A Survey on Formal Verification for Solidity Smart Contracts

Ikram Garfatta
University of Tunis El Manar, National Engineering School of Tunis, OASIS
Tunis, Tunisia
University Sorbonne Paris North, LIPN UMR CNRS 7030
Villetaneuse, France
ikram.garfatta@lipn.univ-paris13.fr

Kaïs Klai
University Sorbonne Paris North, LIPN UMR CNRS 7030
Villetaneuse, France
kais.klai@lipn.univ-paris13.fr

Walid Gaaloul
Institut Mines-Télécom, Télécom SudParis, SAMOVAR UMR 5157
Évry, France

Mohamed Graïet
University of Monastir, Higher Institute for Computer Science and Mathematics
Monastir, Tunisia
National School for Statistics and Information Analysis
Bruz, France

ABSTRACT

One of the 21st century's hottest topics in the world of IT has been the emergence of what some predict to be the foundation stone for a new era of internet (web 3.0): Blockchain technology. Besides being the backbone of what we come to know as cryptocurrencies, Blockchain's features make for a bottomless list of possible applications, especially thanks to the concept of smart contracts. This, however, caused Blockchain to be in the limelight of not only interested investors but also malicious users who started hunting for this technology's vulnerabilities, which resulted in numerous attacks on different Blockchain platforms. In an attempt to mend such loopholes, researchers took an interest in the verification of smart contracts, which are at the heart of Blockchain's applications. In this survey, we aim to present a general overview of the different axes investigated by researchers towards the verification of smart contracts, while taking a special interest in studies that focus on formal verification, the different approaches they apply and vulnerabilities they target.

CCS CONCEPTS

• **General and reference** → **Surveys and overviews**; • **Security and privacy** → *Formal methods and theory of security*; • **Software and its engineering** → *Formal methods*; *Formal software verification*; *Model checking*.

KEYWORDS

Blockchain, Formal Verification, Smart Contract, Ethereum, Solidity

ACSW '21, February 1–5, 2021, Dunedin, New Zealand

ACM ISBN 978-1-4503-8956-3/21/02…$15.00
https://doi.org/10.1145/3437378.3437879

ACM Reference Format:
Ikram Garfatta, Kaïs Klai, Walid Gaaloul, and Mohamed Graïet. 2021. A Survey on Formal Verification for Solidity Smart Contracts. In *2021 Australasian Computer Science Week Multiconference (ACSW '21), February 1–5, 2021, Dunedin, New Zealand*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3437378.3437879

1 INTRODUCTION

Despite the fluctuation in the values of cryptocurrencies, the growing wave of adoption of blockchain-based distributed ledgers has not yet known any ebbing ever since its inception led by the well-known Bitcoin. While the first-generation blockchains were focused mainly on cryptocurrencies, a new generation emerged later, which embeds the distributed ledgers by so-called smart contracts that enable them to function as distributed computing platforms. These smart contracts are, however, often riddled with vulnerabilities. In fact, the most prominent Blockchains have been far from immune to the ill-intentioned attackers especially with the added monetary lure to the mere feeling of satisfaction they get from hacking. The first dangerous attack on a blockchain can be traced back to August 2010, when 92 billion BTC were generated out of thin air by exploiting an integer overflow vulnerability in the Bitcoin blockchain, which resulted in cancelling all relevant transactions and rolling back the blockchain to a previous state. The DAO attack in June 2016, caused by a reentrancy vulnerability, is one of the most infamous attacks Ethereum has ever had to suffer. On top of the tangible loss that evaluated to 3.6M of stolen ether (around 55M USD at the time) the attack resulted in a hard fork in the Ethereum blockchain which could have easily resulted in a community fallout, the worst possible nightmare for Ethereum. The Parity multisig wallet has been subject to two substantial attacks. The first happened in July 2017 when more than 150K ETH were stolen (32M USD). The attacker used a vulnerability in the code (a bad practice) that allowed him to change the ownership of an important contract and take possession of its ether. The second attack which happened in November 2017, did not result in stolen funds but caused 513K ETH to be locked in the attacked contracts (160M USD).

From an academic point of view, numerous methods and tools have therefore emerged to both support the development of secure smart contracts and aid the analysis of already deployed ones. This panoply of studies comprises approaches that use non-formal techniques to detect bugs in certain execution scenarios, as well as approaches based on formal techniques and aim for an automatic formal verification of smart contracts. While informal techniques can test a certain requirement under certain scenarios, they cannot prove the correctness of a smart contract in general. That's why researchers turned to formal verification which has proved to be efficient to reach such correctness goals.

In this paper, we provide an assessment of the state of the art that covers studies that fall under this second category. Hence, we focus on studies that propose formal models (e.g., automata, transition systems) and use formal techniques (e.g., model checking, theorem proving). We categorize and evaluate these studies based on two criteria: the employed verification techniques and the targeted vulnerabilities. Our intention behind this paper is not to merely identify relevant work on formal verification of smart contracts, but to also pinpoint the weaknesses and limits, whether common or individual, of the proposed approaches. We are then able to point out improvement areas to explore in future work.

We organize our paper as follows: we start by a broad overview of the Blockchain technology and common vulnerabilities in smart contracts in Section 2. In Section 3 we identify various studies touching on the formal verification of smart contracts, among which we select a few to present in detail in Section 4. They are then compared and discussed in Section 5. The paper is concluded in Section 6.

Methodology for Collecting Existing Works: In this paper, we tried to put together an exhaustive list of approaches for the verification of smart contracts. We resorted essentially to two sources in our quest, namely the Google Scholar search engine and the DBLP computer science bibliography, and used combinations of the following keywords for the search: smart contract, formal verification, Solidity and Ethereum. We then recursively pursued the references included in these papers' related work citations. We came out with a plethora of material, from which we selected 13 studies to include in our survey on the basis of their relevance to the subject, the uniqueness of the proposed approach and the number of citations. Our selection was also guided by previous surveys that were conducted on more generic scopes. For instance, some studies did not focus on the formal aspect of the proposed verification approaches for Solidity smart contracts but rather on their analysis capabilities [5], while others chose to cover more smart contract languages [16] to the detriment of being exhaustive in their papers selection. Unlike such surveys, we choose to consider Solidity as the most used smart contract language and focus on verification proposals that use formal approaches, offering a more in-depth analysis of the existing formal verification approaches for Solidity smart contracts.

2 BACKGROUND

In the largest sense of the word, a Blockchain is a distributed ledger, designed as an append-only list of so-called blocks which are used to record valid transactions between different parties. Cryptography is used to establish the links between the Blockchain's blocks, and a consensus protocol governs the blocks validation process (i.e., mining).

Ethereum is one of the leading public Blockchain plat-forms nowadays. It supports the notion of smart contracts which practically makes it a distributed computing platform. Smart contracts take the famous saying "code is law" into a new perspective where law becomes code. They can be seen as the equivalent of contracts written on paper, where the agreed upon terms are transcribes in lines of code. The faithful execution of a smart contract is guaranteed by the laws of the Ethereum Virtual Machine (EVM) semantics and its immutable nature gives it a sense of finality.
The most commonly-used high-level programming language for Ethereum smart contracts is Solidity. A Solidity smart contract is a collection of code and data, residing at a specific address on the Ethereum Blockchain, which can be invoked either by an internal account (i.e., a smart contract) or directly by an external account (i.e., user). Every account is characterized by a persistent *storage* (null in case of an external account) and a balance in Ether which is adjusted by transactions. A transaction is a message used to send ether from one account to another and/or invoke a smart contract's function if the message includes a payload and the targeted account is an internal one. The execution of such a payload is carried out according to a *stack* machine called the EVM. Every smart contract features a *memory* and can access certain properties of the current block (e.g., number, timestamp...). Besides the storage, stack and memory, Ethereum has an externally accessible indexed data structure that can be used to implement events and acts as a *log*.

A Solidity smart contract may look like a JavaScript or C program syntax-wise, but as appearances are often deceiving, they are actually dissimilar since the underlying semantics of Solidity functions differently from traditional programs. This naturally calls on more vigilance from the programmers who might be faced by unconventional security issues. According to [6], vulnerabilities in smart contracts seem to often stem from this gap between the semantics of Solidity and the intentions of the programmer.

In the following we list and explain the most common vulnerabilities exploitable by attackers. We will be using the illustrative contracts in Listings 1 and 2 to give examples of such vulnerabilities. We note that these contracts are written for illustration and do not exhibit a logical functionality.

Limited stack size: the call stack of the EVM has a maximum size of 1024 frames which, once reached, would cause the next function call to fail along with its subcalls. An attacker could exploit such a limitation by generating a number of calls to the vulnerable contract as to almost fill the stack, counting on the targeted contract to mishandle (or not handle at all) the incurred full stack failure, and use the next function call to exploit this pitfall. We note that the changes introduced in Ethereum's hardfork in October 2016 make this call stack limit practically unreachable. We still mention this vulnerability for awareness.

Wrong arithmetic/conversion handling: Solidity's mathematical operators do not implement safeguards, and consequently, errors such as overflows and underflows may occur due to the violation of value limitations of integer data types in the results of such operations. For instance, the Multiply function in VulnContract would return 44 instead of raising an exception if executed with (3, 100) as input. This is due to the expected result (i.e., 300) being larger

than the maximum value of the type uint8 (i.e., 255) and Solidity's wrapping in two's complement representation for integers.

Timestamp dependence: since the execution on a Blockchain needs to be deterministic for all the miners to get the same results and reach a consensus, users usually resort to block-related variables such as timestamp as a source of entropy. Sharing the same view on the Blockchain, miners would generate the same result, albeit being unpredictable. Even though this seems to be safe, it gives the miners a small room for manipulation given that they can choose a timestamp within a certain range for the new block, which gives them the possibility to tamper with the results and put some bias towards a certain user for example. Such a vulnerability can be exploited any contract relying on a time constraint to determine its course of action. In our example, the function NewYear() in Listing 1 is timestamp-dependent.

Costly loop: executing operations on the EVM costs gas. A contract invocation can only carry on if the amount of gas sent by the user along a transaction is sufficient, which means that costly loops can easily fail. Such a vulnerability can be exploited if an attacker gets his hands on a mapping or array data structure to externally manipulate and drive the execution to failure. In our example, CostlyLoop() in Listing 1 can fail if it gets called with a very large input.

Reentrancy: this is by far the most notorious vulnerability since it led to the infamous DAO attack. An attack of this type can take several forms (e.g, we can talk about a single function reentrancy attack or a cross-function reentrancy attack), but the main idea behind it is that a function can be interrupted in the middle of its execution and then be safely called again before its initial call completes. Once the second call completes, the initial one resumes correct execution. The simplest example is when a smart contract uses a variable to keep track of balances and offers a withdraw function. A vulnerable contract would make a transfer of funds prior to updating the corresponding balance which an attacker can take advantage of by recursively calling this function and eventually draining the contract. This can be illustrated by a call to the function ReentrancyAttack() in Listing 2 which would start by sending some Ether to the VulnContract in Listing 1 by invoking its Deposit function and then asking for it back by invoking its Withdraw function. The VulnContract proceeds by sending the Ether which invokes the fallback function of the MaliciousContract. This is where the control flow is handed over to the latter contract. The fallback function recursively calls Withdraw(), which is allowed since the condition on its balance is still valid, until VulnContract is drained.

```
contract VulnContract {
  mapping (address=>uint) balances;
  uint256 result;
  event started();
  function Deposit() {
    balances[msg.sender] += msg.value;
  }
  function Withdraw(uint amount) {
    if(balances[msg.sender] >= amount) {
      msg.sender.call.value(amount);
      balances[msg.sender] -= amount;
    }
  }
  function Multiply(uint8 x, uint8 y) returns (uint8) {
      return x * y;}
  function NewYear(){
    if(block.timestamp > 1609459199)
      emit started();}
  function CostlyLoop(uint256 x) {
    for(uint256 i = 0; i < x; i++)
      result += i;
  }
}
```

Listing 1: A vulnerable smart contract in Solidity

```
contract MaliciousContract {
  uint balance;
  VulnContract vc = VulnContract(0xbf0061dc...);
  function ReentrancyAttack() {
    balance = msg.value;
    vc.Deposit.value(balance)();
    vc.Withdraw.(balance);
  }
  function () payable {
    vc.Withdraw.(balance);
  }
}
```

Listing 2: A malicious smart contract in Solidity

3 STATE-OF-THE-ART ON SMART CONTRACTS ANALYSIS: A SYNOPSIS

The different attacks on the different Blockchain platforms brought light on the various vulnerabilities that they may suffer from drove experts to work on finding suitable solutions for such weaknesses. The efforts put into this quest took different directions. Some solutions were based on informal methods while others aimed for more formal verification approaches (see Figure 1).

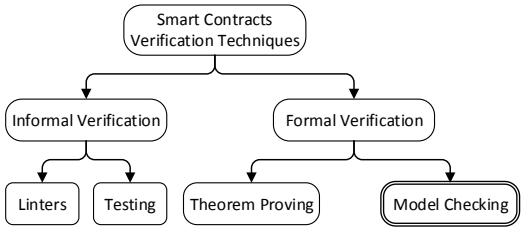

Figure 1: Smart Contracts verification techniques

3.1 Informal Techniques

Informal techniques are usually associated with validation rather than verification. The most common techniques that fall under this category are testing and simulation. In fact, one straightforward way to minimize the risk of deploying a vulnerable smart contract is to take advantage of one of the many existing testnets which are, as their name suggests, alternate blockchains dedicated for testing purposes. A smart contract can, for example, be run on Ropsten [2] before its deployment on the mainnet, which may help with plain defects but not with imperceptible ones.

The Solcover testing tool [28] was developed to offer a free and

automatic testing experience of smart contracts. According to the tool's associated blog article, it "should only be treated as another arrow in a collective quiver", as it is unable to fully ensure the correctness of a smart contract.
Instead of trying out scenarios that may or may not instigate erroneous behaviors, other researchers worked on enforcing security and best practices rules through linters [15], which are tools that analyze the code to identify and flag programming and stylistic errors or suspicious constructs.

3.2 Formal Techniques

While informal methods may reduce the risk of bugs in smart contracts, relying solely on such techniques cannot be enough to get a full insurance that a smart contract would be correct. Formal verification techniques can overcome this weakness, though it may come at the expense of other challenges such as scalability. Such techniques are used to check the conformance of the developed system to a predefined specification. They are based on formal methods of mathematics and are able to provide formal proof of the correctness of the investigated system with reference to its formally specified behavior. We can distinguish mainly two families of formal verification methods, namely those based on *theorem proving* and those based on *model checking*.

3.2.1 Theorem Proving. In this branch of formal verification techniques, the model of the system, along with the properties to prove need to be modeled mathematically. A theorem prover is then used to generate proofs and discharge them by applying axiom and inference rules. Several logics can be used in theorem proving (e.g., first-order logic, propositional logic). Depending on the complexity of the system, the user may need to interact with the theorem prover to discharge proofs. Thus, verification approaches based on theorem proving are seldom totally automated.

Some researchers proposed theorem proving-based approaches for the verification of Solidity smart contracts. The authors in [8] propose Solidity* a prototype tool, implemented in OCaml, that allows the translation of a restricted subset of Solidity into F*, a functional programming language for program verification. In order to detect dangerous patterns, the user then needs to define effects in F* code which are discharged by the F* type-checker. They also propose EVM*, a decompiler for EVM bytecode into F*, along which they propose a model for the cost of bytecode operations which can be used by creating annotations for gas-related violations that can be discharged by the F* type-checker. Using this approach requires, not only expertise in F*, but also an understanding of the proposed translation in order to be able to express the patterns to be checked and understand the generated typechecking errors. Members of the Ethereum community present a prototype for verification integrated into the *Solc* compiler of Solidity [1]. Their proposition leverages the Why3 IDE, a theorem prover which can be used on the WhyML code generated by calling *Solc* with specific attributes. Other partial translations of the EVM bytecode based on assisted proofs like Coq [18] and Isabelle/HOL [3] exist. We note that none of these approaches offer automatic verification of smart contracts.

3.2.2 Model Checking. The goal of model checking is to verify that properties (specified in a temporal logic) are satisfied w.r.t a system (represented as a finite-state model). The general idea here is to construct the state space of the model, and to explore it in order to check a specification that is supposed to define the correctness of the system, and potentially generate counterexamples in case the specification was not met. The *standard* approach to do that would be to generate all the reachable states of the system, represent them individually and then exhaustively explore the state space to check for the specified property. The application of such a method would face a state space explosion problem in case of complex systems which constrains its application. That's why other model checking approaches appeared.
BDD-based symbolic model checking (e.g., SMV [24]) presents a different way to store the states of the system, grouping them into sets of states represented by predicates on its state variables in the form of BDDs (Binary Decision Diagram). Such an approach reduces the size of the state space to be explored, making for a more efficient exhaustive exploration, yet it limits the nature of the variables that can be manipulated. *Bounded model checking* (e.g., SAT [9], SMT [29]) is another form of symbolic model checking that does not rely on a symbolic representation of the states of the system, but rather on applying decision procedures on prepositional logic. Such an approach turns the verification problem into a satisfiability problem. The goal here is to check if there exists values that can be assigned to the variables in the formula to be verified, so as it evaluates to false, within a certain number of exploration steps. While this approach overcomes the state space explosion problem, it cannot be considered complete since variable assignations under which the evaluation is false could exist beyond the considered search depth.

Complementary Techniques. Symbolic model checking is often seen allied to other techniques in order to improve its efficiency or widen its application range.
Abstractions (e.g., [4]) can be used with symbolic model checking to deal with state space explosion in software analysis. An abstraction can be either sound, in which case properties of the abstract specification are also properties of the original one, or complete, in which case properties of the original specification are properties of the abstract one. While a sound (resp. complete) abstraction guarantees false positive-free (resp. false negative-free) results it cannot guarantee the absence of false negatives (resp. false positives).
Symbolic execution (e.g., [20]) can be placed as the crossover between a formal verification technique and a testing technique for programs. Its underlying idea is to represent input variables using symbols over which the program is symbolically executed instead of assigning concrete values, which yields symbolic formulae instead of concrete results. Hence, one result of the symbolic execution encompasses a set of test cases. In such a context, SMT solvers are often used to check for the reachability of some part of the code, which amounts to checking the satisfiability of the conjunction of the formulae encountered on its corresponding path.

Here we cite studies that make use of model checking-related techniques in their proposed verification approaches (see Table 1).

Oyente [21] was the first attempt at formal smart contract verification. It uses symbolic execution applied at the EVM bytecode of the contract to generate symbolic execution traces among which it looks for certain conditions that translate the presence of one of

Table 1: Smart contracts verification approaches categorized by the used methods

Approaches with Theorem Proving	Approaches based on Model Checking-related techniques			
	Symbolic Execution	Abstraction	SAT/SMT solvers	Model Checking
[8] [1] [18] [3]	[26] **[30]** **[21]** [32] [12]	[31] [10]	[26] **[30]** **[21]** [32] [12] **[19]**	**[22, 23]**

the four vulnerabilities it targets. This proposition actually paved the way for other researchers who wanted to do better in several subsequent studies. Some of them reused it as part of their own tools, like in GASPER [12] which exploits the by-product of Oyente (CFG) in its detection of costly bytecode patterns in terms of gas consumption. Other researchers opted for extending Oyente to detect different/additional bugs (e.g., MAIAN [26], SASC [32] and Osiris [30]). Securify [31] is a security analysis tool for Solidity smart contracts. It starts by decompiling the EVM bytecode into a static-single assignment form and symbolically encoding the corresponding dependence graph in stratified DataLog, leverages the Soufflé solver to derive semantic facts on the contract's data- and control-flow dependencies using declarative inference rules and then checks for the presence of predefined patterns that correspond to the properties the user wants to verify. In fact, the authors use a designated DSL to define compliance (resp. violation) patterns for a number of properties to capture sufficient conditions in a given code to satisfy (resp. violate) such properties. Even though the user can define other patterns to check for additional security properties, it is not possible to define patterns that match a contract-specific property, or arithmetic properties. Besides, in some cases the code does not match any defined pattern and cannot decide on the safety of the contract. Vandal [10] follows the same spirit and adopts a logic-driven program analysis approach. It starts by translating the EVM bytecode into an abstract register transfer language exposing its data- and control-flow structures. This language is then translated into logic relations which are then fed to security analyses written in Soufflé to detect certain vulnerabilities in the contract.

4 SELECTED FORMAL APPROACHES

Among the studies that we have collected on formal verification of Solidity smart contracts, 4 can be categorized as theorem proving-based approaches and 9 make use of model checking-related techniques in their propositions. We note that, despite this second category representing a majority, only one work proposed an approach fully based on model checking in the proper sense of the word (see Table 1). Such propositions are rather walking the line between being formal verification-based and testing-based approaches.

In the following subsections, we present four selected approaches proposed for formal verification of smart contracts, presented in bold in Table 1. We choose to detail the two approaches presented in [21] and [19], the two approaches with the most cited papers, as well as [22, 23] for being the single approach based on model checking. We also single out the approach in [30] as one of the propositions based on the veteran Oyente [21].

4.1 FSolidM and VeriSolid

4.1.1 Approach. In [22] the authors propose an FSM-based approach for the design of secure smart contracts. The premise of their work is that writing smart contracts in a language such as Solidity is error-prone because the smart contract writer may not fully grasp the semantics driving the execution process which often leads to a contract that does not reflect the actual intentions of its creator. They hence aim at closing this semantic gap by developing the FSolidM tool which allows users to design a smart contract as an FSM (Finite State Machine) which is then automatically transformed into a Solidity smart contract. To do so, they propose a definition of a smart contract as an FSM and outline the transformation process that generates the corresponding Solidity contract.

To improve the generated smart contract's security, the authors propose so called "plugins" that prevent some common vulnerability patterns [6]. These plugins actually translate into modifiers appended to the contract's functions to be secured. In Solidity, a modifier is used to change the behavior of the functions with which it is associated. In this context, modifiers are used to implement security patterns into the generated Solidity functions, e.g., by adding preconditions to check prior to their execution.

The work presented in [22] was in fact laying ground for the next paper [23] in which the authors present VeriSolid, the improved version of FSolidM. In fact, [23] extends [22] in that it adds formal operational semantics to the formerly proposed FSolidM model and therefore extends the Solidity code generator. This upgrade introduces the aspect of formal verification into the tool, which provides the user with the ability to specify intended behavior in the form of liveness, deadlock freedom and safety properties. It offers customizable templates to express and check some CTL properties by a backend symbolic model checker.

4.1.2 Tool. The FSolidM tool offers four plugins to deal with four vulnerabilities: (1) a locking plugin against the reentrancy attack, (2) a transition counter plugin to enforce transition ordering and avoid falling into unpredictable states, (3) an automatic timed transitions plugin to implement time-constraint patterns and (4) an access control plugin to manage authorization for functions execution.

While VeriSolid extends FSolidM, it does not take into account the same vulnerabilities as the latter since it offers the possibility to express CTL properties through templates such as "transition_b will eventually happen after transition_a" which can be used to check for a denial-of-service vulnerability. Additionally, the authors chose to deal with the reentrancy vulnerability intrinsically by introducing an In-Transition state into which the system goes at the beginning of each transition, thus prohibiting any overlapping calls.

In order to incorporate the formal verification aspect, the authors resort to using the NuXmv symbolic model checker [11] which features SMT-based techniques for the verification of infinite state systems. For that, they opt for augmenting the initial FSM model to take in the semantics of the Solidity functions' statements, transforming the resulting augmented model into a BIP (Behavior-Interaction-Priority) transition system [7] (which is guaranteed to be deadlock-free), using an existing BIP2NuSmv transformation

tool and feeding the result to the NuXmv model checker along with the CTL formulae following the provided templates.

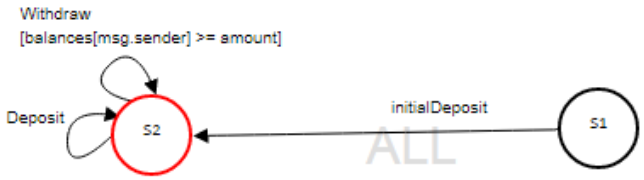

Figure 2: Screenshot of the VeriSolid tool - 1

For lack of an underlying business logic in our example in Listing 1, we choose to test the tool on an excerpt relative to reentrancy. We note that in this model we distinguish the first call to deposit (InitialDeposit) from the rest to get around a modeling restriction that requires the system to contain a minimum of 2 states.

```
contract VulnContract {
  uint private creationTime = now;
  enum States {InTransition, S1, S2}
  States private state = States.S1;
  mapping (address=>uint) balances;
  function Deposit () public {
    require(state == States.S2);
    state = States.InTransition;
    balances[msg.sender] += msg.value;
    state = States.S2;}
  function InitialDeposit () public {
    require(state == States.S1);
    state = States.InTransition;
    balances[msg.sender] += msg.value;
    state = States.S2;}
  function Withdraw (uint amount) public {
    require(state == States.S2);
    require(balances[msg.sender] >= amount);
    state = States.InTransition;
    msg.sender.call.value(amount);
    balances[msg.sender] -= amount;
    state = States.S2;}}
```

Listing 3: Solidity code generated by VeriSolid

Figure 2 shows the model we proposed and Listing 3 the generated Solidity code. We tested this code and confirmed its insusceptibility to reentrancy. Figure 3 shows the representation using VeriSolid of the property stating that the function Withdraw can only be called after the InitialDeposit function had been called, along with its verification result.

4.1.3 Discussion. In the attempt to integrate the formal verification aspect into the approach, the first premise of closing the semantic gap of Solidity got disregarded since the statements constituting the functions' bodies need to be provided by the user in Solidity. On the practical side, it may feel counter-intuitive and even restrictive for the user to have to think about the smart contracts they want to write in terms of states at design time, only to find themselves writing the code themselves nonetheless. Moreover, despite the help that may come with the proposed templates for CTL properties for some users, they might as well be seen as an unnecessary restriction to some other more experienced users who would like to verify more complicated properties that cannot be expressed within the limits of the provided templates. As much as some guidance is appreciated, it should not turn into a barrier to expressivity. Last but not least, it is important to mention that the models in both FSolidM and VeriSolid do not take into account any variables. Therefore, no properties on the evolution of the values of the variables during the execution of the smart contract can be verified, which also cuts back on the range of properties the user can check.

4.2 ZEUS

4.2.1 Approach. Zeus [19] is a framework based on symbolic model checking for the verification of smart contracts. It takes as input a smart contract written in a high-level language along with a so-called policy that contains the criteria to be checked and which the user needs to specify in an XACML-styled template. The input smart contract code is then instrumented with assertion instructions according to its corresponding policy by means of static analysis and is passed on to a translator that the authors had devised to convert it into a low-level intermediate representation (LLVM bitcode) which is then fed to an existing verification engine in order to assert the safety of the smart contract. This is based on the primary description of the approach. As more details are later presented in the paper, we realize that this is not actually the exact right ordering of steps since the static analysis is afterwards said to be performed on top of the intermediate representation rather than the high-level code and the same goes for the added assertions. Later on, we also realize that the high-to-low level transformation is not straightforward. The authors propose an abstract language into which Solidity code is transformed before undergoing the first transformation into LLVM bitcode.

As to the considered properties, the authors distinguish two main families of vulnerable smart contracts: incorrect and unfair smart contracts. They define correctness as the adherence to safe programming practices and fairness as the adherence to agreed upon higher-level business logic.
An *incorrect* smart contract can have one of the following vulnerabilities: (1) reentrancy, (2) unchecked sends, (3) failed sends, (4) integer overflow/underflow or (5) transaction state dependence. An *unfair* smart contract can have one of the following vulnerabilities: (1) absence of logic, (2) incorrect logic or be (3) logically correct but unfair.
Besides vulnerabilities that fall under these two categories, two more vulnerabilities which can actually be caused by the miner's influence are introduced: (1) block state dependence (BSD) and (2) transaction order dependence (TOD).

4.2.2 Tool. Zeus was not made available online, but the authors state that they have implemented a prototype in C++. The tool's main components are the policy builder and the Solidity-to-LLVM bitcode translator. For the former, they leverage the AST output produced by the *Solc* compiler and taint analysis on the source code to extract the information needed to assist the user in forming the conditions to verify. As for the translator, it takes as input the smart contract and uses existing LLVM APIs to generate the bitcode, which will then be instrumented by adding assertions according to the built policy. As a backend verifier, they opt for Seahorn [17].

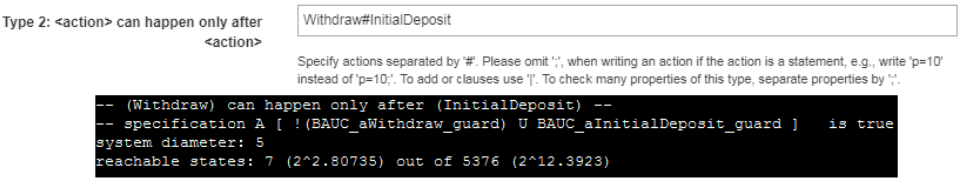

Figure 3: Screenshot of the VeriSolid tool - 2

4.2.3 Discussion. Proposing an abstract interpretation language for Solidity to go through before obtaining an LLVM bitcode contributes in improving the scalability of Zeus. In fact, using over-approximations and reducing functions into summaries and loops into data domains results in a reduced state space for the symbolic model checker to be used later. However, a formal reasoning still needs to be established to prove the actual semantic equivalence between the two languages (Solidity and the proposed abstract language). Furthermore, the authors mention that using the abstract language allows the support of multiple blockchain platforms, yet we think that using this bridge language constrains the high-level languages the tool can support. To integrate a language other than Solidity, new correspondences with the proposed abstract language would have to be defined (if not the whole language), the translation into LLVM would have to be revised and the automation of the assertions insertions would have to be reimplemented.

Leveraging the use of the LLVM bitcode extends the reach of Zeus in the sense that it can make use of any backend symbolic model checker supporting that standard. Seahorn [17] is the first choice of the authors but not the only one. It was chosen for its ability to generate verification conditions using CHCs (Constrained Horn Clauses) over LLVM bitcode. Other symbolic model checkers can be used, such as SMACK [27], but that may require some modifications on the LLVM bitcode as for example some model checkers might use different lengths for the same type which needs to be taken into account when switching the verifier. The authors state that CHCs are suitable for the representation of verification conditions, but do not elaborate. We think that the tool may be able to verify a wider range of properties if it were to support the representation of properties using other logics besides CHC.

We also note that Zeus only accounts for parameters that can be computed at the source code level and hence cannot verify properties relating to parameters as gas consumption.

4.3 OYENTE

4.3.1 Approach. Besides the smart contracts analysis tool they call Oyente, the authors of [21] also propose refinements/recommendations to Ethereum's protocol in the form of improvements to its operational semantics in order to fix certain security problems. In this survey, we are only interested in the Oyente tool that they propose as a "pre-deployment mitigation". It is based on symbolic execution and functions over the bytecode which needs to be provided as input along with Ethereum's global state. The latter would serve as an initialization for the contract's variables. Message call-related variables are however treated as input symbolic values. The general idea behind Oyente is to symbolically explore a control flow graph corresponding to the bytecode by symbolically executing instructions within states of that graph and use a symbolic constraint solver to decide on the feasibility of branching conditions. The possible presence of vulnerabilities is detected by checking for specific conditions in the generated symbolic traces.

This tool targets four bugs: (1) TOD, (2) timestamp dependence, (3) mishandled exceptions and (4) reentrancy.

4.3.2 Tool. Oyente [21] is implemented in Python, uses Z3 [13] as a backend SMT solver and detects the 4 discussed problems (see 4.3.1). Its design has four main components: *(1) CFGBuilder:* it outputs a Control Flow Graph of the bytecode. This graph is only partly constructed statically as some edges are later added after symbolic execution. *(2) Explorer:* this is mainly an interpreter loop that symbolically executes one instruction at one state at a time, starting from the entry node of the CFG generated by the previous component. The Explorer actually simulates the behaviour of EVM instructions and makes use of Z3 to decide on path conditions. The loop ends when no more unexplored states exist or when a timeout is reached. The CFG is potentially enriched by the end of this phase and a set of symbolic traces is outputted. *(3) CoreAnalysis:* it in turn comprises 4 components to detect the 4 previously introduced bugs. These components work by checking specific conditions when analyzing the symbolic traces resulting from the Explorer in order to flag the possible presence of the corresponding bugs. *(4) Validator:* this step is added to further reduce the rate of false positives. The user, however, still needs to intervene to confirm that the flagged bugs are a real threat.

The tool has been in active development up until May 2018, and some unreported features were added to the updated version. Mainly, in its latest version, Oyente can supposedly detect the following issues (in addition to the previously mentioned issues): (1) integer overflow/underflow, (2) Parity Multisig bug 2 and (3) callstack depth attack.

Tested on our example in Listing 1, the paper's version of Oyente was able to detect the timestamp dependence and the reentrancy, as shown in the truncated results in Figure 4.

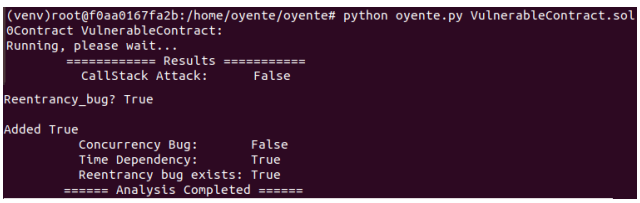

Figure 4: Screenshot of the Oyente tool

4.3.3 Discussion. Oyente can be seen as the first attempt at formal smart contract verification, which paved the way for researchers in several subsequent propositions. Despite its ability to detect important vulnerabilities in smart contracts, Oyente is not a complete verifier. Its major drawback is that its reported errors may be spurious. In other words, its results may contain false positives. One example for that is flagging a false reentrancy vulnerability in a code that uses a send function, which should not pose a threat unless its default gas were altered. This can actually be explained by the fact that Oyente relies on the bytecode of the smart contract, in which both functions send and call are mapped to the same CALL bytecode, which translates into contextual information loss. To detect reentrancy, the tool checks the path condition before each CALL it comes across and checks if it still holds after the bytecode's execution, in which case it is registered as a vulnerability.

4.4 OSIRIS

4.4.1 Approach. This work [30] specifically targets integer vulnerabilities in Solidity smart contracts. More precisely, the authors investigate the presence of 3 types of bugs in such contracts: (1) arithmetic bugs like integer underflows/overflows and bugs caused by divisions where the denominator is zero, (2) truncation bugs which can happen when converting a value into a new type with a shorter range than that of its initial type and (3) signedness bugs that can occur when converting a signed integer typed value into an unsigned integer type (or the opposite).

The proposed approach works on integer bugs detection at the bytecode level and is based on two techniques, namely symbolic execution and taint analysis. It comprises 3 phases:

Integer type inference: even though Solidity is a statically typed language, typing information is supposed to get lost at the bytecode level. The compiler, however, leaves behind discrete trails (e.g., AND bitmask, SIGNEXTEND opcode, etc) that the authors track down to deduce the size and sign of integers in the bytecode.

Integer bugs detection: a different detection technique is proposed for each of the targeted integer bug types: *(1) arithmetic bugs*: a constraint is emitted to the backend solver for each arithmetic instruction. This constraint is formed so that it is only satisfied if a set of predefined in-bounds requirements specific to the instruction in question are not totally met. Consequently, a bug is detected if one of the emitted constraints under some path conditions is found to be satisfiable by the solver. *(2) truncation bugs*: such bugs are detected by tracking the instructions used by Solidity to perform truncation (i.e., AND and SIGNEXTEND for signed and unsigned integers). A constraint is formed for such instructions as to be satisfied if the input value is larger than the output. Consequently, a truncation bug is detected if one of the emitted constraints under some path conditions is found to be satisfiable by the solver, all while ignoring two specific patterns for intentional truncation corresponding to truncation due to a conversion to type address and truncation as a technique to fit more than one variable into the same storage slot. *(3) signedness bugs*: for this type of bugs, the authors reuse an approach that was initially proposed for Linux programs [25] and adapt it for Solidity smart contract. The gist of the applied method is to infer information on signed and unsigned types on the values from the executed EVM instructions and spot the symbolic variables that can be assigned both types.

False positives reduction: the authors actually consider this as two separate steps, since they use two different techniques to reduce the rate of generated false alarms. The first step is to apply taint analysis in order to check only instructions whose input data is tainted (can be manipulated by an attacker) and further validate only the ones that touch sensitive locations (can be harmful in that they may alter the execution path, storage and ether flow). The second step of false positives reduction is recognizing detected integer bugs which originate from unharmful code such as an intentional check (if condition) meant to catch an overflow bug.

4.4.2 Tool. The implemented tool called Osiris is written in Python. It operates over the bytecode but can accept Solidity code as input which it internally compiles into bytecode. It consists of 3 main components: (1) *symbolic analysis* is basically a reuse of the previously presented Oyente tool, used to generate the bytecode's CFG and symbolically execute its instructions, (2) *taint analysis* checks, for each executed instruction, whether it pertains to a specific set of instructions defined by the authors as susceptible of being used by an attacker, in which case the locations it affects (in the stack, memory and storage) are tagged and the propagation is carried out according to the EVM semantics. It then checks if this instruction can be impactful on sensible locations and (3) *integer error detection* is called upon the instructions detected by the taint analysis, implements the errors detection methods discussed above and uses Z3 to check for the feasibility of the created constraints.

Figure 5 shows the result for running Osiris on our example in Listing 1. The tool detected an overflow bug as well as a truncation bug and located them in the code.

4.4.3 Discussion. This work focuses on a restrictive range of vulnerabilities, covering specific integer-related bugs. On the one hand, the approach shows better results than other existing approaches dealing in part with such vulnerabilities, yet its range of application is thereby restricted. Additionally, Osiris points out the origin of the detected vulnerability in the analysed code but does not provide an example of an execution that may lead to an error, which would make it easier for the contract writer to revise the code.

5 COMPARISON AND DISCUSSION

The majority of the discussed approaches are based on the analysis of the EVM bytecode instead of the higher-level Solidity source code which can be explained by Solidity's lack of formal semantics. On the other hand, relying on the bytecode has its own impediment since it leads to the loss of contextual information, hence limiting the range of properties that can be verified on the contract.

We notice that most of the proposed approaches, led by the first proposition [21], use symbolic execution to generate the traces that would be used for the verification. Such approaches usually use under-approximation which means that critical violations can be overlooked.

A survey on the vulnerabilities in smart contracts [14] reports 49 bugs that can occur in a smart contract, 29 of which were categorized using the Bugs Framework of NIST into 10 bug classes. As shown in Table 2, the proposed verification methods only target a

Figure 5: Screenshot of the Osiris tool

Table 2: Vulnerabilities supported by the proposed smart contract verification approaches

Tool	Can express properties using	Reported detected vulnerabilities							
		Limited stack	Arithmetic bugs	TSD	TOD	Reentrancy	Self destruction	Gas run-out	Other
[23]	templates for CTL expressions	-	-	-	-	+	-	-	+
[19]	XACML templates for CHC	-	±	+	+	+	-	-	+
[21]	x	+	±	+	+	+	+	-	+
[30]	x	-	+	-	-	-	-	-	-
[31]	compliance/violation patterns in a DSL	-	-	-	+	+	-	-	+
[10]	security analysis as a logic specification in Soufflé	-	-	-	-	+	+	-	+
[26]	x	-	-	-	-	-	+	-	+
[12]	x	-	-	-	-	-	-	-	+
[32]	x	+	±	+	+	+	-	-	+

limited number of these bugs, with a maximum of 18 claimed by the commercialized version of Securify [31]. We also note that 4 approaches give the user the ability to express customized properties to check. None of them, however, supports contract-specific properties. Moreover, we underline that only single function reentrancy is considered in all of the existing approaches. Furthermore, none of the proposed approaches deals with the verification of interacting contracts. This means that the verification of smart contracts is a field that, despite having been investigated at an early stage, still needs to be further studied to achieve correctness in smart contracts and consolidate the desired trust in the Blockchain environment.

In the following we report tools comparisons included in their corresponding papers.

Zeus vs Oyente in [19] The evaluation of Zeus was done on a dataset of 1524 smart contracts and its results were compared to Oyente's for the commonly treated vulnerabilities (reentrancy, unchecked send, BSD and TOD). 54 contracts were reported by Zeus to be vulnerable to reentrancy against 265 by Oyente. The undetected bugs by Zeus were said to be false positives caused by Oyente considering reentrancy possible with send calls. This is not totally true, as using send can still be susceptible to reentrancy if the allocated amount of gas were to be manually increased. For the unchecked send vulnerability, Zeus was reported to detect 324 bugs with 3 false positives, against 112 bugs by Oyente with 89 false positives. The results for BSD show more detected bugs by Zeus than Oyente, which is only logical since the former considers multiple block variables while the latter only considers the block's timestamp. Zeus is also reported to detect more TOD bugs (607) than Oyente (126) with a lower false alarm rate.

Osiris vs Zeus in [30] The authors of [30] evaluated their tool using a subset of the dataset of smart contracts previously used by Zeus (they retrieved 883 out of 1524 contracts), and compared it to the latter for their commonly detectable bugs. Their reported results show a big difference in the number of detected integer overflow/underflow bugs with Zeus detecting 628/883 and Osiris detecting 172/883. They claim that this difference can be explained

by Zeus prioritizing completeness over the real exploitability of its reported bugs. They also bring into question Zeus's soundness by manually investigating 5 contracts that were reported as containing bugs by Osiris but not by Zeus and confirming their unsafety.

SASC vs Oyente in [32] The authors in [32] report more detected timestamp dependence bugs using their tool (866 by SASC against 292 by Oyente out of 2952 contracts). In fact, this can be explained by the different ways both tools use to target such a vulnerability. While Oyente detects the use of timestamp whenever it is related to ether transfer only, SASC tarets its use in other operations as well. We also note that SASC is able to locate the bugs in the corresponding code, unlike Oyente that only signals their presence.

Securify vs Oyente in [31] Results were compared to Oyente in [31] for the detection of reentrancy, TOD and mishandled exceptions. The authors report better results overall for Securify. Reentrancy was detected in the same number of contracts by both tools, with presence of false positives with Oyente but not with Securify. As for the other two vulnerabilities, Securify was reported to detect more valid occurrences than Oyente and no false negatives at all, albeit with a slightly higher number of false positives.

6 CONCLUSION

As the monetary value circulating on Ethereum keeps rising, and since smart contracts are responsible for the management of Ether across the Blockchain, providing means to rigorously guarantee the security of smart contracts becomes an inescapable requisite.

Our overview of contributions towards the formal verification of Solidity smart contracts shows that while a big step has been taken, the extra mile has yet to be walked. In fact, the proposed approaches target but a limited number of vulnerabilities (e.g., in comparison with the number of reported bugs in [14]). More importantly, as the reach of smart contracts gains more ground touching more application fields, the need for the verification of domain-specific properties grows more urgent. We think that developing a verification approach that relies on defining such properties in appropriate logic languages might just bring an answer to such an exigency.

REFERENCES

[1] [n.d.]. Formal Verification for Solidity Contracts — Ethereum Community Forum. https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts.

[2] [n.d.]. Ropsten. https://ropsten.etherscan.io/.

[3] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. New York, NY, USA, 66–77.

[4] Saswat Anand, Corina S. Pasareanu, and Willem Visser. [n.d.]. Symbolic execution with abstraction. *Int. J. Softw. Tools Technol. Transf.* 11, 1 ([n. d.]).

[5] Monika Di Angelo and Gernot Salzer. 2019. A Survey of Tools for Analyzing Ethereum Smart Contracts. *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)* (2019), 69–78.

[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. [n.d.]. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust - 6th International Conference, POST 2017, Uppsala, Sweden, April 22-29, Proceedings*.

[7] Ananda Basu, Saddek Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. 2011. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software* 28, 3 (2011), 41–48.

[8] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*. 91–96.

[9] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*. 193–207.

[10] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. *CoRR* abs/1809.03981 (2018).

[11] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. [n.d.]. The nuXmv Symbolic Model Checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of VSL 2014, Austria*.

[12] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 442–446.

[13] Leonardo Mendonça de Moura and Nikolaj Bjørner. [n.d.]. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Budapest, Hungary, March 29-April 6, 2008*.

[14] Wesley Dingman, Aviel Cohen, Nick Ferrara, Adam Lynch, Patrick Jasinski, Paul E. Black, and Lin Deng. 2019. Defects and Vulnerabilities in Smart Contracts, a Classification using the NIST Bugs Framework. *IJNDC* 7, 3 (2019), 121–132.

[15] Nick Dodson. 2016. Solint: A linting utility for Ethereum solidity smart-contracts. https://github.com/SilentCicero/solint.

[16] Vimal Dwivedi, Vipin Deval, Abhishek Dixit, and Alex Norta. 2019. Formal-Verification of Smart-Contract Languages: A Survey. In *Advances in Computing and Data Sciences*. Singapore, 738–747.

[17] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. 343–361.

[18] Yoichi Hirai. 2017. Ethereum VM for Coq (v0.0.2). https://medium.com/@pirapira/ethereum-virtual-machine-for-coq-v0-0-2-d2568e068b18.

[19] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*.

[20] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*. 553–568.

[21] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Austria, October 24-28*.

[22] Anastasia Mavridou and Aron Laszka. [n.d.]. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security - 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 - March 2, 2018*.

[23] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. [n.d.]. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security - 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019*.

[24] Kenneth L. McMillan. 1993. *Symbolic model checking*. Kluwer.

[25] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*. 67–82.

[26] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. 653–663.

[27] Zvonimir Rakamaric and Michael Emmi. [n.d.]. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014*.

[28] Alex Rea. 2016-2020. SolCover: Code coverage for Solidity smart-contracts. https://github.com/sc-forks/solidity-coverage.

[29] Cesare Tinelli. 2012. SMT-Based Model Checking. In *NASA Formal Methods - 4th International Symposium, NFM 2012, USA, April 3-5, 2012. Proceedings*.

[30] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for Integer Bugs in Ethereum Smart Contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, USA, December 03-07*.

[31] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Canada, October 15-19*.

[32] Ence Zhou, Song Hua, Bingfeng Pi, Jun Sun, Yashihide Nomura, Kazuhiro Yamashita, and Hidetoshi Kurihara. 2018. Security Assurance for Smart Contract. In *9th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2018, Paris, France, February 26-28, 2018*. 1–5.