishchenko, M. Maffei, and C. Schneidewind, "Foundations and tools for the static analysis of ethereum smart contracts," in *International Conference on Computer Aided Verification*, pp. 51–78, Springer, 2018.

[8] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[9] J. Li, B. Zhao, and C. Zhang, "Fuzzing: a survey," *Cybersecurity*, vol. 1, p. 6, Jun 2018.

[10] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Berlin, Heidelberg: Springer-Verlag, 1995.

[11] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15, IEEE, 2019.

[12] "Surya's repository." https://github.com/ConsenSys/surya. Accessed: 2019-10-23.

[13] "Solgraph's repository." https://github.com/raineorshine/solgraph. Accessed: 2019-10-23.

[14] "Solmet's repository." https://github.com/chicxurug/SolMet-Solidity-parser. Accessed: 2019-10-23.

[15] "Remix-ide's documentation regarding its static analysis." https://remix-ide.readthedocs.io/en/latest/static_analysis.html. Accessed: 2019-10-23.

[16] "Ethersplay' repository." https://github.com/crytic/ethersplay. Accessed: 2019-10-23.

[17] F. Guth, V. Wüstholz, M. Christakis, and P. Müller, "Specification mining for smart contracts with automatic abstraction tuning," *arXiv preprint arXiv:1807.07822*, 2018.

[18] "Rattle's repository." https://github.com/crytic/rattle. Accessed: 2019-10-23.

[19] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *Proceedings of the 41st International Conference on Software Engineering*, pp. 1176–1186, IEEE Press, 2019.

[20] "Jeb's official website." https://www.pnfsoftware.com/blog/Ethereum-smart-contract-decompiler/. Accessed: 2019-10-23.

[21] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *International Symposium on Automated Technology for Verification and Analysis*, pp. 513–520, Springer, 2018.

[22] "Octopus' repository." https://github.com/quoscient/octopus. Accessed: 2019-10-23.

[23] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, "Erays: reverse engineering ethereum's opaque smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 1371–1385, 2018.

[24] F. Schrans, S. Eisenbach, and S. Drossopoulou, "Writing safe smart contracts in flint," in *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming*, pp. 218–219, ACM, 2018.

[25] "Bamboo' repository." https://github.com/CornellBlockchain/bamboo. Accessed: 2019-10-23.

[26] M. Coblenz, "Obsidian: a safer blockchain programming language," in *Proceedings of the 39th International Conference on Software Engineering Companion*, pp. 97–99, IEEE Press, 2017.

[27] R. O'Connor, "Simplicity: A new language for blockchains," in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pp. 107–120, ACM, 2017.

[28] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[29] S. Azzopardi, J. Ellul, and G. J. Pace, "Monitoring smart contracts: Contractlarva and open challenges beyond," in *International Conference on Runtime Verification*, pp. 113–137, Springer, 2018.

[30] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 814–819, ACM, 2018.

[31] D. C. Sánchez, "Raziel: Private and verifiable smart contracts on blockchains," *arXiv preprint arXiv:1807.09484*, 2018.

[32] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[33] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later." 2010.

[34] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269, ACM, 2016.

[35] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 442–446, IEEE, 2017.

[36] C. F. Torres, J. Schütte, *et al.*, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 664–676, ACM, 2018.

[37] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–5, IEEE, 2018.

[38] "Pakala's repository." https://github.com/palkeo/pakala. Accessed: 2019-10-03.

[39] "Mythril's repository." https://github.com/ConsenSys/mythril-classic. Accessed: 2019-10-22.

[40] B. Mueller, "Smashing ethereum smart contracts for fun and real profit." https://github.com/b-mueller/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf. Accessed: 2019-10-22.

[41] "Mythx's official website." https://mythx.io/. Accessed: 2019-10-22.

[42] "The tech behind mythx smart contract security analysis." https://medium.com/consensys-diligence/the-tech-behind-mythx-smart-contract-security-analysis-32c849aedaef. Accessed: 2019-11-28.

[43] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 1317–1333, 2018.

[44] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang, "scompile: Critical path identification and analysis for smart contracts," *arXiv preprint arXiv:1808.00624*, 2018.

[45] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user friendly symbolic execution framework for binaries and smart contracts." https://github.com/trailofbits/publications/blob/master/papers/manticore.pdf, jul 2019.

[46] "Trail of bits' official website." https://www.trailofbits.com/. Accessed: 2019-09-12.

[47] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 259–269, ACM, 2018.

[48] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pp. 65–68, ACM, 2018.

[49] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," 2019.

[50] "Fuzzing smart contracts using input prediction." https://medium.com/consensys-diligence/fuzzing-smart-contracts-using-input-prediction-29b30ba8055c. Accessed: 2019-10-19.

[51] V. Wüstholz and M. Christakis, "Learning inputs in greybox fuzzing," *arXiv preprint arXiv:1807.07875*, 2018.

[52] "Echidna's repository." https://github.com/crytic/echidna. Accessed: 2019-09-13.

[53] "Synthetic minds' official website." https://synthetic-minds.com. Accessed: 2019-10-22.

[54] Y. Feng, E. Torlak, and R. Bodik, "Precise attack synthesis for smart contracts," *arXiv preprint arXiv:1902.06067*, 2019.

[55] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[56] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 653–663, ACM, 2018.

[57] "Certora's official website." https://www.certora.com/. Accessed: 2019-10-22.

[58] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 9–16, IEEE, 2018.

[59] W. F. Clocksin and C. S. Mellish, *Programming in Prolog: Using the ISO standard.* Springer Science & Business Media, 2012.

[60] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases.* Addison-Wesley, 1995.

[61] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 116, 2018.

[62] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 67–82, ACM, 2018.

[63] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977.

[64] I. Grishchenko, M. Maffei, and C. Schneidewind, "Ethertrust: Sound static analysis of ethereum bytecode," *Technische Universität Wien, Tech. Rep*, 2018.

[65] V. Wüstholz and M. Christakis, "Targeted greybox fuzzing with static lookahead analysis," *arXiv preprint arXiv:1905.07147*, 2019.

[66] R. Jhala, A. Podelski, and A. Rybalchenko, *Predicate Abstraction for Program Verification*, pp. 447–491. Cham: Springer International Publishing, 2018.

[67] "Verx's official website." https://verx.ch/. Accessed: 2019-09-11.

[68] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev, "Verx: Safety verification of smart contracts." To be published in IEEE S&P 2020, 2019.

[69] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proc. ACM Program. Lang.*, vol. 2, pp. 48:1–48:28, Dec. 2017.

[70] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 1020 states and beyond," *Inf. Comput.*, vol. 98, pp. 142–170, June 1992.

[71] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in Computers*, vol. 58, pp. 117–148, 2003.

[72] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts.," in *NDSS*, 2018.

[73] "University class on taint analysis." http://web.cs.iastate.edu/~weile/cs513x/2018spring/taintanalysis.pdf. Accessed: 2019-11-18.

[74] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[75] L. Alt and C. Reitwiessner, "Smt-based verification of solidity smart contracts," in *International Symposium on Leveraging Applications of Formal Methods*, pp. 376–388, Springer, 2018.

[76] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," *arXiv preprint arXiv:1907.04262*, 2019.

[77] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: A new symbolic model verifier," in *Proceedings of the 11th International Conference on Computer Aided Verification*, CAV '99, (London, UK, UK), pp. 495–499, Springer-Verlag, 1999.

[78] Z. Nehai, P.-Y. Piriou, and F. Daumas, "Model-checking of smart contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pp. 980–987, IEEE, 2018.

[79] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "Verisolid: Correct-by-design smart contracts for ethereum," *arXiv preprint arXiv:1901.01292*, 2019.

[80] A. Mavridou and A. Laszka, "Designing secure ethereum smart contracts: A finite state machine based approach," in *International Conference on Financial Cryptography and Data Security*, pp. 523–540, Springer, 2018.

[81] Y. Wang, S. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer, "Formal specification and verification of smart contracts for azure blockchain." April 2019.

[82] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*, pp. 364–387, Springer, 2005.

[83] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Computer Aided Verification* (P. Madhusudan and S. A. Seshia, eds.), (Berlin, Heidelberg), pp. 427–443, Springer Berlin Heidelberg, 2012.

[84] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *International Symposium of Formal Methods Europe*, pp. 500–517, Springer, 2001.

[85] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*, pp. 243–269, Springer, 2018.

[86] Y. Minsky, "Ocaml for the masses," *Queue*, vol. 9, pp. 44:40–44:49, Sept. 2011.

[87] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pp. 91–96, ACM, 2016.

[88] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell, "Lem: reusable engineering of real-world semantics," in *ACM SIGPLAN Notices*, vol. 49, pp. 175–188, ACM, 2014.

[89] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," in *International Conference on Financial Cryptography and Data Security*, pp. 520–535, Springer, 2017.

[90] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in isabelle/hol," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 66–77, ACM, 2018.

[91] Z. Nehai and F. Bobot, "Deductive proof of ethereum smart contracts using why3," *arXiv preprint arXiv:1904.11281*, 2019.

[92] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, *et al.*, "Kevm: A complete formal semantics of the ethereum virtual machine," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pp. 204–217, IEEE, 2018.

[93] A. Dika, "Ethereum smart contracts: Security vulnerabilities and security tools," Master's thesis, NTNU, 2017.

[94] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pp. 103–113, IBM Corp., 2018.

[95] "The ethernaut by openzeppelin." https://ethernaut.openzeppelin.com/. Accessed: 2019-11-30.

[96] M. Di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, IEEE, 2019.

[97] R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen, “Visual emulation for ethereum’s virtual machine,” in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–4, IEEE, 2018.

[98] D. Harz and W. Knottenbelt, “Towards safer smart contracts: A survey of languages and verification methods,” *arXiv preprint arXiv:1809.09805*, 2018.

[99] D. Perez and B. Livshits, “Smart contract vulnerabilities: Does anyone care?,” *arXiv preprint arXiv:1902.06710*, 2019.

[100] I. Sergey, A. Kumar, and A. Hobor, “Temporal properties of smart contracts,” in *International Symposium on Leveraging Applications of Formal Methods*, pp. 323–338, Springer, 2018.

[101] B. Cook, E. Koskinen, and M. Vardi, “Temporal property verification as a program analysis task,” in *International Conference on Computer Aided Verification*, pp. 333–348, Springer, 2011.

[102] G. Roşu, “On safety properties and their monitoring.,” *Scientific Annals of Computer Science*, vol. 22, no. 2, 2012.

[103] E. Bartocci and Y. Falcone, *Lectures on Runtime Verification: Introductory and Advanced Topics*, vol. 10457. Springer, 2018.