Automated Inference on Financial Security of Ethereum Smart Contracts

Wansen Wang and Wenchao Huang, *University of Science and Technology of China;* Zhaoyi Meng, *Anhui University;* Yan Xiong and Fuyou Miao, *University of Science and Technology of China;* Xianjin Fang, *Anhui University of Science and Technology;* Caichang Tu and Renjie Ji, *University of Science and Technology of China*

https://www.usenix.org/conference/usenixsecurity23/presentation/wang-wansen

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Automated Inference on Financial Security of Ethereum Smart Contracts

Wansen Wang[1], Wenchao Huang[1]*, Zhaoyi Meng[2]*, Yan Xiong[1],
Fuyou Miao[1], Xianjin Fang[3], Caichang Tu[1], Renjie Ji[1]
[1]*University of Science and Technology of China,* [2]*Anhui University,*
[3]*Anhui University of Science and Technology*

Abstract

Nowadays millions of Ethereum smart contracts are created per year and become attractive targets for financially motivated attackers. However, existing analyzers are not sufficient to analyze the financial security of a large number of contracts precisely. In this paper, we propose and implement FASVERIF, an automated inference system for fine-grained analysis of smart contracts. FASVERIF automatically generates models to be verified against security properties of smart contracts. Besides, different from existing approaches of formal verifications, our inference system also automatically generates the security properties. Specifically, we propose two types of security properties, invariant properties and equivalence properties, which can be used to detect various types of finance-related vulnerabilities and can be automatically generated based on our statistical analysis. As a result, FASVERIF can automatically process source code of smart contracts, and uses formal methods whenever possible to simultaneously maximize its accuracy. We also prove the soundness of verifying our properties using our translated model based on a custom semantics of Solidity.

We evaluate FASVERIF on a vulnerabilities dataset of 549 contracts by comparing it with other automatic tools. Our evaluation shows that FASVERIF greatly outperforms the representative tools using different technologies, with respect to accuracy and coverage of types of vulnerabilities. We also evaluate FASVERIF on a real-world dataset of 1700 contracts, and find 13 contracts with bugs that can still be leveraged by adversaries online.

1 Introduction

Smart contracts on Ethereum have been applied in many fields such as financial industry [3], and manage assets worth millions of dollars [57], while the market cap of the Ethereum cryptocurrency, *i.e.*, ethers, grows up to $177 billions on July 27, 2022 [14]. Unfortunately, this makes smart contracts become attractive targets for attackers. The infamous vulnerability in the DAO contract led to losses of $150M in June 2016 [1]. In July 2017, $30M worth of ethers were stolen from Parity wallet due to a wrong function [4]. Most recently, there were $27M worth of ethers stolen from the Poly Network contract in August 2021 [13]. It is therefore necessary to guarantee the financial security of smart contracts, *i.e.*, the ethers and tokens of contracts are not lost in unexpected ways.

Nevertheless, existing analyzers are not sufficient to analyze the financial security of numerous contracts accurately. Current security analyzers for smart contracts can be divided into the following three categories: automated bug-finding tools, semi-automated verification frameworks, and automated verifiers. The bug-finding tools [44] [35] [38] support automated analysis on a great amount of smart contracts, motivated by the fact that 10.7 million contracts are created in 2020 [41]. However, the analysis is based on pre-defined patterns and is not accurate enough [54]. The verification frameworks target to formally verify the correctness or security of smart contracts, with the requirement of manually defined properties [54] or user assistance in verification [48] [29]. It is therefore difficult for these analyzers to analyze a large number of contracts. The automated verifiers try to provide sound and automated verification of pre-defined properties for smart contracts. To the best of our knowledge, there are three automated verifiers eThor [52], SECURIFY [60] and ZEUS [40]. However, eThor does not aim for the financial security of smart contracts, and only detects reentrancy vulnerabilities [9] and checks assertions automatically. SECURIFY does not support solving numerical constraints and cannot detect numerical vulnerabilities, *e.g.*, overflow. ZEUS has soundness issues [52] in transforming contracts into IR and thus cannot analyze smart contracts accurately.

We propose and implement FASVERIF, a system of automated inference [51] [34], *i.e.*, a static reasoning mechanism where the properties are expected to be automatically derived, for achieving full automation on fine-grained financial security analysis of Ethereum smart contracts. Firstly,

*Corresponding Authors

FASVERIF automatically generates two kinds of finance-related security properties along with the corresponding models for verification. Secondly, FASVERIF can verify these finance-related security properties automatically. Overall, the goal of FASVERIF is to analyze the financial security of numerous contracts accurately, whereby the security properties are generated automatically based on our statistical analysis, the soundness of modeling is proven and the verification is implemented using the formal tools Tamarin prover [46] and Z3 [26]. Moreover, FASVERIF generates properties based on the financial losses caused by vulnerabilities instead of known vulnerability patterns, thus covering various vulnerabilities and suitable for the analysis of financial security.

We collect a vulnerabilities dataset consisting of 549 contracts from other works [38] [53] [42] [36], and evaluate FASVERIF on it with other automatic tools. Our evaluation shows that FASVERIF greatly outperforms the representative tools using different technologies, in which it achieves higher accuracy and F1 values in detection of various types of vulnerabilities. We also evaluate FASVERIF on 1700 contracts randomly selected from a real-world dataset. FASVERIF finds 13 contracts deployed on Ethereum with exploitable bugs, including 10 contracts with vulnerabilities of transferMint [7] that can evade the detection of current automatic tools to the best of our knowledge.

In summary, this paper makes the following contributions:

1) We propose a novel framework for achieving automated inference, where finance-related security properties and corresponding models are generated from the source code of a smart contract and used for automated verification.

2) We propose a method for property generation based on a statistical analysis of 30577 smart contracts. We design two types of properties, financial invariant properties and transactional equivalence properties, which correspond to various finance-related vulnerabilities such as transferMint [7], and we abbreviate them as invariant properties and equivalence properties, respectively.

3) We propose modeling methods for our invariant properties and equivalence properties and prove the soundness of verifying these two types of properties using our translated model based on a custom semantics of Solidity [39].

4) We implement FASVERIF for supporting property generation, modeling and verification, where we embed Z3 into Tamarin prover, the state-of-the-art tool for verifying security protocols, to use trace properties of reachability and numerical constraint solving for verifying finance-related properties.

5) We evaluate the effectiveness of FASVERIF and find 13 contracts with exploitable vulnerabilities using FASVERIF.

2 Preliminaries

2.1 Smart contracts on Ethereum

Ethereum is a blockchain platform that supports two types of accounts: contract accounts, and external accounts. Each account has an ether balance and a unique address. A contract account is associated with a piece of code called a smart contract, which controls the behaviors of the account, and a storage that stores global variables denoting the state of the account. External accounts are controlled by humans without associated code or global variables.

Functions in the smart contracts can be invoked by transactions sent by external accounts. A transaction is packed into a block by the miner and when that block is published into the blockchain, the function invoked by the transaction is executed. Functions can also be invoked by internal transactions sent by contract accounts and the sending of an internal transaction can only be triggered by another transaction or internal transaction.

2.2 Solidity programming language

$$
\begin{aligned}
\texttt{Contract C} &:= \texttt{contract}\ n_c\ \{\texttt{d}, \texttt{func}^+\} \\
\texttt{Func func} &:= \texttt{function}\ f(\texttt{d})\{\texttt{stmt}\} \\
\texttt{Stmt stmt} &:= \texttt{stmt}_A \mid \texttt{if}\ \texttt{e}_b\ \texttt{then stmt else stmt} \mid \texttt{stmt;stmt} \\
\texttt{Atom Stmt stmt}_A &:= v \leftarrow \texttt{e} \mid \tau\, v \leftarrow \texttt{e} \mid \texttt{require}\ \texttt{e}_b \mid \texttt{e}_c \mid \texttt{return} \\
\texttt{Bool Expr e}_b &:= \texttt{e} == \texttt{e} \mid \texttt{e} < \texttt{e} \mid \texttt{e} > \texttt{e} \mid \texttt{e} != \texttt{e} \\
\texttt{Call Expr e}_c &:= v.\texttt{transfer}(v) \mid v.\texttt{send}(v) \mid \\
&\quad\ v.\texttt{call}().\texttt{value}(v) \mid n_c(v).f(\texttt{p}) \\
\texttt{Expr e} &:= v \mid v + \texttt{e} \mid v - \texttt{e} \mid v * \texttt{e} \mid v/\texttt{e} \mid v\%\texttt{e} \mid \\
&\quad\ v ** \texttt{e} \\
\texttt{Type}\ \tau &:= \tau_B \mid \tau_B \mapsto \tau \mid n_c \\
\texttt{BasicType}\ \tau_B &:= \texttt{uint} \mid \texttt{bool} \mid \texttt{address}
\end{aligned}
$$

Figure 1: Core subset of Solidity

```
1  contract Ex1{
2      mapping(address=>uint) balances;
3      constructor() public{
4          balances[0x12] = 100;
5      }
6      function transfer(address to,uint value) public{
7          uint val1 = balances[msg.sender] - value;
8          uint val2 = balances[to] + value;
9          balances[msg.sender] = val1;
10         balances[to] = val2;
11         return;
12     }
13 }
```

Figure 2: Example contract Ex1.

The most popular programming language for Ethereum smart contracts is Solidity [24]. We take the smart contracts written in Solidity as the object of study in this paper. For brevity, we focus on a core subset of Solidity as shown in Fig. 1. Taking the contract Ex1 in Fig. 2 for example, a contract consists of declarations of global variables (Line 2) and functions (Line 3 to 12). Here, `constructor` is a special function used to initialize global variables. The function bodies

consist of atom statements $\texttt{stmt}_\texttt{A}$ and conditional statements. Taking the function `transfer` as an example, $\texttt{stmt}_\texttt{A}$ can be a declaration statement on Line 7, an assignment statement on line 9, or a return statement on line 11, etc. Specially, there is a kind of atom statements $\texttt{e}_\texttt{c}$ which are used to invoke official functions of Solidity or custom functions of contracts. The variables used in contracts fall into the following types: 1) basic types τ_B. 2) $\tau_B \mapsto \tau$ denoting a mapping from variables of type τ_B to variables of type τ, *e.g.*, `balances` in Fig. 2. 3) n_c denoting a contract. Additionally, there are some special built-in variables of Solidity that cannot be assigned: 1) `block.timestamp` denoting the timestamp of the block that contains the current transaction. 2) c`.balance` expressing the ether balance of contract in address c. 3) `msg.sender` denoting the address of the sender of the current transaction. Note that the functions of different visibilities are handled in similar ways, so we only introduce how to process public functions in this paper for brevity while FASVERIF supports analysis of all kinds of them.

Currently, there is no official formal semantics of Solidity to the best of our knowledge. Instead, we design FASVERIF and prove the soundness of our translation based on a custom semantics of Solidity, named KSolidity [39]. KSolidity is defined using K-framework [50], and the definition of KSolidity consists of 3 parts: Solidity syntax, the runtime configuration, and a set of rules constructed based on the syntax and the configuration. Configurations form of cells that store information related to the executions of contracts, *e.g.*, the variables of contracts. The rules specify the transitions of configurations.

2.3 Multiset rewriting system

FASVERIF leverages the multiset rewriting system in Tamarin prover [46] to model smart contracts and attackers. Each state of a multiset rewriting system is a multiset of facts, denoted as $F(t_1,\ldots,t_n)$, where F is a fact symbol, and $t_1,\ldots,t_n$ are terms. The transitions of states are defined by labeled rewriting rules. A labeled rewriting rule is denoted as $l-[a]\rightarrow r$, where l, a and r are three parts called premise, action, and conclusion, respectively. The rule is applicable to state s, if a ground instance $l\sigma$ (where σ is a substitution [45]) to be a subset of s. To obtain the successor state s', the ground instance $l\sigma$ is removed and $r\sigma$ is added. The action a is also a multiset of facts representing the label of the rule. Meanwhile, global restrictions on facts in a can be made such that the execution of the protocol can be further restrained.

3 OVERVIEW

3.1 Design of FASVERIF

As shown in Fig. 3, FASVERIF contains 4 modules:

Independent modeling: given the source code of a smart contract as input, the module generates a partial model of the

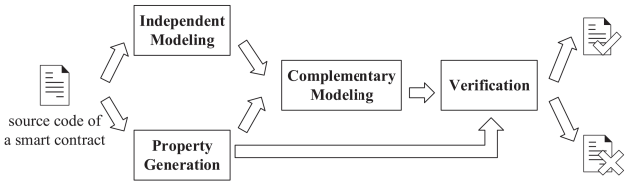

Figure 3: Design of FASVERIF.

contract, which gives the initial state of the running contract and general rules for state transitions. It translates the contract, as well as the possible behaviors of adversaries, into the model, which is independent of specific security properties. Note that this partial model cannot be verified directly.

Property generation: FASVERIF then generates a set of security properties that the smart contract should satisfy.

Complementary modeling: the module outputs additional rules for each property to complement the partial model, and tries to reduce the size of the model for different properties.

Verification: we finally design the method of verification to determine whether the properties are valid. We also modify the code of Tamarin prover for supporting the verification where numerical constraint solving is additionally required.

3.2 Adversary model

We assume that the adversaries can launch attacks by leveraging the abilities of three types of entities: external accounts, contract accounts and miners. The concerned attacks on a smart contract are processes that affect the variables related to the smart contract and thus the results of the smart contract executions. The variables that can be changed by the adversary fall into two categories: some global variables of contracts and `block.timestamp`. An external account or a contract account needs to invoke functions in victim contracts to change the values of their global variables, while a miner can manipulate `block.timestamp` in a range [2] [22]. In summary, we assume that the adversary can perform the following operations: **C1**. Sending a transaction to invoke any function in victim contracts with any parameters. **C2**. Implementing a fallback function to send an internal call message. This message can invoke any function in victim contracts with any parameters. **C3**. Increasing the timestamp of a block by up to 15 seconds [2] [22]. Besides, the changes in exchange rates between tokens and ethers are not considered in FASVERIF.

4 INDEPENDENT MODELING

Given a smart contract, the module of independent modeling automatically outputs general rules for modeling the executions of the contract and the behaviors of external accounts and the adversaries. The rules in the multiset rewriting system correspond to the sequences of transitions of the configurations of KSolidity. Therefore, we firstly define the terms used in the rules, and sequences using the terms. Then, we show the

processes of modeling the behaviors using the terms. Finally, a comprehensive example is given to illustrate the usage of the rules, and discussions are made on technical challenges of property generation and complementary modeling based on the independent modeling.

4.1 Terms and sequences

The terms in multiset rewriting system are translated from the names in Solidity language. There are two types of terms: constant terms and variable terms. Correspondingly, as shown in Fig. 1, a name v in Solidity may represent a contract, a function, a variable, or a constant. Therefore, given a name v, we compute a tuple $\langle name, type, range, ether\rangle$. Here, *name* is a term used in multiset rewriting system, which corresponds to v. Term $type \in \{\mathsf{T_v}, \mathsf{T_c}\}$. If v is a variable, $type = \mathsf{T_v}$; otherwise, $type = \mathsf{T_c}$. Term $range \in \{\mathsf{R_g}, \mathsf{R_l}, \mathsf{R_o}\}$. If v is a global variable and a local variable, *i.e.*, a variable defined inside a function, $range = \mathsf{R_g}$ and $\mathsf{R_l}$, respectively; otherwise, *e.g.*, v is a constant, $range = \mathsf{R_o}$. If v is a variable representing the ether balance of an account, $ether = \mathsf{E_y}$; otherwise $ether = \mathsf{E_n}$. Note that we consider the variables denoting ether balances as global. Since value of v is unchanged if $type = \mathsf{T_c}$, in this case *name* is assigned with the value of v; otherwise, $name = v$.

Denote $[\![e_1, e_2, ..., e_n]\!]$ as a sequence, where each element e_i has the same type, *i.e.*, a term, a name, or the aforementioned tuple. $T_1 \cdot T_2$ represents the concatenation of sequence T_1 and T_2. $T|^{t}_{t'}$ is a sequence obtained by replacing element t of sequence T with another element t'. $T_1 \backslash T_2$ represents a new sequence by removing all the elements in sequence T_1 that are the same as those in sequence T_2. We additionally define operations for a tuple sequence ω. Here, $\omega[j]$ indicates *name* of the jth tuple in ω. $\sigma(\omega)$ outputs a term sequence consisting of all *name* in ω. $g(\omega)$, $e(\omega)$ outputs a term sequence by obtaining the *name* of all tuples in ω whose $range = \mathsf{R_g}$ and $ether = \mathsf{E_y}$, respectively. The order of terms in $\sigma(\omega), g(\omega), e(\omega)$, are in accordance of the order of terms in ω.

Furthermore, to translate names into terms, we define and implement two functions σ_v, σ_a. σ_v translates a variable name into a variable term, and σ_a translates a name that represents a contract, a function, or a constant into a constant term.

4.2 Modeling the behaviors

Based on the above notations, we propose to model the initialization of contracts and transitions of configurations of KSolidity. Specifically, given a contract account of address c, we will introduce how to model the executions of functions in the contract codes of the account. For brevity, we will refer to the account of address c as account c in the following paper.

Modeling the initialization. Assume that the contract of account c is deployed on blockchain and the following data will be initialized in the corresponding configuration of KSolidity: 1) the ether balances of account c; 2) the global variables of account c. Besides, the ether balances of other accounts also need to be initialized since they may be modified during the executions of codes of account c. We use ω_0 to model the configuration of KSolidity after initialization of account c. There are three kinds of tuples in ω_0 in order: 1) $\langle \sigma_a(c), \mathsf{T_c}, \mathsf{R_o}, \mathsf{E_n}\rangle$ that represents the address of account c; 2) tuple sequence $g(\omega_0)\backslash e(\omega_0)$ denoting the global variables of account c except the variable denoting the ether balance of c; 3) tuple sequence $e(\omega_0)$ denoting the ether balance of account c and the ether balances of all accounts who have ether exchanges with c. Therefore, $\omega_0[1] = \sigma_a(c)$. The tuples in ω_0 are then used to determine the order of parameters of facts in generated rules. Hence we define the following rules to model the initialization:

$$[\mathsf{FR}(e(\omega_0))] - [\mathsf{Init_E}()] \rightarrow [\mathsf{Evar}(e(\omega_0))] \ (\texttt{init\_evars})$$

$$[\mathsf{FR}(g(\omega_0)\backslash e(\omega_0))] - [\mathsf{Init_G}(\omega_0[1])] \rightarrow [\mathsf{Gvar}([\![\omega_0[1]]\!] \cdot g(\omega_0)\backslash e(\omega_0))] \ (\texttt{init\_gvars})$$

Here, Evar represents the current ether balances of all accounts on blockchain in initialization. Gvar represents the current global variables of account c. For brevity, we use $\mathsf{FR}(e(\omega_0))$ to denote a sequence that consists of $\mathsf{Fr}(t)$ for all elements t in $e(\omega_0)$. $\mathsf{Fr}(t)$ here is a built-in fact of Tamarin prover [46] that denotes a freshly generated name, we use it to denote that term t is with arbitrary initial values. In practice, the ether balances of all accounts can be initialized once and the global variables can be initialized once for every contract account. Thus, the restrictions requiring that `init_evars` and `init_gvars` can be only applied once are added.

Translation of functions. After initialization, external accounts can send transactions to invoke any function in the contract of c. To model the invocation of functions, we define $\mathcal{R}$ partly shown in Fig. 4 to recursively translate a function in the contract into rules. Generally, in each recursive step, $\mathcal{R}$ translates a fragment of codes into a rule or multiple rules and leaves the translation of the rest in the next steps. The first argument of $\mathcal{R}$ represents the codes to be translated. If the first argument is a sequence of statements, the second argument i is a string encoding the position of the sequence in its function and $i \circ a$ denotes a string obtained by concatenating i and a string a; otherwise, if the first argument is a function, the second argument is an empty string $\varnothing$. The third argument is a tuple sequence ω.

In the following, we introduce how $\mathcal{R}$ translates a function into rules using the function `add` in Fig. 5 as an example. Since function `add` does not modify the ether balance of any account, we omit Evar fact in the rules.

First, $\mathcal{R}(\texttt{function add(uint } v2)\{\texttt{stmt}\}, \varnothing, \omega_0)$ is applied and two rules are output, which correspond to `ext_call`

$$\mathcal{R}(\texttt{function}\ f(\mathrm{d})\{\texttt{stmt}\},\varnothing,\omega_0) = \mathcal{R}(\texttt{stmt},1,[\![\langle\sigma_a(f),\mathsf{T_c},\mathsf{R_o},\mathsf{E_n}\rangle,\langle\sigma_v(c_b),\mathsf{T_v},\mathsf{R_l},\mathsf{E_n}\rangle,\langle\sigma_v(calltype),\mathsf{T_v},\mathsf{R_l},\mathsf{E_n}\rangle,\langle\sigma_v(depth,\mathsf{T_v},\mathsf{R_l},\mathsf{E_n}\rangle]\!]$$
$$\cdot\,\omega_0\cdot seq(\mathrm{d}))\cup\{[\mathsf{Fr}(\sigma_v(c_b)),\mathsf{FR}(\sigma(seq(\mathrm{d})))]-[]\rightarrow[\mathsf{Call_e}([\![\omega_0[1],\sigma_a(f),\sigma_v(c_b)]\!]\cdot\sigma(seq(\mathrm{d})))],\quad(\texttt{ext\_call})$$
$$[\mathsf{Call_e}([\![\omega_0[1],\sigma_a(f),\sigma_v(c_b)]\!]\cdot\sigma(seq(\mathrm{d}))),\mathsf{Evar}(e(\omega_0)),\mathsf{Gvar}([\![\omega_0[1]]\!]\cdot g(\omega_0)\backslash e(\omega_0))]-[]\rightarrow[\mathsf{Var_1}([\![\sigma_a(f),\sigma_v(c_b),\mathsf{EXT},0]\!]\cdot\sigma(\omega_0)\cdot\sigma(seq(\mathrm{d})))],\ \ldots\}$$
$$(\texttt{recv\_ext})$$

$$\mathcal{R}(v_1\leftarrow v_2;\texttt{stmt},i,\omega)=\mathcal{R}(\texttt{stmt},i\circ 1,\omega)\cup\{[\mathsf{Var}_i(\sigma(\omega))]-[]\rightarrow[\mathsf{Var}_{i\circ 1}(\sigma(\omega)|\frac{\sigma_v(v_1)}{\sigma_v(v_2)})]\}\qquad(\texttt{var\_assign})$$

$$\mathcal{R}(\tau\ v_1\leftarrow v_2;\texttt{stmt},i,\omega)=\mathcal{R}(\texttt{stmt},i\circ 1,\omega\cdot[\![\langle\sigma_v(v_1),\mathsf{T_v},\mathsf{R_l},\mathsf{E_n}\rangle]\!])\cup\{[\mathsf{Var}_i(\sigma(\omega))]-[]\rightarrow[\mathsf{Var}_{i\circ 1}(\sigma(\omega)\cdot[\![\sigma_v(v_2)]\!])]\}\qquad(\texttt{var\_declare})$$

$$\mathcal{R}(\texttt{return},i,\omega)=\{[\mathsf{Var}_i(\sigma(\omega))]-[\mathsf{Pred\_eq}(\omega[3],\mathsf{EXT})]\rightarrow[\mathsf{Gvar}([\![\omega[5]]\!]\cdot g(\omega)\backslash e(\omega)),\mathsf{Evar}(e(\omega))],\ \ldots\}\qquad(\texttt{ret\_ext})$$

Figure 4: Parts of the translation of functions and statements.

```
1 contract Ex2{
2     uint v1;
3     function add(uint v2) public{
4         v1 = v1 + v2;
5         return;
6     }
7 }
```

Figure 5: Example contract Ex2.

and `recv_ext` in Fig. 4, respectively:

$$[\mathsf{Fr}(\sigma_v(c_b)),\mathsf{Fr}(\sigma_v(v2))]-[]\rightarrow[\mathsf{Call_e}(\sigma_a(c),\sigma_a(f),\sigma_v(c_b),\sigma_v(v2))]$$
$$[\mathsf{Call_e}(\sigma_a(c),\sigma_a(f),\sigma_v(c_b),\sigma_v(v2)),\mathsf{Gvar}(\sigma_v(v1))]-[]\rightarrow[\mathsf{Var_1}(\sigma_a(f),\sigma_v(c_b),\mathsf{EXT},0,\sigma_a(c)\sigma_v(v1),\sigma_v(v2))]$$

The first rule denotes an event that an external account c_b sends a transaction to invoke `add`. Here $seq(\mathrm{d})=[\![\sigma_v(v2)]\!]$ is a term sequence generated according to the parameter of `add`. According to C1 in adversary model, c_b and $\sigma_v(v2)$ are initialized by using Fr facts. The second rule denotes the reception of a transaction. The $\mathsf{Var_1}$ fact represents all the values required in executing `add`, whereby terms in $\mathsf{Call_e}$, Gvar are merged into terms in $\mathsf{Var_1}$. Therefore, $\mathcal{R}$ also updates ω_0 with a sequence of the corresponding tuples. Here, *calltype* $\in\{\mathsf{EXT},\mathsf{IN}\}$ indicates whether c_b is an external account or a contract account and *depth* denotes current call depth.

Then, $\mathcal{R}$ translates the statements in the function into rules for modeling the execution of the function `add`. The assignment statement in line 4 is translated into the following rule, which corresponds to `var_assign` in Fig. 4:

$$[\mathsf{Var_1}(\sigma_a(f),\sigma_v(c_b),\sigma_v(calltype),\sigma_v(depth),\sigma_a(c),\sigma_v(v1),\sigma_v(v2))]-[]\rightarrow[\mathsf{Var_{11}}(\sigma_a(f),\sigma_v(c_b),\sigma_v(calltype),\sigma_v(depth),\sigma_a(c),\sigma_v(v1)\oplus\sigma_v(v2),\sigma_v(v2))]$$

The term $\sigma_v(v_1)$ is replaced by $\sigma_v(v_1)\oplus\sigma_v(v_2)$ when applying the rule. Here $\oplus$ is translated from the operator $+$ and introduced in the complete version of our paper [27].

Additionally, the return statement in line 5 is translated into the following rule corresponding to `ret_ext`:

$$[\mathsf{Var_{11}}(\sigma_a(f),\sigma_v(c_b),\sigma_v(calltype),\sigma_v(depth),\sigma_a(c),\sigma_v(v1),\sigma_v(v2))]-[\mathsf{Pred\_eq}(\sigma_v(calltype),\mathsf{EXT})]\rightarrow[\mathsf{Gvar}(\sigma_v(v1))]$$

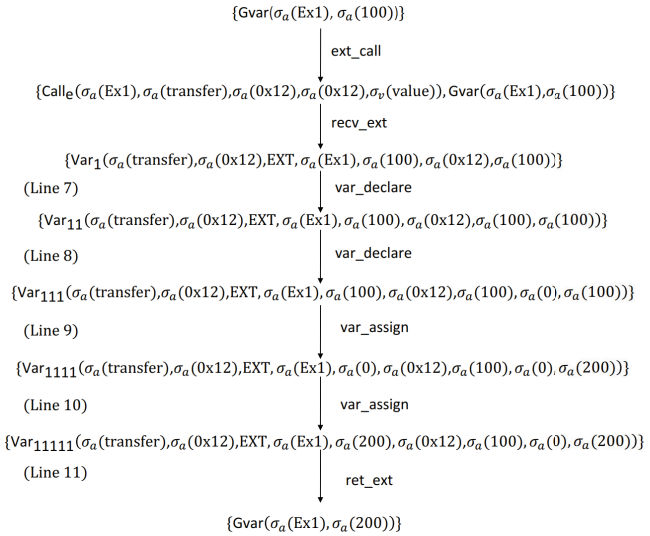

Figure 6: The execution that models an attack on Ex1.

The term $\sigma_v(v1)$ denoting the global variable of contract `Ex2` is put into Gvar facts. The local variables will no longer be used and the corresponding terms will not be maintained. Here, Pred_eq is a fact denoting equality between terms [46]. We use it to determine whether $\sigma_v(calltype)$ is equal to EXT, corresponding to the case that the function is invoked by external accounts. Similarly, this statement can be translated into a rule denoting the case that the function is invoked by contract accounts as shown in [27].

Adversaries. Here we introduce the modeling of the capability C1 and C2 of adversaries mentioned in Section 3.2, and the modeling of C3 is introduced in [27].

C1: The operation that an adversary, besides normal participants, sends transactions can also be modeled by `ext_call`. Therefore, no additional rules for the operation are provided.

C2: For each function f in the contract of account c, multiple rules are generated to indicate that if the fallback function of the adversary is triggered by the execution of the contract of c, the adversary can send an internal transaction to invoke any function f in the contract of c. The details of these rules are shown in [27].

4.3 An illustrative example

Fig. 2 shows a simplified version of a practical smart contract, which is with a vulnerability of transferMint [7]. The global variable `balances` denotes the token balances of accounts. When the function `transfer` is invoked, the token balance of `msg.sender` is supposed to decrease when the token balance of `to` increases. However, assume that the account on address $0x12$ invokes `transfer` with the parameter `to` = $0x12$, `balances`$[0x12]$ will increase while the balance of no other account will decrease. By exploiting this vulnerability, the account on address $0x12$ can mint tokens for profit or eventually make this type of tokens valueless through repeated attacks.

An execution of the model that corresponds to the attack is shown in Fig. 6. We use the contract name `Ex1` to denote the address of the account who owns this contract. Since function `transfer` does not modify the ether balance of any account, we omit Evar fact in the figure. Hence, in the execution, the initial state is $\{\mathsf{Gvar}(\sigma_a(\texttt{Ex1}), \sigma_a(100))\}$ where $\omega_0[1] = \sigma_a(\texttt{Ex1})$ and $g(\omega_0) \backslash e(\omega_0) = [\![\sigma_a(100)]\!]$. Next, an external account invokes `transfer` whereby the rule `ext_call` is applied such that $\mathsf{Call_e}(\sigma_a(\texttt{Ex1}), \sigma_a(\texttt{transfer}), \sigma_v(c_b), \sigma_v(\texttt{to}), \sigma_v(\texttt{value}))$ is added to the new state. Since $\sigma_v(c_b)$, $\sigma_v(\texttt{to})$ and $\sigma_v(\texttt{value})$ can be arbitrary values, in this execution, they can be instantiated as $\sigma_a(0x12)$, $\sigma_a(0x12)$ and $\sigma_c(100)$ respectively. In the following steps, the state is updated in similar ways. When the transaction invoking `transfer` finishes, the state is $\mathsf{Gvar}(\sigma_a(\texttt{Ex1}), \sigma_a(200))$, which implies that `balances`$[0x12]$ changes into 200 in an unexpected way. Note that the numerical instantiation cannot be supported by the original Tamarin prover. Moreover, the independent model cannot be verified directly to find an attack as shown in Fig. 6, since several technical challenges need to be addressed.

4.4 Technical challenges and main solutions

Since the module of independent modeling only provides a framework that automatically generates models of smart contracts partially, we have to address the following technical challenges to complement the model for the verification.

Challenge 1: recognizing security requirements. Given an execution shown in Fig. 6, a corresponding property is still needed for the verifier to recognize this execution as an instance of some vulnerabilities. However, there is no uniform standard for the security requirements of contracts in practical scenarios, which makes the precise generation of security properties difficult. There are automated bug-finding tools and verifiers defining patterns or properties according to known vulnerabilities [60] [38] [40]. However, the vulnerabilities covered by these tools are limited to known ones, and a variant of a known vulnerability may evade their detection [49].

To address this challenge, we perform statistical analysis on 30577 real-world smart contracts and obtain an observation: most of the smart contracts (91.11%) are finance-related, *i.e.*, the executions of these contracts may change the cryptocurrencies of themselves and others. Therefore, we divide the smart contracts into different categories according to the cryptocurrencies that they use and propose security properties to check whether the cryptocurrencies may be lost unexpectedly.

Challenge 2: contract-oriented automated reasoning. Given an independent model, the rule `ext_call` can be applied repeatedly, which is corresponding to the practical scenarios that a function can be invoked any times. This may lead to non-termination of verification. Besides, the independent model is insufficient for verifying 2-safety properties [31].

We address the challenge based on the fact that a transaction is atomic and cannot be interfered by other transactions. Therefore, the independent model can be reduced for different types of properties: (1) the properties that should be maintained for a single transaction; (2) the properties that may be affected by other transactions. For the first type, we propose to automatically generate invariant properties and the corresponding reduced model that the behaviors of other transactions are ignored. For the second type, since a transaction is atomic, the rest way to trigger an attack is to leverage different results of a sequence of transactions caused by different orders of the transactions or different block variables. Hence, we propose the equivalence properties and also the modeling method to achieve effective automated reasoning. We also modify the code of Tamarin for supporting the verification where numerical constraint solving is additionally required.

5 PROPERTY GENERATION

To address Challenge 1, we divide finance-related smart contracts into three categories according to the type of cryptocurrencies they use: ether-related, token-related, and indirect-related. An ether-related contract may transfer or receive ethers, *i.e.*, the official cryptocurrency of Ethereum. Similarly, a token-related contract may send or receive tokens, *i.e.*, the cryptocurrency implemented by the contract itself. An indirect-related contract is used in the former two contracts to provide additional functionality. Hence, to check whether the cryptocurrencies may be lost in unexpected ways, we focus on generating security properties for ether-related and token-related contracts. We propose to recognize the category and key variables related to cryptocurrencies from the codes of smart contracts and use the information to generate the security properties. Note that we analyze the indirect-related contracts in an indirect way and do not generate properties for the indirect-related contracts. For example, given a token contract C_1 and an indirect-related contract C_2, assume that C_1 implements authentication by invoking functions in C_2. In this case, we can specify C_1 to be analyzed and then generate a model for it, which considers the interaction of C_1 and C_2.

5.1 Recognizing categories and key variables

Ether-related contracts. The ethers can be transferred by using official functions, *e.g.*, `transfer`, `send` and `call`. The modifier `payable` is only used in ether-related contracts for receiving ethers. Therefore, FASVERIF recognize an ether-related contract by determining if there are keywords, *i.e.*, `transfer`, `send`, `call` or `payable` in the contract. If a contract is recognized as an ether-related contract, then we use the built-in variable `balance` as the key variable, which denotes the ether balance of an account.

Token-related contracts. The token-related contracts can be divided into token contracts and token managing contracts. A token contract is used to implement a kind of customized cryptocurrency, *i.e.*, tokens, which can be traded and have financial value. A token managing contract, *e.g*, an ICO contract [5], is used to manage the distribution or sale of tokens.

We propose a method to recognize token contracts based on another observation from our statistical result in Section 7.2: developers tend to use similar variable names to represent the token balance of an account. Therefore, a contract is identified as a token contract, if there is a variable of type `mapping(address=>uint)` with a name similar to two commonly used names: `balances` or `ownedTokenCount`. Specifically, we calculate the similarity of names using Python package fuzzywuzzy [8]. When the similarity is larger than 85, we consider two names similar. The threshold 85 is set based on our evaluation in Section 7.2. For the contracts using uncommon names for token balances, FASVERIF also supports the users to provide their own variable names. In addition to `balances`, we observe that some token-related contracts define a variable of `uint` type to record the total number of tokens. Similarly, we use the most common variable name `totalSupply` to match the variables representing the total amount of tokens. This kind of variables are not used to recognize the token contract, but rather for the subsequent generation of properties. After the recognition of token contracts, we search for contracts instantiating token contracts and regard them as token managing contracts.

5.2 Generating security properties

As mentioned in Section 4.4, we propose two kinds of properties: invariant properties and equivalence properties.

Invariant properties. The invariant property requires that for any transaction a proposition (a statement that denotes the relationship between values of variables) ϕ holds when the transaction finishes, if ϕ holds when the transaction starts executing. Since a transaction is atomic, FASVERIF checks invariant properties in single transactions instead of the total executions to achieve effective automated reasoning. Here, we design the invariants to ensure that the token balances in token-related contracts are calculated in an expected way. Note that we do not design invariant properties for ether-related contracts, since the calculation of ether balances is performed by the EVM and its correctness is guaranteed [25].

For a token contract with key variable balances, the following invariant is generated:

$$\sum_{a\in A_1} \mathsf{balances}(a) = C_1 \qquad (\texttt{token\_inv})$$

Since a transaction can only affect a limited number of accounts, A_1 is the set of addresses of the accounts whose token balances may be modified in the transaction. C_1 is an arbitrary constant value and the invariant implies that the sum of token balances of all accounts should be unchanged after a transaction. If the invariant is broken, it indicates an error in the process of recording token balances, which would make this kind of tokens worthless [6]. Here, balances can be replaced by any variable name denoting the token balances. If there are multiple variables denoting token balances of different types, all of them will be used. Specially, if there is a key variable totalSupply denoting the total amount of tokens in the token contract, the constant C_1 in `token_inv` will be replaced by totalSupply. For a token managing contract, the invariant `token_inv` is generated for the token contract that it manages. FASVERIF also supports the users to provide customized invariants to check the security of contracts.

Equivalence properties. We define the equivalence property as follows: The equivalence of a global variable v holds for a transaction sequence T, if the value of v after T's execution is always the same. Here we study the equivalence of the token or ether balance of the adversary. Given two sequences T_A and T_B that have the same transactions, we propose the following property:

$$\mathsf{balances}_A(c_{adv}) = \mathsf{balances}_B(c_{adv}) \wedge \\ \mathsf{balance}_A(c_{adv}) = \mathsf{balance}_B(c_{adv}) \qquad (\texttt{equivalence})$$

Here, denote $\mathsf{balances}_A(c_{adv})$ and $\mathsf{balance}_A(c_{adv})$ as the token balance and ether balance of the adversary after execution of T_A, respectively. Similarly, $\mathsf{balances}_B(c_{adv})$ and $\mathsf{balance}_B(c_{adv})$ represent the corresponding balances for T_B. `equivalence` requires that the adversary cannot change its own balances by changing the orders of transactions or other conditions; otherwise, the difference of the balances may be the illegal profit of the adversary.

5.3 Relationship between properties and common vulnerabilities

```
1 function f(){
2     require(!transfered);
3     msg.sender.call.value(10);
4     transfered = True;
5 }
```

Figure 7: An example with reentrancy vulnerability.

The properties of FASVERIF are designed with a basic idea: leveraging the phenomenon that the loss of ethers and tokens is one of the popular intentions of attackers [33]. FASVERIF generates properties based on key variables denot-

ing the token balances or ether balances. We aim to cover vulnerabilities causing financial losses. As a result, FASVERIF covers 6 types of vulnerabilities, including *transferMint* not supported by existing automatic tools, through the two properties. Note that these vulnerabilities do not necessarily cause financial loss and those that do not are ignored by FASVERIF as they do not affect the financial security of contracts.

To explain the usage of our properties, we provide examples of several common vulnerabilities, detailing how contracts with these vulnerabilities violate the above two properties.

Gasless send. During the executions of official functions `send` and `call`, if the gas is not enough, the transaction will not be reverted and a result will return. If a contract does not check the execution results of `send` or `call`, it may mistakenly assume that the execution was successful. Given two sequences of same transactions that invoke functions with gasless send vulnerability, one with sufficient gas and one without sufficient gas, the results of them will be different. Therefore, the equivalence property is violated.

Reentrancy. Taking the contract in Fig. 7 as an example, suppose that the adversary sends a transaction to invoke `f` and the statement on line 4 sends ethers to the adversary. According to Section 3.2, the adversary can then send an internal transaction through the fallback function to call `f` again, and since the code on line 5 is not executed, the check on line 3 will still be passed, allowing the adversary to get ethers one more time. Assume that there are two sequences T_1 and T_2, T_1 consisting of two transactions invoking `f` and T_2 consisting of one transaction invoking `f` and one internal transaction invoking `f` through a reentrancy vulnerability. We treat the internal transaction sent by the adversary as a transaction and consider T_1 and T_2 as consisting of the same transactions. In this case, the ether balances of adversary after T_1 and T_2 are different, which means that the equivalence property is broken.

TD&TOD. When some statements are control dependent on the `block.timestamp`, the adversary can control the execution of these statements by modifying `block.timestamp` in a range, which is called TD. Given two same sequences of transactions that invoke a function with TD, the execution results may be different with different `block.timestamp`, which violates the equivalence property. Similarly, the contracts with TOD vulnerability violate our equivalence property when the order of transactions changes.

Overflow/underflow. Overflow/underflow is a kind of arithmetic error. Since the goal of FASVERIF is to analyze the financial security of contracts, FASVERIF detects overflow/underflow vulnerabilities that can change the number of tokens. For the remaining overflow/underflow vulnerabilities, FASVERIF can also support them through custom invariants.

Certain new vulnerability can be detected directly by FASVERIF if it is covered by our properties, such as the transferMint vulnerability. If the vulnerability is not covered, we need to propose new properties or modify the rules in our models to support more features. For example, the airdrop hunting vulnerability [61], which is used by attackers to collect bonuses from airdrop contracts, is not currently supported by FASVERIF. To extend FASVERIF to cover airdrop hunting, we can propose a new invariant requiring the number of contract accounts to remain zero. However, it is challenging to model the identification of contract accounts. We would like to study the extension of FASVERIF in our future work.

6 COMPLEMENTARY MODELING AND VERIFICATION

In this section, we introduce how we address Challenge 2. According to different properties of a contract, we propose the method of complementary modeling to generate customized models built upon the independent models with rules replaced or added. Besides, we propose a solution to check whether a customized model satisfies the corresponding property.

6.1 Complementary modeling

The goal of complementary modeling is to generate a customized model, which satisfies that the invariant property or equivalence property is not valid in the KSolidity Semantics, only if there exists an execution in the model that breaks the property. Besides, to support automated verification, the model is added with more constraints such that each execution that reaches a certain state breaks the property. Then, the property is not valid if and only if the state is reachable. Hence, we design the method for invariant property and equivalence property as follows.

Invariant properties. The generated model for invariant properties has the following features:

i) The invariant holds at the beginning of any execution. **ii)** An execution simulates the execution of one transaction. **iii)** The invariant is assumed to be broken at the end of any execution, which corresponds to the state that breaks the property.

To make the generated model conform to feature i), we first replace the rule `init_gvars` with rule `init_gvars_inv`. In rule `init_gvars_inv`, a fact $\theta_e(\phi)$ is added to denote that the invariant ϕ holds after the initialization. Here, ϕ is the invariant `token_inv` in Section 5.2 and $\theta_e(e)$ is a function translating mathematical expressions into numerical facts in rules. Numerical facts denote the relationships between numeric variables and are processed in the verification module. Similarly, we define $\theta_{ne}(e)$ to translate the negation of e.

Then we replace the rule `ext_call` with `ext_call_inv`, added with an action and a restriction requiring that the rule can be applied only once, to achieve feature ii).

Finally, we modify the rule `ret_ext` into `ret_ext_inv`. `ret_ext_inv` has additional facts $\theta_{ne}(\phi)$ and $\mathsf{End}()$ compared to `ret_ext`, which together achieve feature iii). $\mathsf{End}()$ serves as an indicator that an execution of the model reaches the end of the transaction if rule `ret_ext_inv` is applied, and $\theta_{ne}(\phi)$ means that invariant ϕ is broken at the same time.

Equivalence properties. The generated model for equivalence properties has the following features: **i)** An execution of the model simulates the executions of two sequences T_A and T_B consisting of the same transactions but possibly with different orders. **ii)** Before the executions of T_A and T_B, the values of global variables and ether balances of all accounts are the same. **iii)** The ether or token balances of the adversary are assumed to be different at the end of any execution, which corresponds to the state that breaks the equivalence property.

Firstly, to achieve feature ii), we replace `init_evars` and `init_gvars` with `init_evars_AB` and `init_gvars_AB`, respectively. In `init_gvars_AB`, the Gvar fact is duplicated into Gvar_A and Gvar_B facts, which indicates that the global variables are the same before T_A and T_B. Similarly, the Evar fact is duplicated in `init_evars_AB`.

Then, we replace `ext_call` with `ext_call_AB`, in which Call_e fact is duplicated, indicating that two transactions with same parameters and same sender are sent.

Except rules `init_evars`, `init_gvars`, `ext_call`, each of the remaining rules in the model is replicated into two rules, and the facts of the two rules are added with different subscripts A and B to represent the execution of transactions in sequences T_A and T_B, respectively. For example, the rule `recv_ext` is replaced with `recv_ext_A` and `recv_ext_B`. Specially, actions and restrictions are added into `recv_ext_A` and `recv_ext_B` to achieve feature i). The complete form of the above rules is shown in [27].

Finally, to achieve feature iii), we add a rule `compare_AB` to compare the ether balances and token balances of the adversary, where $\theta_{ne}(\phi_{equ})$ and $\mathsf{End}()$ are added for subsequent verification. ϕ_{equ} is property `equivalence` in Section 5.2.

6.2 Verification

The verification module is implemented by modifying the source code of Tamarin prover [46] to achieve modeling using multiset rewriting rules with additional support for numerical constraint solving by Z3 [26]. Taking a generated property and the corresponding model as input, the workflow of this module is as follows: 1) Search for an execution that reaches $\mathsf{End}()$ without considering the numerical constraints. 2) If the search fails, the module terminates and outputs that the property is valid; otherwise, go to step 3). 3) Collect the numerical constraints that the execution must satisfy and solve the constraints by Z3. 4) If the set of constraints is satisfied, which indicates that the execution that violates the property exists, the module terminates and outputs the execution as a counterexample; otherwise, add a constraint to the model that the execution does not exist, and go to step 1).

6.3 Formal guarantee

We prove the soundness of translation from Solidity language to our models based on KSolidity [39], which is claimed to fully cover the high-level core language features specified by the official Solidity documentation and be consistent with the official Solidity compiler. However, the completeness of our translation is not guaranteed due to two reasons: 1) the initialization of global variables and ether balances in rules `init_evars` and `init_gvars` assumes the initial values of global variables and ether balances to be arbitrary, which may over-approximate the range of values for these variables. 2) the specific values of the block timestamps are not considered. Specifically, we prove Theorem 1 (informal description). Note that Theorem 1 only holds for the contracts supported by FASVERIF (See Section 9). The precise description of Theorem 1 is presented in [27].

Theorem 1 *(Soundness). If an invariant property (or equivalence property) holds in the complementary model of FASVERIF, it holds in real-world transactions interpreted by KSolidity semantics.*

Proof *Please refer to [27].*

7 EVALUATION

In this section, we firstly make preparations on the experimental setup, including the types of vulnerabilities, datasets, and representative tools that we choose. Then, we report the experimental results and analyze the effectiveness of FASVERIF. Finally, we verify real-world contracts using FASVERIF and demonstrate the exploitable bugs that FASVERIF finds.

7.1 Experimental setup

Types of Vulnerabilities. First, we introduce the vulnerabilities that FASVERIF currently targets. We divide the 37 types of vulnerabilities in SWC Registry [10], a library consisting of smart contracts' vulnerabilities, into three categories: a) vulnerabilities that can be detected through syntax checking, *e.g.*, outdated compiler version. b) vulnerabilities that do not have clear consequences, *e.g.*, dangerous delegatecall. c) vulnerabilities that can cause losses of ethers or tokens. FASVERIF targets the vulnerabilities in category c) as they can cause financial loss and are difficult to detect. There are 6 types of vulnerabilities that FASVERIF currently supports: 1) *transaction order dependency (TOD)*; 2) *timestamp dependency (TD)*; 3) *reentrancy*; 4) *gasless send*; 5) *overflow/underflow*; 6) *transferMint* [7]. The relationship between these vulnerabilities and our properties has been mentioned in Section 5.3. We divide the *TOD* vulnerabilities into two groups: *TOD-eth* changing ether balances of accounts, *TOD-token* changing token balances of accounts, since SECURIFY and OYENTE only support the detection of the former.

Datasets. We use two datasets [20] of smart contracts to evaluate FASVERIF. The first dataset, called *vulnerability dataset*, is used to test the performance of FASVERIF in detecting different types of vulnerabilities compared with other

Table 1: A comparison of representative automated analyzers for smart contracts. (Acc and F1 outside brackets correpsond to the finance-vulnerable contracts, while those inside brackets correpsond to the vulnerable contracts, * denote automated verifiers)

Types of Vulnerabilities	Osiris		SECURIFY*		Mythril		OYENTE		VERISMART		SmartCheck		Slither		Manticore		eThor*		FASVERIF *		
	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	Acc(%)	F1	U
TOD-eth	/	/	96.43	0.98	/	/	42.86	0.6	/	/	/	/	/	/	/	/	/	/	100	1	10
TOD-token	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	100	1	0
TD	71.60 (70.37)	0.83 (0.82)	/	/	45.68 (44.44)	0.62 (0.62)	76.54 (75.31)	0.87 (0.86)	/	/	/	/	16.05 (14.81)	0.26 (0.25)	24.69 (23.46)	0.38 (0.38)	/	/	95.06 (93.83)	0.97 (0.96)	33
reentrancy	66.67 (69.05)	0.79 (0.81)	78.57 (76.19)	0.85 (0.84)	71.42 (69.04)	0.81 (0.8)	73.81 (76.19)	0.85 (0.86)	/	/	73.81 (76.19)	0.85 (0.86)	85.71 (83.33)	0.91 (0.90)	38.09 (35.71)	0.41 (0.40)	83.72 (86.05)	0.92 (0.93)	90.48 (88.10)	0.94 (0.93)	2
gasless send	/	/	92.19	0.95	82.35	0.67	/	/	/	/	92.19	0.95	85.94	0.91	29.69	0.26	/	/	100	1	7
overflow/underflow	81.20 (81.20)	0.89 (0.89)	/	/	95.30 (95.30)	0.97 (0.97)	90.27 (90.27)	0.95 (0.95)	98.99 (98.99)	0.99 (0.99)	/	/	/	/	19.40 (19.40)	0.11 (0.11)	/	/	99.33 (99.33)	0.99 (0.99)	4
transferMint	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	/	100	1	0

automated tools. We collect 611 smart contracts with vulnerabilities in category c) mentioned above from public dataset of other works [38] [53] [42] [36]. We filter out 6 smart contracts whose codes are incomplete and 56 smart contracts that FASVERIF does not support. We illustrate the number of contracts unsupported by FASVERIF in the last column of Table 1, and the reasons that FASVERIF does not support them will be introduced in Section 9. Finally we get *vulnerability dataset* with 549 contracts. The second dataset, called *real-world dataset*, is used to evaluate the effectiveness of FASVERIF in detecting real-world smart contracts. We crawl 46453 Solidity source code files from Etherscan [19], and then filter the contracts to remove duplicates. We calculate the similarity of two files using difflib [17] package of Python, and considered two contracts as duplicates when their similarity is larger than 90%. Finally, we obtained 17648 Solidity files containing 30577 contracts as *real-world dataset*. We add 11 smart contracts with the vulnerability of *transferMint* from the *real-world dataset* to the *vulnerability dataset*, since the previous datasets have not gathered this type of contracts.

Tools. We compare FASVERIF with the following representative automatic tools: OYENTE [44], Mythril [35], SECURIFY (version 2) [60], ContractFuzzer [38], Osiris [58], Slither [37], SmartCheck [55], VERISMART [53], Manticore [23] and eThor [52].

We do not compare FASVERIF with ZEUS [40], another automated verifier, since it is not publicly available. Besides, we do not compare FASVERIF with semi-automated verification frameworks while they need manual input of properties, which require certain expertise and is labor-intensive when evaluating hundreds of contracts. Meanwhile, how to express the properties in different specification languages equivalently becomes a problem and may affect the fairness of comparison.

Experimental Environment. We experiment on a server with 2.50GHz CPU, 128G memory and 64-bit Ubuntu 16.04.

7.2 Statistical analysis

We first perform statistical analysis on *real-world dataset*. We manually classify the contract as finance-related or others taking the following parts of contracts into account and try our best to avoid misclassification: 1) Contract names. The usage of some contracts can be shown in their name. 2) Contract annotations. The annotations of contracts can provide us some information, *e.g.*, the contracts' usage.3) Inheritance of contracts. The children of token contracts can possibly be token contracts. 4) Contract creation statements. The contracts creating token contracts can possibly be token managing contracts. 5) Ether transfer statements. The contracts transferring ethers are ether-related. Note that the contracts that are difficult to distinguish their usage are classified as others.

Table 2: The effectiveness of our method for identifying token contracts.

threshold	70	75	80	85	90
Acc(%)	98.31	98.32	98.32	98.50	98.46
F1(%)	98.13	98.14	98.14	98.31	98.27

After the above classification, we find 27858 finance-related contracts, including 6307 ether-related contracts (20.63%), 7661 token-related contracts (25.05%), 5994 contracts both ether-related and token-related (19.60%) and 7896 indirect-related contracts in total (25.82%). The remaining contracts account for 8.89%. Hence, finance-related contracts make up a major portion (91.11%) of the real-world contracts, which validates the goal of generating properties aiming to protect cryptocurrencies shown in Section 5.2.

During the classification, we find that since the official ERC20 [11] standard of Ethereum recommends using variable name `balances` to denote token balances, most token contracts use names similar to `balances` to denote token balances. Besides, there are also token contracts using names similar to `ownedTokenCount` due to ERC721 [18] standard.

To validate our observation and evaluate the effectiveness of our methods to identify token contracts, we perform an evaluation on *real-world dataset*. We search for contracts with variables of type `mapping(address=>uint)` that have names similar to `balances` or `ownedTokenCount`, while the similarity of two names is calculated based on fuzzywuzzy [8], and the thresholds are set to 70, 75, 80, 85 and 90, respectively. We collect the following data under different thresholds: 1) *TP*: the number of token contracts correctly identified. 2) *FN*: the number of token contracts that are missed. 3) *FP*: the number of contracts misclassified as token contracts. 4) *TN*: the number of contracts that are not token contracts correctly classified. 5) *Accuracy*: $Acc = \frac{TP+TN}{TP+TN+FP+FN}$. 6) *F1*: $F1 = \frac{2TP}{2TP+FP+FN}$. We only show 5) and 6) in Table 2 due to

the page limit. According to Table 2, our method achieves Accuracy and F1-score higher than 98% under different thresholds. We choose 85 as our threshold finally.

7.3 Comparison

Unlike other automatic tools, FASVERIF detects the effect of the vulnerabilities, *i.e.*, whether causing the financial loss. To fairly compare FASVERIF and other automatic tools, we run the tools on *vulnerability dataset*, and collect two sets of results as shown in TABLE 1. Here we call a contract with a vulnerability as a vulnerable contract, and call a contract with a vulnerability causing financial loss as a finance-vulnerable contract. We regard the number of contracts correctly recognized as a finance-vulnerable / vulnerable contract as *TP*, and regard the number of contracts correctly recognized as a contract that is not finance-vulnerable / vulnerable as *TN*. The calculation formulas of accuracy and F1 are mentioned above. Due to the page limit, we only show the accuracy and *F1* of tools in TABLE 1. Note that *TOD-eth*, *TOD-token*, *gasless send* and *transferMint* always cause financial loss, thus the two sets of results for them are the same.

Totally, FASVERIF outperforms the representative tools that it achieves higher accuracy and F1 values in the detection of vulnerable and finance-vulnerable contracts in *vulnerability dataset*. Meanwhile, FASVERIF is the only one that is able to detect all the types of vulnerabilities in TABLE 1 among the automated tools mentioned above. Note that we fail to make ContractFuzzer report any findings. Though being in contact with the authors, we are unable to fix the issue and both sides eventually give up. SECURIFY can output alerts for all contracts using timestamp, but is not targeted to detect *TD*, so we do not compare its ability to detect *TD* with FASVERIF.

We analyze the reason for the false results produced by different automatic tools shown in Table 1.

TOD-eth*,*TD*, *gasless send: The above automatic tools detect these types of vulnerabilities based on their pre-defined patterns and their accuracy depends on the patterns. On one hand, progressive patterns can result in false negatives. For example, SECURIFY decides if a contract is secure against *gasless send* by matching the pattern whether each return value of `send` is checked. However, a contract checks the result of `send` but does not handle the exception, which evades the detection of SECURIFY. On the other hand, conservative patterns can lead to false positives. For example, OYENTE and SECURIFY detect *TOD-eth* according to the pattern that when the transaction orders changes, the recipient of ethers may also change. A contract returns ethers to their senders and the first sender will be the first receiver. For this case, both OYENTE and SECURIFY falsely report *TOD-eth* vulnerability. However, all senders eventually receive ethers, *i.e.*, the result is not changed with the transaction order, whereas our equivalence property holds. Besides, the tools using symbolic execution, *e.g.*, OYENTE and Mythril, may produce false negatives as they explore a subset of contracts' behaviors.

reentrancy: EThor defines a property: an internal transaction can only be initiated by the execution of a `call` instruction, which over-approximates the property that a contract free from *reentrancy* should satisfy. Therefore, eThor gets more false positives than FASVERIF in detection of *reentrancy*. The reasons for the false reports of the other tools in the detection of *reentrancy* are still inaccurate patterns.

Overflow/underflow: OYENTE, Mythril, Osiris assume that the values of all the variables are arbitrary and output FPs for this category. Differently, FASVERIF and VERISMART consider additional constraints of variables, e.g., for the variables whose values are constant, their values should be equal to the initial values. VERISMART outputs 2 false positives due to its assumption: every function can be accessed.

FASVERIF also produces 9 false negatives due to the error of property generation. Specifically, FASVERIF fails to detect 2 contracts with *overflow*. In these two contracts, the variable `allowance` may overflow. We currently do not design the invariants for this variable. So we manually define a new invariant according to the two contracts and FASVERIF successfully discovers the vulnerabilities. FASVERIF also misses 3 contracts with *TD* and 4 contracts with *reentrancy*. These contracts use uncommon variable names to denote token balances. We manually specify the key variable names and finally find out the missed vulnerabilities.

To compare the efficiency of the above tools, we calculate the average time taken by them to analyze one contract in *vulnerability dataset* as follows: Slither (2.16 s), SmartCheck (4.93 s), eThor (11.95 s), OYENTE (20.81 s), Mythril (55.00 s), VERISMART (63.45 s), Osiris (73.52 s), SECURIFY (222.99 s), FASVERIF (829.61 s).

7.4 Security analysis of real-world smart contracts

To evaluate the effectiveness of FASVERIF in real-world contracts, we conduct an experiment on randomly-selected 1700 contracts from *real-world dataset*. FASVERIF reports 15 contracts with vulnerabilities, of which 11 violates the invariant property and 4 violates the equivalence property. We simulate attacks on these contracts on a private chain of Ethereum and check the exploitability of the vulnerabilities in them with on-chain states. We eventually find that among the 15 contracts, there is one contract destroyed and another contract with non-exploitable vulnerabilities, whereas the vulnerabilities in the remaining 13 contracts are exploitable. Among the exploitable bugs, there are 10 of *transferMint* vulnerabilities, which cannot be detected by existing automatic tools as shown in Table 1. Considering the proportion of vulnerable contracts found and the vulnerabilities in them causing financial losses, we hope our work can raise security concerns. The unexploitable contract is a crowdsale contract selling tokens. The contract specifies that users who buy tokens within a

certain time frame can get bonuses. However, the bonuses are no longer available after September 7, 2017, thus the vulnerability in this contract is not exploitable but misclassified due to the incompleteness of FASVERIF.

Ethical Considerations. As Ethereum accounts are anonymous, we attempt to identify the owners of the vulnerable contracts by checking the contract code, the addresses of the contract creators, and 685 bug bounty programs [16]. We also use a chat software [15] to send messages to the addresses of the contract creators but do not receive replies after waiting for 40 days. To avoid the abuse of these vulnerabilities, we do not provide the addresses of the vulnerable contracts or open-source FASVERIF. Instead, we present a simplified version of the destroyed contract and provide a website with an interface to use the restricted version of FASVERIF [21]. Also, our tool is available upon request for researchers with validated identities for academic purposes.

Example. The contract Ex1 shown in Fig. 5 is a contract with an exploitable bug. Note that this contract is simplified. In practice there are conditional statements to avoid numerical operations causing *overflow/underflow*. FASVERIF recognizes Ex1 as a token contract and chooses the invariant `token_inv`. Specifically, assume that the sum of token balances of `to` and `msg.sender` before the transaction, *i.e.*, $\texttt{balances}_0[\texttt{to}] + \texttt{balances}_0[\texttt{msg.sender}]$ is value C_1, FASVERIF checks whether the sum after the transaction, i.e., $\texttt{balances}_1[\texttt{to}] + \texttt{balances}_1[\texttt{msg.sender}]$ can be different from C_1. In the verification, FASVERIF finds an execution that reaches End() and has a constraint Pred_eq(`to`,`msg.sender`). According to the constraint, `msg.sender` in all expressions are replaced with `to`. Moreover, since the rules `var_declare` and `var_assign` are in the execution, $\texttt{balances}_1[\texttt{to}]$ is replaced by $\texttt{balances}_1[\texttt{to}] + \texttt{value}$. Hence, the constraints $\texttt{balances}_0[\texttt{to}] + \texttt{balances}_0[\texttt{to}] = C_1$, $\texttt{balances}_0[\texttt{to}] + \texttt{value} + \texttt{balances}_0[\texttt{to}] + \texttt{value} \neq C_1$ are added to Z3. As a result, the constraints are satisfied with $\texttt{value} \neq 0$, which indicates the invariant `token_inv` is broken and FASVERIF decides this contract as vulnerable. Comparatively, SECURIFY, OYENTE and Mythril fail to detect this type of vulnerability with unknown patterns. VERISMART cannot detect this vulnerability that does not cause *overflow/underflow*.

We set the verification timeout as 5 hours but 12 contracts cannot be verified within that time. The remaining contracts take an average of 2 hours and 40 minutes to verify. During the verification, we manually set variable names for 14 contracts in which FASVERIF cannot find key variables.

Besides, we compare TeEther [43] with FASVERIF on the 1700 real-world contracts. TeEther aims to reveal critical parts of code that can be abused to get ethers and assumes that if the attacker as an external account can obtain ethers from a contract, the contract is vulnerable. TeEther considers two contracts vulnerable, while FASVERIF considers them non-vulnerable. For the first contract, the attacker can destroy it whereby get ethers, but FASVERIF cannot detect this vulnerability which is not covered by our properties. For the second contract, the attacker cannot disrupt its execution and can only get ethers in normal ways. FASVERIF considers this contract safe since no ether or tokens will be lost unexpectedly.

8 RELATED WORK

8.1 Automated bug-finding tools for contracts

Automated bug-finding tools fall into two categories: tools using symbolic execution and tools using other technologies. Among the tools using symbolic execution, OYENTE [44] executes EVM bytecode symbolically and checks for vulnerability patterns in execution traces. Mythril [35] uses taint analysis and symbolic execution to find vulnerability patterns. Osiris [59] is specially designed for detecting arithmetic bugs. In the tools using other technologies, ContractFuzzer [38] instruments EVM to search for executions that match patterns. SmartCheck [56] searches for specific patterns in the XML syntax trees of contracts. VERISMART [53] generates and checks invariants to find the overflow in smart contracts.

Compared with the above tools, there are differences between FASVERIF and them: 1) FASVERIF provides a proof of our translation and implements the verification using formal tools. 2) The vulnerabilities detected by these tools are in a particular category or dependent on pre-defined known patterns. Comparatively, FASVERIF generates security properties on demand and covers various types of vulnerabilities.

8.2 Verification frameworks for contracts

Verification frameworks formally verify the properties of contracts. SMARTPULSE [54] is used to check given temporal properties of smart contracts. Similarly, VerX [48] performs a semi-automatic verification of temporal safety specifications. ConCert [29] is a proof framework for functional smart contract languages. These tools can verify functional properties of contracts, which are not currently supported by FASVERIF, but need human involvement to produce results. Differently, FASVERIF can generate and verify finance-related properties for contracts automatically. Besides, according to their literature, the above tools cannot verify our equivalence properties. CFF [30] is a formal verification framework for reasoning about the economic security properties of DeFi contracts. CFF proposes *extractable value* (EV), which is similar to our equivalence property. Specifically, the equivalence property is used to check whether an adversary can obtain profits through operations such as reordering transactions, while EV is used to quantify the profits an adversary can obtain. However, CFF takes into account more financial features, e.g., changes in exchange rates, which are not considered in FASVERIF.

8.3 Automated verifiers for contracts

To the best of our knowledge, there are three automated verifiers for smart contracts: eThor [52], SECURIFY [60] and ZEUS [40]. eThor is a sound static analyzer that abstracts the semantics of EVM bytecode into Horn clauses. As the literature of eThor states, it can only detect *reentrancy* or check assertions automatically. In addition, eThor cannot verify our equivalence property. SECURIFY detects specified patterns extracted from control flows of contracts. SECURIFY cannot solve numerical constraints and thus cannot detect *overflow* and *transferMint*. ZEUS transforms smart contracts into LLVM bitcode and uses existing symbolic model checkers. The transformations are claimed to be semantics preserving which however are refuted by [52]. Besides, ZEUS uses predefined policies based on known patterns. Thus, ZEUS may miss unknown vulnerabilities or variants of known vulnerabilities, *e.g.*, *transferMint* supported by FASVERIF.

8.4 Generating properties for other verifiers

We investigate whether our properties can be used by other verifiers. We study the following verifiers that can verify properties automatically: ZEUS, VerX, SECURIFY, eThor and SMARTPULSE. Among them, ZEUS is not publicly available, and VerX only provides a website that is no longer maintained. SECURIFY cannot solve numerical constraints and thus cannot verify our properties. eThor analyzes the bytecode of contracts, ignoring semantic information like variable names, so it is non-trivial to convert our properties, which require variable names as part of them, into a form that eThor can verify. Besides, we fail to make SMARTPULSE [12] work by following the instructions on its webpage.

9 LIMITATIONS AND DISCUSSION

Limitations. We summarize limitations of FASVERIF as follows:

1) The average time to analyze a contract using FASVERIF is longer than the one using other automated tools. According to the experiment on *vulnerability dataset*, FASVERIF take an average of 829.61 seconds to analyze a contract, while the most time-consuming one of the other automated tools take an average of 222.99 seconds.

2) FASVERIF currently cannot detect vulnerabilities that do not cause financial losses, *e.g.*, the overflow vulnerabilities that lead to DoS, which is supported by some automated tools.

3) FASVERIF can only support vulnerabilities that are covered by our properties under our assumptions. Specifically, we do not consider the exchange rates and focus only on vulnerabilities that result in abnormal token amounts or that allow attackers to gain differently with different transaction orders or block timestamps. Thus, the economical security property (considering the exchange rates) proposed in [30], the airdrop hunting and self-destruction vulnerabilities (not covered by our properties) are unsupported by FASVERIF.

4) Solidity language is not fully supported. Due to the Turing-completeness of Solidity [24], it is challenging to fully support its features. Thus, we add the following restrictions to define a fragment of Solidity supported by FASVERIF:

- Loops. FASVERIF supports unrolling of bounded loops, i.e., the execution times of loops are constant, where the loop statement is replaced by equivalent statements without loops. The unbounded loops, whose execution times cannot be determined statically, are not supported. We find 2988 contracts (9.77%) with unbounded loops in *real-world dataset* and omit them in our analysis.
- Revert. FASVERIF verifies the properties under the assumption that all transactions can be executed to completion. For transactions where a revert occurs, we assume that the executions of the transactions do not result in the modification of any variables.
- Contract creation. FASVERIF supports the case of static creation of contracts in the constructors. To trade off efficiency and coverage for Solidity features, we omit the contracts creating contracts via function calls. However, we only find 4.67% (1428/30577) of contracts in *real-world dataset* that create contracts via function calls.
- Function call. Given a set of contracts with Solidity codes, FASVERIF requires them not to invoke functions in contracts outside the set whose codes are unknown. FASVERIF can only analyze codes given beforehand, which is an inherent defect of static analyzers [47]. We also count the contracts calling unknown codes in *real-world dataset* and finally find 1754 contracts (5.74%).

In summary, even with the above restrictions, FASVERIF can still cover 82.41% (25197/30577) real-world contracts.

5) FASVERIF may get incorrect key variables or invariants. Though our method of identifying key variables achieves accuracy higher than 98% in *real-world dataset*, it still may misidentify some key variables. Additionally, the correctness of the generated invariants is also not guaranteed. As a result, incorrect variables or invariants can lead to legitimate contracts being ruled out. Thus, we offer users the option to manually set invariants and key variables instead.

6) The incompleteness of FASVERIF may lead to misclassifying safe contracts as vulnerable, *e.g.*, the online contract that is unexploitable mentioned in Section 7.4.

Discussion. We choose Tamarin due to its well-supported modeling of concurrent systems [45]. Using Tamarin gives us the flexibility to add or modify rules in our models to verify hyperproperties [32] like the equivalence properties requiring simultaneous reasoning of multiple executions. In comparison, using other tools may introduce more difficulties when modeling and verifying hyperproperties [32] [28]. However, our extensions to Tamarin are specific to finance-related properties and some features of Tamarin are not used. It is

interesting to further extend Tamarin in the future.

10 CONCLUSION

We propose and implement FASVERIF, which can automatically generate finance-related properties and the corresponding models for smart contracts, and verify the properties automatically. FASVERIF outperforms other automatic tools in detecting finance-related vulnerabilities in accuracy and coverage of types of vulnerabilities, and it finds 13 contracts with exploitable bugs, including 10 contracts evading the detection of other automated tools to the best of our knowledge.

Acknowledgments

We would like to thank our shepherd and reviewers for their helpful comments. This research is supported by the National Key R&D Program of China 2021QY2104, National Natural Science Foundation of China under Grant No.61972369, No.62102385, No.62272434, Anhui Province Natural Science Foundation under Grant No.2108085QF262, and the Fundamental Research Funds for the Central Universities No. WK2150110024.

References

[1] The DAO hack. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/, 2016.

[2] Ethereum smart contract best practices. https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/timestamp-dependence/, 2016.

[3] Blockchain is empowering the future of insurance. https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/, 2017.

[4] The Parity bug. http://hackxingdistributed.com/2017/07/22/deep-dive-parity-bug, 2017.

[5] A Closer Look at ICO Smart Contracts. https://tokeny.com/a-closer-look-at-ico-smart-contracts/, 2018.

[6] New multioverflow bug identified in multiple erc20 smart contracts. https://peckshield.medium.com/new-multioverflow-bug-identified-in-multiple-erc20-smart-contracts-cve-2018-10706-8e55946c252c, 2018.

[7] Fatal transfermint bug in multiple trc20 smart contracts. https://twitter.com/peckshield/status/1115226918855401479?cxt=HHwWjoCj5aq-ivoeAAAA, 2019.

[8] fuzzywuzzy 0.18.0. https://pypi.org/project/fuzzywuzzy/, 2020.

[9] Swc-107: Reentrancy. https://swcregistry.io/docs/SWC-107, 2020.

[10] Swc registry. https://swcregistry.io/, 2020.

[11] Erc20 standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md, 2021.

[12] Main repository for smartpulse. https://github.com/utopia-group/SmartPulseTool, 2021.

[13] The Poly Network attack. https://en.wikipedia.org/wiki/Poly_Network_exploit, 2021.

[14] Aicoin — information about ethereum. https://www.aicoin.com/currencies/ethereum.html?lang=en, 2022.

[15] Blockscan chat. https://chat.blockscan.com/start, 2022.

[16] Bug bounty programs. https://consensys.github.io/smart-contract-best-practices/bug-bounty-programs/, 2022.

[17] difflib — helpers for computing deltas. https://docs.python.org/3/library/difflib.html, 2022.

[18] Erc721 standard. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md, 2022.

[19] Etherscan. https://etherscan.io/, 2022.

[20] Fasverif-dataset. https://github.com/secwisf/FASVERIF-dataset/tree/master, 2022.

[21] Fasverif online. https://101.200.87.174/, 2022.

[22] go-ethereum. https://github.com/ethereum/go-ethereum/blob/4e474c74dc2ac1d26b339c32064d0bac98775e77/consensus/ethash/consensus.go, 2022.

[23] Manticore. https://github.com/trailofbits/manticore/, 2022.

[24] Solidity documentation. https://solidity.readthedocs.io/en/latest, 2022.

[25] Source codes of evm. https://github.com/ethereum/go-ethereum/tree/master/core/vm/evm.go, 2022.

[26] Z3: An efficient smt solver. https://www.microsoft.com/en-us/research/project/z3-3/, 2022.

[27] The complete version of our paper. http://arxiv.org/abs/2208.12960, 2023.

[28] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 53–70. USENIX Association, 2016.

[29] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 215–228. ACM, 2020.

[30] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. *CoRR*, abs/2109.04347, 2021.

[31] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.

[32] Jan Baumeister, Norine Coenen, Borzoo Bonakdarpour, Bernd Finkbeiner, and César Sánchez. A temporal logic for asynchronous hyperproperties. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 694–717. Springer, 2021.

[33] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.*, 53(3):67:1–67:43, 2020.

[34] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[35] ConsenSys. Mythril. https://github.com/ConsenSys/mythril, 2022.

[36] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47, 587 ethereum smart contracts. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 530–541. ACM, 2020.

[37] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 8–15. IEEE / ACM, 2019.

[38] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 259–269. ACM, 2018.

[39] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanán, Yang Liu, and Jun Sun. Semantic understanding of smart contracts: Executable operational semantics of solidity. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1695–1712. IEEE, 2020.

[40] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 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*. The Internet Society, 2018.

[41] Harith Kamarul. Ethereum in 2020: the view from the block explorer. https://medium.com/etherscan-blog/ethereum-in-2020-the-view-from-the-block-explorer-2f9a1db2ee15, 2020.

[42] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In Dongmei Zhang and Anders Møller, editors, *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 363–373. ACM, 2019.

[43] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1317–1333. USENIX Association, 2018.

[44] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher

Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 254–269. ACM, 2016.

[45] Simon Meier. *Advancing automated security protocol verification*. PhD thesis, ETH Zurich, Zürich, Switzerland, 2013.

[46] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.

[47] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon J. Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Trans. Priv. Secur.*, 22(2):14:1–14:34, 2019.

[48] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin T. Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1661–1677. IEEE, 2020.

[49] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[50] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010.

[51] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[52] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 621–640. ACM, 2020.

[53] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. VERISMART: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1678–1694. IEEE, 2020.

[54] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu Lahiri, and Isil Dillig. Smartpulse: Automated checking of temporal properties in smart contracts. In *42nd IEEE Symposium on Security and Privacy*. IEEE, May 2021.

[55] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In Roberto Tonelli, Giuseppe Destefanis, Steve Counsell, and Michele Marchesi, editors, *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*, pages 9–16. ACM, 2018.

[56] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*, pages 9–16. ACM, 2018.

[57] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *CoRR*, abs/2008.02712, 2020.

[58] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 664–676. ACM, 2018.

[59] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 664–676. ACM, 2018.

[60] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the*

2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018, pages 67–82. ACM, 2018.

[61] Shunfan Zhou, Zhemin Yang, Jie Xiang, Yinzhi Cao, Min Yang, and Yuan Zhang. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2793–2810. USENIX Association, 2020.