SmartIFSyn: Automated Information Flow Security Policy Synthesis for Smart Contracts

ANONYMOUS AUTHOR(S)
SUBMISSION ID: #514

Smart contracts have achieved significant success, however, their security remains a long-standing challenge. The immutability and transparency of smart contracts require establishing a strong mechanism to prevent private leakage and trusted data tampering. Apart from traditional logic and code-level vulnerabilities arising from insufficient control over contract variables and function parameters, smart contracts may store private-dependent information in blockchain records, which is a critical type of vulnerability, but often overlooked in existing security analysis. In this paper, we present an automated approach for synthesizing security policies, named SmartIFSyn, to eliminate information flow vulnerabilities in smart contracts. We formalize the semantics of Solidity, the most widely used smart contract language, and analyze information flow security of Solidity smart contracts from two perspectives: local-variable security and global-interaction security. We present a type system to guide the elimination of local-variable vulnerabilities by inferring a policy and resort to constraint solving to synthesize a desired policy in case that the type system fails. The policy ensures both local-variable and global-interaction security while it is maximally aligned with user preference. Furthermore, the policy can be subsequently converted into enforceable specifications. We implement our approach in a tool and evaluate it on 17,160 real-world Ethereum smart contracts. The experimental results demonstrate the efficacy of our approach, e.g., detected 243 vulnerabilities in 223 real-world Ethereum smart contracts.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Program analysis**; • **Security and privacy → Formal security models**; **Logic and verification**.

Additional Key Words and Phrases: Smart contracts, Solidity, Information flow security, Formal semantics, Type inference, Constraint solving

ACM Reference Format:
Anonymous Author(s). 2026. SmartIFSyn: Automated Information Flow Security Policy Synthesis for Smart Contracts. In *Proceedings of The ACM International Conference on the Foundations of Software Engineering submission (FSE'26).* ACM, New York, NY, USA, 22 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Smart contracts are self-executing programs that run on a blockchain platform (e.g., Ethereum [85]) to automate the actions required in trusted transactions without the need for a central authority [78]. Due to their immutability and transparency, smart contracts have been widely adopted in, e.g., games [6], outsourced computation [57], and financial services [84]. However, these characteristics also pose critical security challenges. On one hand, immutability means that it is difficult to modify smart contracts after deployment, thus any vulnerabilities in a deployed contract could lead to an irreversible damage to the blockchain. For instance, in June 2016, Ethereum suffered the infamous DAO attack due to a reentrancy vulnerability, resulting in a loss of approximately 3.6 million Ethers and forcing a hard fork of the Ethereum blockchain [62]. On the other hand, transparency

FSE'26, Montreal, Canada

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

means that all the information in the blockchain is publicly accessible which could be exploited by attackers to infer private data or tamper trusted data. For example, the Parity wallet attack freezes numerous wallets by tampering trusted data, causing a loss of millions of dollars [12].

While a number of security analysis have been studies for smart contracts [3, 12, 14, 16, 24, 32, 40, 41, 44, 45, 47, 52, 56, 64, 68, 70, 86, 87], information flow security (IFS [80]) is often overlooked or less-explored. IFS could be used to enforce both confidentiality [75] and integrity [8], where confidentiality ensures that private information can only be accessed by authorized users, while integrity ensures that trusted data remains unaltered by any unauthorized modifications.

While there are some studies on IFS analysis for smart contracts [12, 14, 24, 32, 41, 87], they suffer from the following three limitations. (1) They exclusively focus on integrity while neglecting confidentiality. However, there is a risk of privacy leakage, because smart contracts, transactions and logs are stored in publicly accessible blockchains. For example, the Smart Contract Weakness Classification (SWC) Registry [18] highlights that attackers can exploit transaction data to extract private information from blockchains. (2) They focus on integrity over state variables and function parameters (called local-variable security hereafter), but fail to comprehensively capture global-interaction security arising from contract interactions with external entities. This limitation primarily stems from the lack of comprehensive modeling of the contract language associated with the blockchain during the analysis. (3) They primarily focus on detection rather than elimination of vulnerabilities, while manually eliminating vulnerabilities is labor-consuming and error-prone.

In this work, we propose a novel approach for synthesizing IFS policies for smart contracts, named SmartIFSyn (Smart contract Information Flow security policy Synthesis). Specifically, we formalize the semantics of Solidity [22] that covers more language constructs than previous works and incorporates blockchain records into the semantics for the first time. Based on the semantics, we define a policy as a mapping from variables to security levels and define two security types: *local-variable security* for securing control and data flows, and *global-interaction security* for securing interactions with external entities. We then design a type system to detect and eliminate potential vulnerabilities by inferring a policy. While the type system is sound w.r.t. the local-variable security, it is *not* sound w.r.t. the global-interaction security due to interactions with external entities (e.g., transaction-related operations). Thus, this policy is adopted as the final IFS policy *only* if it also ensures the global-interaction security, hence the contract's IFS. Otherwise, we resort to constraint solving (i.e., MaxSAT [9]) to synthesize a desired IFS policy, ensuring both local-variable security and global-interaction security. Finally, the IFS policy is transformed into enforceable specifications by dynamically tracking security levels and adding `require` statements in the contract ensuring that the contract satisfies both local-variable security and global-interaction security.

We implement our approach in a tool and evaluate SmartIFSyn using 17,160 real-world Ethereum smart contracts. The results showcase the performance of our approach in detecting and eliminating information flow vulnerabilities for both confidentiality and integrity. In particular, SmartIFSyn generally outperforms promising tools (AChecker [32], Ethainter [12], Securify [82], SoMo [24], and STC/STV [41]) and detected 243 vulnerabilities in 223 real-world Ethereum smart contracts.

In summary, our main contributions are:

- We formulate the IFS of smart contracts from two perspectives: local-variable security and global-interaction security, covering both confidentiality and integrity.
- We design a novel IFS policy synthesis approach that integrates type inference and constraint solving, and enforce synthesized policies using program specifications.
- We implement our approach in a tool SmartIFSyn and evaluate it on 17,160 real-world smart contracts, demonstrating the practical efficacy in automatically detecting and eliminating information flow vulnerabilities.

Expressions:	e	::=	$\epsilon \mid n \mid id \mid g \mid \mathbf{op_u}\ e \mid e_1\ \mathbf{op_b}\ e_2 \mid e_1[e_2] \mid e_1.e_2$
Statements:	p	::=	$\texttt{skip} \mid p; p \mid e_1 := e_2 \mid \texttt{while}(e)\ p \mid \texttt{if}(e)\ \texttt{then}\ p_1\ \texttt{else}\ p_2 \mid \texttt{assert}(e)\ \texttt{in}\ p$
		|	$\texttt{require}(e_1, e_2)\ \texttt{in}\ p \mid e_0 := id_c.id_f(e^*) \mid e_0 := id_c.\ \texttt{call}(id_f, e^*)$
		|	$e_0 := id_c.\ \texttt{delegatecall}(id_f, e^*) \mid e_0 := id_c.\ \texttt{staticcall}(id_f, e^*) \mid \texttt{emit}\ id_{\text{evt}}(e^*)$
		|	$\texttt{letvar}\ id := e\ \texttt{in}\ p \mid e_1.\ \texttt{push}(e_2) \mid id_c.\ \texttt{transfer}(id) \mid \texttt{selfdestruct}(id_c)$
		|	$\texttt{revert}\ id_{\text{err}}(e^*) \mid e_0 := \texttt{new}\ id_c(e^*)$
Functions:	f	::=	$id_c.id_f(id^*)\ V\ M\ id_{m,1}(e_1^*)\ \cdots\ id_{m,n}(e_n^*)\ \texttt{returns}(id_{\text{ret}})\ \{\ p; \texttt{return}\ e_{\text{ret}}\ \}$
Modifiers:	m	::=	$id_c.id_m(id^*)\ \{\ p_1; \_; p_2\ \}$
Contracts:	ctr	::=	$id_c\ \{\ id^* \mid m^* \mid f^+\ \}$

Fig. 1. Syntax of Solidity.

Outline. Section 2 introduces Solidity. Section 3 provides motivating examples and an overview of our approach. Section 4 details our approach. Section 5 reports the experimental results. Section 6 discusses the related work. We conclude the paper in Section 7.

2 Preliminaries

2.1 The Solidity Language

Syntax. Fig. 1 presents the syntax of the Solidity language according to the official Solidity language [22], where optional constructs may be explicitly present for the sake of simplicity.

A smart contract ctr comprises a contract identifier id_c and a series of declarations of state variables (id^*), modifiers (m^*) and functions (f^+) one of which is the contract's constructor.

A modifier m, defined as $id_c.id_m(id^*)\ \{\ p_1; \_; p_2\ \}$, can be used to change the behavior of functions in a declarative way, where id_c is a contract identifier, id_m is a modifier identifier, id^* is a list of modifier parameters, _; is a placeholder where the body of the function being modified should be placed, p_1 and p_2 are respectively the pre-statement (e.g., pre-condition) and post-statement (e.g., post-condition) of the function body.

A function f comprises a contract identifier id_c, a function identifier id_f, input parameters id^*, two decorators (V, M), a list of modifiers $id_{m,i}(e_i^*)$, return variable id_{ret}, and statement p, followed by a `return` statement. The decorator $V \in \{$`public`, `private`, `external`, `internal`$\}$ specifies the visibility of the function, and the decorator $M \in \{$`pure`, `view`, `payable`, `non-payable`$\}$ specifies the state mutability of the function. A `pure` function neither reads nor modifies the blockchain state; a `view` function reads but does not modify the blockchain state; a `payable` function can read and modify the blockchain state and execute transactions. A function is `non-payable` in default if no other mutability is specified, which is the same as `payable` except that it cannot receive Ether.

Statements p include common statements (i.e., `skip`, sequential statements, assignments, `while` and `if-then-else`), `assert`/`require` statements, standard function calls, low-level function calls (i.e., `call`, `delegatecall`, `staticcall`), event emit statements `emit`, `letvar` statements for defining local variables, and other built-in atomic statements (i.e., `push`, `transfer`, `selfdestruct`, `revert`, and `new`). Standard function calls can be inter-contract and intra-contract for which the contract identifier id_c is often dropped. Low-level function calls are similar to inter-contract function calls but are not type checked and interface validated.

Expressions e include empty expression ϵ, constants n, identifiers id, Solidity predefined variables g, arithmetic operations (unary operation $\mathbf{op_u}\ e$ and binary operation $e_1\ \mathbf{op_b}\ e_2$), array/mapping entry accesses $e_1[e_2]$, struct/enum field accesses $e_1.e_2$. The expression ϵ is only used for optional expressions, e.g., `require`(e, ϵ); Solidity predefined variables g provide key information about the contract and execution environment, e.g, `msg.sender` can be the address of the external user.

Semantics. Fix a smart contract ctr. Let $\mathbb{E}$ denote the set of all the expressions and $\mathbb{X} \subseteq \mathbb{E}$ denote the set of all the assignable expressions (i.e., lvalues) in ctr, including state variables, struct/enum fields, array/mapping entries, and their compositions (e.g., x, $x.y$, $a[b]$). Let $\mathbb{V}$ denote the set of possible

$$\frac{}{\langle s, \texttt{skip}, r\rangle \Rightarrow \langle s, r\rangle}\ \textsc{Skip} \qquad \frac{s' = s[e_1 \mapsto s(e_2)]}{\langle s, e_1 := e_2, r\rangle \Rightarrow \langle s', r\rangle}\ \textsc{Asg} \qquad \frac{s' = s\,[e_1 \mapsto s(e_1) \cdot s(e_2)]}{\langle s, e_1.\texttt{push}(e_2), r\rangle \Rightarrow \langle s', r\rangle}\ \textsc{Push}$$

$$\frac{s(e_1) = \textbf{true} \quad \langle s, p, r\rangle \Rightarrow \langle s', r'\rangle}{\langle s, \texttt{require}(e_1, e_2)\ \texttt{in}\ p, r\rangle \Rightarrow \langle s', r'\rangle}\ \textsc{Req}_T \qquad \frac{s(e) = \textbf{false}}{\langle s, \texttt{while}(e)\ p, r\rangle \Rightarrow \langle s, r\rangle}\ \textsc{Whl}_F \qquad \frac{r' = \texttt{update}(s, r, id_{\text{evt}}, e^*)}{\langle s, \texttt{emit}\ id_{\text{evt}}(e^*), r\rangle \Rightarrow \langle s, r'\rangle}\ \textsc{Evt}$$

$$\frac{s(e) = \textbf{true} \quad \langle s, p, r\rangle \Rightarrow \langle s', r'\rangle}{\langle s, \texttt{assert}(e)\ \texttt{in}\ p, r\rangle \Rightarrow \langle s', r'\rangle}\ \textsc{Asr}_T \qquad \frac{s(e) = \textbf{false}}{\langle s, \texttt{assert}(e)\ \texttt{in}\ p, r\rangle \Rightarrow \langle err, r\rangle}\ \textsc{Asr}_F \qquad \frac{r' = \texttt{update}(s, r, id_{\text{err}}, e^*)}{\langle s, \texttt{revert}\ id_{\text{err}}(e^*), r\rangle \Rightarrow \langle err, r'\rangle}\ \textsc{Rev}$$

$$\frac{s(e_1) = \textbf{false} \quad r' = \texttt{update}(s, r, e_2)}{\langle s, \texttt{require}(e_1, e_2)\ \texttt{in}\ p, r\rangle \Rightarrow \langle err, r'\rangle}\ \textsc{Req}_F \qquad \frac{s(e) = \textbf{true} \quad \langle s, p, r\rangle \Rightarrow \langle s', r'\rangle \quad \langle s', \texttt{while}(e)\ p, r'\rangle \Rightarrow \langle s'', r''\rangle}{\langle s, \texttt{while}(e)\ p, r\rangle \Rightarrow \langle s'', r''\rangle}\ \textsc{Whl}_T$$

$$\frac{r' = \texttt{update}(s, r, id_t, id_c, id_B)}{\langle s, \texttt{selfdestruct}(id_c), r\rangle \Rightarrow \langle exit, r'\rangle}\ \textsc{Dst} \qquad \frac{r' = \texttt{update}(s, r, id_t, id_c, id_f, e^*) \quad \langle s_c, e_c := id_c.id_f(e^*), r_c\rangle \Rightarrow \langle s_c', r_c'\rangle}{\langle s, e_0 := id_c.\texttt{call}(id_f, e^*), r\rangle \Rightarrow \langle s[e_0 \mapsto s_c'(e_c)], r'\rangle}\ \textsc{Call}$$

$$\frac{s(e) = \textbf{true} \quad \langle s, p_1, r\rangle \Rightarrow \langle s', r'\rangle}{\langle s, \texttt{if}(e)\ \texttt{then}\ p_1\ \texttt{else}\ p_2, r\rangle \Rightarrow \langle s', r'\rangle}\ \textsc{If}_T \qquad \frac{r' = \texttt{update}(s, r, id_t, id_c, id_f, e^*) \quad \langle s_c, e_c := id_c.id_f(e^*), r_c\rangle \Rightarrow \langle s_c, r_c\rangle}{\langle s, e_0 := id_c.\texttt{staticcall}(id_f, e^*), r\rangle \Rightarrow \langle s[e_0 \mapsto s_c(e_c)], r'\rangle}\ \textsc{SCall}$$

$$\frac{s(e) = \textbf{false} \quad \langle s, p_2, r\rangle \Rightarrow \langle s', r'\rangle}{\langle s, \texttt{if}(e)\ \texttt{then}\ p_1\ \texttt{else}\ p_2, r\rangle \Rightarrow \langle s', r'\rangle}\ \textsc{If}_F \qquad \frac{r' = \texttt{update}(s, r, id_t, id_c, id_f, e^*) \quad \langle s, e' := id_c.id_f(e^*), r'\rangle \Rightarrow \langle s', r''\rangle}{\langle s, e_0 := id_c.\texttt{delegatecall}(id_f, e^*), r\rangle \Rightarrow \langle s'[e_0 \mapsto s'(e')], r''\rangle}\ \textsc{DCall}$$

$$\frac{r' = \texttt{update}(s, r, id_t, id_c, id) \quad s' = s[id_B \mapsto s(id_B) - s(id)]}{\langle s, id_c.\texttt{transfer}(id), r\rangle \Rightarrow \langle s', r'\rangle}\ \textsc{Trf} \qquad \frac{e^* = e_1, \cdots, e_n \quad r' = r \cdot ((e_1, s(e_1)), \cdots, (e_n, s(e_n)))}{\texttt{update}(s, r, e^*) : r'}\ \textsc{Upd}$$

$$\frac{\langle s, p_1, r\rangle \Rightarrow \langle s', r'\rangle \quad \langle s', p_2, r'\rangle \Rightarrow \langle s'', r''\rangle}{\langle s, p_1; p_2, r\rangle \Rightarrow \langle s'', r''\rangle}\ \textsc{Seq} \qquad \frac{\langle s[id \mapsto s(e)], p, r\rangle \Rightarrow \langle s_1, r'\rangle \quad s' = s_1[id \mapsto s(id)]}{\langle s, \texttt{letvar}\ id := e\ \texttt{in}\ p, r\rangle \Rightarrow \langle s', r'\rangle}\ \textsc{LetVar}$$

$$\frac{\langle s_c, e_{\text{new}} := id_c.id_{\text{con}}(e^*), r_c\rangle \Rightarrow \langle s_c', r_c'\rangle \quad s' = [e_0 \mapsto s_c'(e_{\text{new}})] \quad r' = \texttt{update}(s', r, id_t, e_0, e^*)}{\langle s, e_0 := \texttt{new}\ id_c(e^*), r\rangle \Rightarrow \langle s', r'\rangle}\ \textsc{New}$$

$$id_c.id_f(id^*)\ V\ M\ id_{m,1}(e_1^*)\ \cdots\ id_{m,n}(e_n^*)\ \text{returns}(id_{\text{ret}})\ \{\ p; \texttt{return}\ e_{\text{ret}}\ \} \quad \forall_{1 \le i \le n}.\ id_c.id_{m,i}(e_i^*)\ \{\ p_{1,i};\ \_;\ p_{2,i}\}$$
$$id^* = id_1, \cdots, id_n \quad e^* = e_1, \cdots, e_n \quad s_0 = s[id_1 \mapsto s(e_1), \cdots, id_n \mapsto s(e_n)]$$
$$r_{\text{user}} = (\text{external call} \wedge M \in \{\texttt{payable}, \texttt{non-payable}\})\ ?\ \texttt{update}(s_0, \langle\rangle, g_{\text{adr}}, id_c, id_f, g_{val}, e^*))\ :\ \langle\rangle$$
$$r_0 = r \cdot r_{\text{user}} \quad p_1' = p_{1,1}; \cdots; p_{1,n}; p \quad p_2' = p_{2,n}; \cdots; p_{2,1}$$
$$\frac{\langle s_0, p_1', r_0\rangle \Rightarrow \langle s_1, r_1\rangle \quad \langle s_1[id_{\text{ret}} \mapsto s_1(e_{\text{ret}})], p_2', r_1\rangle \Rightarrow \langle s_2, r_2\rangle \quad s_2' = s_2[e_0 \mapsto s_2(id_{\text{ret}}), id_1 \mapsto s(id_1), \cdots, id_n \mapsto s(id_n)]}{\langle s, e_0 := id_c.id_f(e^*), r\rangle \Rightarrow \langle s_2', r_2\rangle}\ \textsc{Fun}$$

Fig. 2. Semantics of Solidity, where $\cdot$ denotes array/sequence concatenation, id_t denotes current contract's address; id_c denotes callee's contract address; id_B denotes current contract's balance; id_{con} denotes the constructor of the callee's contract; id_{evt} and id_{err} denote the event identifier and error identifier, respectively; e_{new} denotes the address of the newly created contract; e_c denotes the return value of a standard function call in the context of the callee's contract; s_c and r_c denote the state and record of the callee's contract, respectively; g_{adr} and g_{val} denote the address and amount from external user, respectively.

right-values (rvalues). A *state* $s : \mathbb{X} \to \mathbb{V}$ is a mapping from lvalues $x \in \mathbb{X}$ to rvalues $s(x) \in \mathbb{V}$. The evaluation $s(e)$ of an expression e in a state s is defined as usual (i.e., $s(n) = n$, $s(\mathbf{op_u}\ e) = \mathbf{op_u}\ s(e)$, $s(e_1\ \mathbf{op_b}\ e_2) = s(e_1)\ \mathbf{op_b}\ s(e_2)$), particularly $s(\epsilon) = \bot$ meaning that the rvalue of ϵ is undefined. The update of a state s is written as $s[e \mapsto v]$, namely, the lvalue of e is updated to the rvalue v while the lvalues of others remain the same. We define *err* as an exceptional state that causes a rollback of the current transaction. This includes violations of `assert`/`require` conditions, explicit rollbacks triggered by the `revert` statement and out-of-gas exceptions [13, 31, 36, 58]. Similarly, we define *exit* as an exit state that represents a normal termination. A *record* r is defined as a finite sequence of expression-value pairs $\langle (e_1, v_1), \cdots, (e_n, v_n)\rangle$, where $e_i \in \mathbb{E}$ is an expression and v_i is its rvalue. A *configuration* c is a triple $\langle s, p, r\rangle$ consisting of a state s, a statement p to be executed and a record r.

The (big-step operational) semantics is defined as a judgment $\langle s, p, r\rangle \Rightarrow \langle s', r'\rangle$, namely, the execution of p under the state s and the record r results in the state s' and the record r'. Fig. 2 shows the semantics of Solidity, where `update` is an auxiliary function, introduced to simplify the description of blockchain record updates. Most of them are consistent with those in other common high-level languages (i.e., `skip`, sequential statements, assignments, `while` statements, `if-then-else` statements, `assert` statements, `letvar` statements, `push` statements). Below, we explain Solidity-specific statements which involve external interactions.

$\textsc{Req}_F$ states that when the condition e_1 of `require`(e_1, e_2) in p is false, the execution aborts with a rollback, transitioning the system to an error state *err*, where the error message e_2 is recorded in the blockchain when it is not ϵ. Otherwise, p is executed as usual (i.e., $\textsc{Req}_T$). $\textsc{Dst}$ states that when

`selfdestruct` is invoked to destroy a contract, the current contract's address id_t, the recipient's address id_c, and the remaining Ether balance id_B are recorded, and the execution enters the *exit* state. Trf states that when a `transfer` statement is executed, a specified amount of Ether id is transferred to the target contract id_c, and the execution records the sender's and recipient's addresses (id_t and id_c), along with the transferred amount id. Evt states that when executing an event emit statement `emit`, the log information e^* and the event id_{evt} are recorded. Rev for an error revert statement `revert` is defined similarly except that the execution enters the *err* state. New states that executing a `new` statement creates a new contract with address e_{new} by invoking its constructor id_{con}, moreover, the caller's contract address id_t, the new contract address e_{new} and the parameters e^* are recorded in the blockchain.

Fun states that the execution of a standard function call first executes the pre-statements $p_{1,1}; \cdots ; p_{1,n}$ given in the modifiers, then the function body p; `return` e_{ret}, and finally the post-statements $p_{2,n}; \cdots ; p_{2,1}$ given in the modifiers. We note that the return value is assigned to an return variable id_{ret} and returned after executing the post-statements. The record is updated according to the decorators V and M. If it is an external call to a `public`/`external` function with `payable`/`non-payable` state mutability, the address g_{adr} and amount g_{val} from the external user, the address of callee's contract id_c, the function id_f, and its parameters e^* are recorded. Intuitively, such a function call is treated as a transaction, thus all the information is recorded on the blockchain. Call and DCall are similar to Fun except that the current contract's address id_t, callee's contract address id_c, function id_f and parameters e^* are recorded. We note that the callee is executed within the context (s_c and r_c) of the callee's contract for `call` and within the context (s and r') of the caller's contract for `delegatecall`. SCall is similar to Call except that it is restricted to read-only operations to the callee's contract, i.e., s_c and r_c remain the same.

In this work, we assume all smart contract executions terminate, either normally or by entering the error state *err*. Moreover, for simplicity, array bounds are not explicitly checked in our semantics; instead, we assume they are verified using `assert` statements. As a result, executions halt in an error state when indices fall outside the array's range. Finally, we assume that local variables are uniquely identified by their function signatures and names, and global variables by their names.

2.2 Information Flow Security of Smart Contracts

We define Information Flow Security (IFS) of smart contracts from two perspectives: local-variable security and global-interaction security, based on the security lattice and policy.

We fix a security lattice $\mathbb{L} = \{\mathrm{L}, \mathrm{H}\}$ with $\mathrm{L} \sqsubseteq \mathrm{L}$, $\mathrm{L} \sqsubseteq \mathrm{H}$, $\mathrm{H} \sqsubseteq \mathrm{H}$, and $\mathrm{H} \not\sqsubseteq \mathrm{L}$, and denote by the least upper bound of two security levels τ_1 and τ_2 by $\tau_1 \sqcup \tau_2$, where $\tau_1, \tau_2 \in \mathbb{L}$. Specifically, $\tau \sqcup \mathrm{H} = \mathrm{H} \sqcup \tau = \mathrm{H}$ for $\tau \in \mathbb{L}$, and $\mathrm{L} \sqcup \mathrm{L} = \mathrm{L}$. For confidentiality, public variables should be annotated by L while secret variables should be annotated by H. For integrity, trusted variables should be annotated by L while untrusted variables should be annotated by H. A *policy* $\sigma : \mathbb{U} \to \mathbb{L}$ is a function that assigns to each $u \in \mathbb{U}$, a security level $\tau \in \mathbb{L}$, where $\mathbb{U} = \mathbb{X} \cup \mathbb{P}$, and $\mathbb{P} \subseteq \mathbb{E}$ denotes the set of all Solidity predefined variables and function parameters. Roughly speaking, a desired policy should forbid flowing from H-level data to L-level data.

Local-variable security. Local-variable security aims to ensure the security of control and data flows within a contract, such as preventing the leakage of private data to public variables via assignments. Given a set $U \subseteq \mathbb{U}$, we denote by U^{σ}_{L} (resp. U^{σ}_{H}) the subset of U that have the security level L (resp. H) under a policy σ. Two states s and s' (resp. configurations $\langle s, p, r\rangle$ and $\langle s', p', r'\rangle$) are said to be *$U$-equivalent*, written as $s \simeq_U s'$ (resp. $\langle s, p, r\rangle \simeq_U \langle s', p', r'\rangle$), if for every $u \in U$, $s(u) = s'(u)$. The input variable set of a function, $\mathbb{U}^{in} \subseteq \mathbb{U}$, consists of three types of variables: function parameters, state variables, and Solidity predefined variables, where the latter two types

are included because they may vary with function calls as well. For clarity, each struct, enum, array, or mapping is uniformly annotated a security level, either L or H, indicating that all contained elements share the same security level.

Definition 1 (Local-variable security)**.** Given a smart contract ctr, for every function f of the contract ctr with the body statement p and input variable set $\mathbb{U}^{in}$, f is *local-variable secure* under a policy σ, if for any pair of executions of p: $c_1 \Rightarrow \langle s_1, r_1\rangle$ and $c_2 \Rightarrow \langle s_2, r_2\rangle$, we have:

$$c_1 \simeq_{\mathbb{U}_{\mathrm{L}}^{in,\sigma}} c_2 \implies s_1 \simeq_{\mathbb{X}_{\mathrm{L}}^{\sigma}} s_2.$$

The contract ctr is *local-variable secure* under a policy σ if every function in ctr is local-variable secure under σ. In this case, σ is referred to as a *local-variable security policy* (denoted σ_{loc}).

Global-interaction security. Global-interaction security focuses the security during interactions with external entities, such as ensuring trusted transactions and low-level function calls.

Definition 2 (Global-interaction leakage model)**.** Given a configuration $c = \langle s, p, r\rangle$ with $s \neq err$, and an execution $\rho : \langle s, p, r\rangle \Rightarrow \langle s', r'\rangle$, let $\Delta r = r' - r$ denotes the sequence of new pairs added to the record r after executing the statement p, we define the observation $O(\rho)$ inductively as follows:

(1) if p is a loop statement `while`(e) p' and the loop condition e depends on a dynamic data structure (e.g., an array or a mapping), then $O(\rho)$ is $\langle(e, s(e))\rangle$ if $s(e)$ is false (note that in this case Δr is empty), otherwise $\langle(e, s(e))\rangle \cdot O(\rho')$, where $\rho' : \langle s, p'; p, r\rangle \Rightarrow \langle s', r'\rangle$;
(2) if p is an event emit statement `emit`(e^*) or an error revert statement `revert`(e^*), then $O(\rho) = \Delta r$ for confidentiality and $O(\rho) = \langle\rangle$ (i.e., empty observation) for integrity;
(3) if p is a `require` statement `require`(e_1, e_2) in p, then for integrity, $O(\rho) = \langle\rangle$ if $s(e_1)$ is false, otherwise $O(\rho) = \Delta r$; for confidentiality, $O(\rho) = \Delta r$;
(4) if p is a standard function call $id_c.id_f(e^*)$, then $O(\rho) = \Delta r$ for confidentiality, and $O(\rho) = \Delta r - r_{\text{user}}$ for integrity, where r_{user} is the record of an invocation by an external user (cf. Fig. 2);
(5) if p is a sequential statement $p_1; p_2$ with executions $\rho_1 : \langle s, p_1, r\rangle \Rightarrow \langle s_1, r_1\rangle$ and $\rho_2 : \langle s_1, p_2, r_1\rangle \Rightarrow \langle s_2, r_2\rangle$, then $O(\rho) = O(\rho_1) \cdot O(\rho_2)$, i.e., the concatenation of the observations $O(\rho_1)$ and $O(\rho_2)$;
(6) otherwise, $O(\rho) = \Delta r$, i.e., the newly added interaction records.

Two executions ρ_1 and ρ_2 of a statement p are *indistinguishable* w.r.t. the leakage model O if $O(\rho_1) = O(\rho_2)$. Intuitively, if a loop condition e depends on a dynamic data structure, the number of iterations must remain consistent to avoid out-of-gas exceptions caused by unbounded loops [31, 36]. Because when such conditions are influenced by sensitive information or adversary-controlled variables, an adversary may infer confidential data or disrupt contract execution by observing if an out-of-gas execution occurs or not. For records generated by `emit` and `revert`, error messages from `require`, and transaction record related to standard function calls from external users, strict equality is required to ensure confidentiality; however, integrity does not require equality since these records only reflect information output rather than data tampering. In contrast, remaining record-related statements (i.e., `transfer`, `new`, `selfdestruct`, low-level function calls) not only record data on-chain, which may cause leakage, but also influence the actual execution, potentially leading to unauthorized state updates. For instance, a `transfer` record includes information about both interacting parties. Differences in such records may reveal private data or indicate that the transaction was influenced by untrusted inputs. Therefore, to ensure both confidentiality and integrity, the recorded information of these statements must remain indistinguishable.

Definition 3 (Global-interaction security)**.** Given a smart contract ctr, for every function f of the contract ctr with body statement p and input variable set $\mathbb{U}^{in}$, f is *global-interaction secure* under a policy σ, if for any pair of executions of p: $\rho_1 : c_1 \Rightarrow \langle s_1, r_1\rangle$ and $\rho_2 : c_2 \Rightarrow \langle s_2, r_2\rangle$, we have:

$$c_1 \simeq_{\mathbb{U}_{\mathrm{L}}^{in,\sigma}} c_2 \implies O(\rho_1) = O(\rho_2)$$

```
1  contract WalletLibrary1 {
2    address public owner;
3    ...
4    function initWallet(address _owner) public {
5      // require(owner_security_level == H);
6      owner = _owner; // owner : L, _owner : H
7      ...
8    }
9    ...
10 }
11
12 contract WalletLibrary2 {
13   function kill() public {
14     // require(msg_sender_security_level == L);
15     selfdestruct(msg.sender); // msg.sender : H
16   }
17   ...
18 }
```

Fig. 3. Two attacks against the Parity wallet.

```
1  contract OddEven {
2    struct Player { address addr; uint number; }
3    Player[2] private players;
4    uint count = 0;
5    ...
6    // keyNumber : H
7    function play(uint keyNumber) public payable {
8      // require(keyNumber_security_level == L);
9      players[count] = Player(msg.sender, keyNumber);
10     count++;
11     if (count == 2) selectWinner();
12   }
13   function selectWinner() private {
14     uint n = players[0].number + players[1].number;
15     address winner = players[n % 2].addr;
16     ...
17   }
18 }
```

Fig. 4. SWC-136: unencrypted private data on-chain.

The contract *ctr* is *global-interaction secure* under σ if every function in *ctr* is global-interaction secure under σ. In this case, σ is referred to as a *global-interaction security policy* (denoted σ_{glb}).

Definition 4 (IFS). A smart contract *ctr* satisfies IFS under a policy σ, if σ is both σ_{loc} and σ_{glb}. In this case, σ is called an IFS policy, denoted σ_{IFS}.

3 Motivation and Approach Overview

3.1 Motivating Examples

Example 1. The two notorious Parity wallet attacks severely violated the integrity by maliciously tampering with or compromising trusted smart contract data [12]. As shown in Fig. 3, the first attack exploited a permission control vulnerability in the function `initWallet`, allowing the attacker to modify contract ownership (line 6) and consequently steal funds from the wallet. The second attack misused the `selfdestruct` function, successfully destroying the contract and causing funds to become frozen and non-transferable (line 15). The core cause of these two attacks lies in the incorrect setting of function visibility by the developers. Specifically, the `initWallet` and `kill` functions were mistakenly set as `public`, where the `initWallet` function allowed attackers to gain full control of the contract by supplying untrusted addresses, while the `kill` function allowed attackers to directly destroy the contract, resulting in unsafe fund transfers.

Example 2. SWC-136, classified as *Unencrypted Private Data On-Chain* in the SWC Registry [18], represents a privacy leakage scenario. A common misconception in smart contract development is that variables declared as `private` cannot be read. However, SWC-136 shows that an attacker can obtain privacy information by analyzing transaction records [4]. Fig. 4 illustrates this vulnerability using an *Odd or Even* game where players submit numbers, and the winner is determined based on if the sum of the two numbers is odd or even. In this contract, the `players` array stores the submitted numbers in plain text. Though the `players` array is marked as `private`, this visibility specifier only prevents access by other smart contracts—it does not stop anyone from viewing the transaction records on the blockchain. Consequently, the first player's number is exposed (i.e., the invocation of the function at line 7 is recorded on-chain), allowing the second player to choose a number that fully control the winner. Due to the transparency of the blockchain, relying solely on visibility specifiers like `private` or `internal` fails to prevent privacy from being leaked. Therefore, developers must ensure that sensitive data is encrypted if it is to be stored on-chain.

The contract `WalletLibrary1` violates local-variable security: the contract owner address `owner` (line 2) is trusted data (L), whereas the parameter `_owner` (line 4) can be supplied by any user and is therefore untrusted (H). The assignment at line 6 induces a data flow from H to L, causing

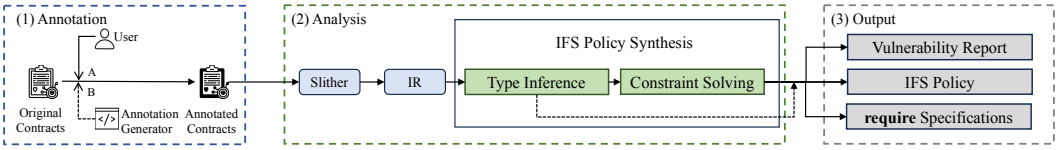

Fig. 5. Overview of our approach: A three-step pipeline for analyzing IFS in smart contracts.

`owner` to potentially take different values and thus violating local-variable security. Both contracts `WalletLibrary2` and `OddEven` violate global-interaction security. In the former, `msg.sender` (line 15) denotes the user's address and can take different values via the `public` function `kill`. Since its security level is H, according to rule Dst in Fig. 2, the observations of user addresses are different. In the latter, `keyNumber` (line 7) is a secret number provided by a player with security level H; according to rule Fun in Fig. 2, its values may be different in the blockchain records.

3.2 Approach Overview

The overview of our approach, named SmartIFSyn, is shown in Fig. 5. It uses a type system to detect potential information flow vulnerabilities in a given contract and infers a local-variable security policy to mitigate them. If the policy is not strong enough to be an IFS policy, SmartIFSyn resorts to constraint solving to synthesize a desired IFS policy, which is then enforced using program specifications. To achieve this, SmartIFSyn works in the following three main steps.

In the first step (1), security levels are assigned to variables by manual annotation (A), allowing users to provide domain-specific insights and enforce stricter security constraint that may not be inferred automatically. However, it is time-cost and error-prone. Thus, SmartIFSyn also employs an automated inference to assign security levels to user-unannotated input variables according to their visibility and types. For confidentiality, since all data on the blockchain is inherently public, we employ a heuristic, conservative strategy for assigning security levels. Specifically, prior studies [53, 72, 73] have shown that variable names often reflect their semantic intent; drawing on this insight, state variables whose names contain keywords such as *guess*, *key*, *password*, or *secret*, and whose visibility is `internal`/`private`, are annotated with H. Similarly, parameters of `external`/`public` functions are also annotated with H if their names contain any of the aforementioned keywords. All other state variables, function parameters and Solidity predefined variables are annotated with L. For integrity, constructor parameters, Solidity predefined variables within constructors, parameters of `internal`/`private` functions, and state variables are annotated as L. In contrast, parameters of `external`/`public` functions, Solidity predefined variables in non-constructor functions or related to blocks (e.g., `block.number`, `block.timestamp`) are annotated as H. Together, SmartIFSyn ensures that all input variables are annotated while balances user efforts with an automated inference.

In the second step (2), SmartIFSyn leverages Slither [25], a widely used open-source static analyzer for smart contracts, to transform the annotated contract into an intermediate representation (IR) in static single assignment (SSA) form. Next, we infer a local-variable security policy via type inference, aiming to eliminate vulnerabilities that violate the local-variable security. If the inferred policy also ensures the global-interaction security, it is directly adopted as the final IFS policy. However, type inference alone is insufficient to fully enforce IFS. For instance, certain operations such as transaction-related statements require information at a low-security level, but this requirement often imposes direct constraints on external inputs, which cannot be captured purely through type inference. To address this issue, we apply constraint solving (i.e., MaxSAT [9]) to derive an IFS policy that both guarantees IFS and remains as close as possible to the users' annotations.

In the third step (3), we enforce the IFS policy by dynamically tracking security levels and adding `require` statements into the contract to check both local-variable security and global-interaction

security. Our approach also generates a vulnerability report based on the analysis results, including the number, locations, and types of vulnerabilities.

4 Methodology

In this section, we first present the type system and then describe our constraint solving to synthesize a desired IFS policy.

4.1 Type System for Local-Variable Security

Type system. The typing judgment is in the form of $\mathrm{c} \vdash p : (\sigma, \psi) \Rightarrow (\sigma', \psi')$, where c is a security level of the current control flow, p is a statement under typing, σ is a policy and ψ is a constraint (initially set to **true**). The typing judgment states that given a security level of the current control flow c, after executing the statement p, the policy σ and constraint ψ are updated to σ' and ψ'. The initial security level of the control flow is set to L.

The type inference rules for Solidity are given in Fig. 6, where $\sigma(e)$ denotes the least upper bound of the security levels of all variables in the expression e, $\sigma_1 \sqcup \sigma_2$ is a policy such that for every $u \in \mathbb{U}$, $(\sigma_1 \sqcup \sigma_2)(u) = \sigma_1(u) \sqcup \sigma_2(u)$, and $\texttt{lfp}(\mathrm{c}, p, \sigma, \psi)$ is (σ, ψ) if $\sigma' = \sigma$, otherwise $\texttt{lfp}(\mathrm{c}, p, \sigma', \psi')$, where $\mathrm{c} \vdash p : (\sigma, \psi) \Rightarrow (\sigma', \psi')$. These rules are designed to prevent both control and data flows from security level H to L. In addition, we extract constraints by updating ψ. The auxiliary function `update` is introduced to infer a local-variable security policy and extracts constraints guided by the partial order $\sqsubseteq$ over security levels. The latter ensures that all record-related information is explicitly constrained to L, thereby preventing the adversary from distinguishing between executions.

T-Asg states that for an assignment $e_1 := e_2$, the security level of e_1 is inferred as the least upper bound of the control flow level c and the security level of e_2. The constraint enforces $\mathrm{c} \sqcup \sigma(e_2) \sqsubseteq \sigma(e_1)$ to prevent information flowing from H to L. T-Push states that when adding an element to a dynamic array, the security level of the resulting lvalue e_1 is given by the least upper bound of the security levels of the array e_1, the expression e_2, and the control flow level c. T-If propagates the least upper bound of c and $\sigma(e)$ to subsequent branches, preventing control flow induced vulnerabilities. In addition, the policies of the two branches (σ_1, σ_2) and the constraints (ψ_1, ψ_2) are merged at the next control point. T-Whl states that the policy should be the least fixed point (lfp) of the loop, as the loop may repeatedly update σ. Furthermore, the constraint ensures that if the loop condition e depends on a dynamic data structure, the least upper bound of c and $\sigma(e)$ must be L. T-Asr (resp. T-Req) states that when handling `assert` (resp. `require`) statement, the least upper bound of c and $\sigma(e)$ (resp. $\sigma(e_1)$) will be propagated to subsequent statements. In addition, for T-Req, when confidentiality is considered and e_2 is present, the least upper bound of c, $\sigma(e_1)$, and $\sigma(e_2)$ is required to be L. T-Letvar, similar to T-Asg, states that the security level of the newly declared local variable id is determined by $\mathrm{c} \sqcup e$.

T-Evt, T-Rev, T-Dst, and T-Trf primarily update the constraint ψ rather than the policy σ, as they involve interactions between the contract and external entities but do not directly depend on data flow. Specifically, the event emit statement `emit` and the error revert statement `revert` update ψ only when confidentiality is considered; the `transfer` statement updates ψ while also ensuring that the contract balance id_B remains secure after the transaction. The least upper bound of c and the security level of the record-related information is required to be L enforced by the constraint.

T-Call, T-SCall, and T-DCall state that the `call` and `staticcall` statements perform inference on the callee's policies and constraints (σ_c, ψ_c) under the callee's control flow security level c_c. Consequently, only the returned value e_c is considered when updating the caller's policy (i.e., the security level of e_0 is the least upper bound of c and the security level of e_c). In contrast, `delegatecall` performs inference within the caller's contract, using the control flow security level c, and updates the caller's policies. Furthermore, the least upper bound of c and the security level

$$\frac{}{\mathsf{c} \vdash \texttt{skip} : (\sigma, \psi) \Rightarrow (\sigma, \psi)}\ \text{T-Skip} \qquad \frac{\psi_1 = (\text{dynamic structure})\ ?\ \texttt{update}(\sigma, \psi, \mathsf{c}, e) : \psi \quad (\sigma', \psi') = \texttt{lfp}(\mathsf{c}, p, \sigma, \psi_1)}{\mathsf{c} \vdash \texttt{while}(e)\ p : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-Whl}$$

$$\frac{(\sigma', \psi') = \texttt{update}(\sigma, \psi, \mathsf{c}, (e_1, \sigma(e_2)))}{\mathsf{c} \vdash e_1 := e_2 : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-Asg} \qquad \frac{\mathsf{c} \sqcup \sigma(e_1) \vdash p : (\sigma, \psi) \Rightarrow (\sigma', \psi') \quad \psi'' = \psi' \wedge (\text{confidentiality} \wedge (e_2 \neq \epsilon) \implies \mathsf{c} \sqcup \sigma(e_1) \sqcup \sigma(e_2) = \mathsf{L})}{\mathsf{c} \vdash \texttt{require}(e_1, e_2)\ \text{in}\ p : (\sigma, \psi) \Rightarrow (\sigma', \psi'')}\ \text{T-Req}$$

$$\frac{\mathsf{c} \sqcup \sigma(e) \vdash p_1 : (\sigma, \textbf{true}) \Rightarrow (\sigma_1, \psi_1) \quad \mathsf{c} \sqcup \sigma(e) \vdash p_2 : (\sigma, \textbf{true}) \Rightarrow (\sigma_2, \psi_2) \quad \sigma' = \sigma_1 \sqcup \sigma_2 \quad \psi' = \psi \wedge \psi_1 \wedge \psi_2}{\mathsf{c} \vdash \texttt{if}(e)\ \texttt{then}\ p_1\ \texttt{else}\ p_2 : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-If}$$

$$\frac{(\sigma', \psi') = \texttt{update}(\sigma, \psi, \mathsf{c}, (e_1, \sigma(e_1) \sqcup \sigma(e_2)))}{\mathsf{c} \vdash e_1.\texttt{push}(e_2) : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-Push} \qquad \frac{\mathsf{c} \vdash p_1 : (\sigma, \psi) \Rightarrow (\sigma_1, \psi_1) \quad \mathsf{c} \vdash p_2 : (\sigma_1, \psi_1) \Rightarrow (\sigma_2, \psi_2)}{\mathsf{c} \vdash p_1; p_2 : (\sigma, \psi) \Rightarrow (\sigma_2, \psi_2)}\ \text{T-Seq}$$

$$\frac{\mathsf{c} \sqcup \sigma(e) \vdash p : (\sigma, \psi) \Rightarrow (\sigma', \psi')}{\mathsf{c} \vdash \texttt{assert}(e)\ \text{in}\ p : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-Asr} \qquad \frac{(\sigma', \psi') = \texttt{update}(\sigma, \psi, \mathsf{c}, (id, \sigma(e))) \quad \mathsf{c} \vdash p : (\sigma', \psi') \Rightarrow (\sigma'', \psi'')}{\mathsf{c} \vdash \texttt{letvar}\ id := e\ \text{in}\ p : (\sigma, \psi) \Rightarrow (\sigma''[id \mapsto \sigma(id)], \psi'')}\ \text{T-LetVar}$$

$$\frac{(\sigma, \psi') = \text{confidentiality}\ ?\ \texttt{update}(\sigma, \psi, \mathsf{c}, e^*) : (\sigma, \psi)}{\mathsf{c} \vdash \texttt{emit}\ id_{\text{evt}}(e^*) : (\sigma, \psi) \Rightarrow (\sigma, \psi')}\ \text{T-Evt} \qquad \frac{(\sigma, \psi') = \text{confidentiality}\ ?\ \texttt{update}(\sigma, \psi, \mathsf{c}, e^*) : (\sigma, \psi)}{\mathsf{c} \vdash \texttt{revert}\ id_{\text{err}}(e^*) : (\sigma, \psi) \Rightarrow (\sigma, \psi')}\ \text{T-Rev}$$

$$\frac{(\sigma, \psi') = \texttt{update}(\sigma, \psi, \mathsf{c}, id_t, id_c, id_B)}{\mathsf{c} \vdash \texttt{selfdestruct}(id_c) : (\sigma, \psi) \Rightarrow (\sigma, \psi')}\ \text{T-Dst} \qquad \frac{(\sigma', \psi') = \texttt{update}(\sigma, \psi, \mathsf{c}, (id_B, \sigma(id_B) \sqcup \sigma(id)), id_t, id_c, id)}{\mathsf{c} \vdash id_c.\texttt{transfer}(id) : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-Trf}$$

$$\frac{\mathsf{c}_c \vdash e_{\text{new}} := id_c.id_{\text{con}}(e^*) : (\sigma_c, \psi_c) \Rightarrow (\sigma_c', \psi_c') \quad (\sigma', \psi') = \texttt{update}(\sigma, \psi, \mathsf{c}, (e_0, \sigma_c'(e_{\text{new}})), id_t, e_0, e^*)}{\mathsf{c} \vdash e_0 := \texttt{new}\ id_c(e^*) : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-New}$$

$$\frac{\mathsf{c}_c \vdash e_c := id_c.id_f(e^*) : (\sigma_c, \psi_c) \Rightarrow (\sigma_c', \psi_c') \quad (\sigma', \psi') = \texttt{update}(\sigma, \psi, \mathsf{c}, (e_0, \sigma_c'(e_c)), id_t, id_c, e^*)}{\mathsf{c} \vdash e_0 := id_c.\texttt{call}(id_f, e^*) : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-Call}$$

$$\frac{\mathsf{c}_c \vdash e_c := id_c.id_f(e^*) : (\sigma_c, \psi_c) \Rightarrow (\sigma_c, \psi_c) \quad (\sigma', \psi') = \texttt{update}(\sigma, \psi, \mathsf{c}, (e_0, \sigma_c(e_c)), id_t, id_c, e^*)}{\mathsf{c} \vdash e_0 := id_c.\texttt{staticcall}(id_f, e^*) : (\sigma, \psi) \Rightarrow (\sigma', \psi')}\ \text{T-SCall}$$

$$\frac{\mathsf{c} \vdash e' := id_c.id_f(e^*) : (\sigma, \psi) \Rightarrow (\sigma', \psi') \quad (\sigma'', \psi'') = \texttt{update}(\sigma', \psi', \mathsf{c}, (e_0, \sigma'(e')), id_t, id_c, e^*)}{\mathsf{c} \vdash e_0 := id_c.\texttt{delegatecall}(id_f, e^*) : (\sigma, \psi) \Rightarrow (\sigma'', \psi'')}\ \text{T-DCall}$$

$$\frac{\begin{array}{c} id_c.id_f(id^*)\ V\ M\ id_{m,1}(e_1^*)\ \cdots\ id_{m,n}(e_n^*)\ \text{returns}(id_{\text{ret}})\ \{\, p; \texttt{return}\ e_{\text{ret}}\,\} \quad \forall_{1 \le i \le n}.\ id_c.id_{m,i}(e_i^*)\ \{\, p_{1,i};\ \_;\ p_{2,i}\} \\ id^* = id_1, \cdots, id_n \quad e^* = e_1, \cdots, e_n \quad (\sigma_0, \psi_0) = \texttt{update}(\sigma, \psi, \mathsf{c}, (id_1, \sigma(e_1)), \cdots, (id_n, \sigma(e_n))) \\ (\sigma_0, \psi_1) = (\text{external call} \wedge M \in \{\texttt{payable}, \texttt{non-payable}\} \wedge \text{confidentiality})\ ?\ \texttt{update}(\sigma_0, \psi_0, \mathsf{c}, g_{\text{adr}}, id_c, g_{val}, e^*) : (\sigma_0, \psi_0) \\ p_1' = p_{1,1}; \cdots; p_{1,n}; p \quad p_2' = p_{2,n}; \cdots; p_{2,1} \quad \mathsf{c} \vdash p_1' : (\sigma_0, \psi_1) \Rightarrow (\sigma_1, \psi_2) \quad \mathsf{c} \vdash p_2' : (\sigma_1[id_{\text{ret}} \mapsto \sigma_1(e_{\text{ret}})], \psi_2) \Rightarrow (\sigma_2, \psi_3) \\ (\sigma_3, \psi_4) = \texttt{update}(\sigma_2, \psi_3, \mathsf{c}, (e_0, \sigma_2(id_{\text{ret}}))) \quad \sigma_4 = \sigma_3[id_1 \mapsto \sigma(id_1), \cdots, id_n \mapsto \sigma(id_n)] \end{array}}{\mathsf{c} \vdash e_0 := id_c.id_f(e^*) : (\sigma, \psi) \Rightarrow (\sigma_4, \psi_4)}\ \text{T-Fun}$$

$$\frac{\begin{array}{c} (e, \tau)^* = (e_1, \tau_1), \cdots, (e_n, \tau_n) \quad e'^* = e_1', \cdots, e_m' \quad \forall_{1 \le i \le n}.\ \sigma_i = \sigma[e_i \mapsto \mathsf{c} \sqcup \tau_i] \\ \sigma' = \sigma[e_1 \mapsto \sigma_1(e_1), \cdots, e_n \mapsto \sigma_n(e_n)] \quad \psi' = \psi \wedge \left(\bigwedge_{i=1}^{m} (\mathsf{c} \sqcup \sigma'(e_i')) = \mathsf{L}\right) \wedge \left(\bigwedge_{i=1}^{n} (\mathsf{c} \sqcup \tau_i \sqsubseteq \sigma'(e_i))\right) \end{array}}{\texttt{update}(\sigma, \psi, \mathsf{c}, (e, \tau)^*, e'^*) : (\sigma', \psi')}\ \text{T-Upd}$$

Fig. 6. Type inference rules and constraints conversion. Symbols such as id_t, id_c, id_B, id_{con}, e_{new}, e_c, g_{adr}, and g_{val} are consistent with Fig. 2. New symbols introduced here are: c_c, σ_c and ψ_c denote the control-flow security level, policy and constraints of the callee's contract.

of the record-related information is required to be L enforced by the constraint. T-New, similar to T-Call and T-SCall, states that the security level of a newly created contract is inferred within the context of the callee's contract. T-Fun states that the transaction record generated by an external user's invocation is considered only for confidentiality. In addition to performing inference on the pre-statements, function body, and post-statements, we also ensure that the argument passing process from each actual argument expression e to its formal parameter id during a function call is secure (i.e. $\mathsf{c} \sqcup \sigma(e) \sqsubseteq \sigma(id)$).

Example 1. Recall the motivating example in Fig. 3, where the assignment at line 6 induces a data flow from H to L, thereby violating the local-variable security. To avoid this, the security level of `owner` is computed as $\mathsf{c} \sqcup \sigma(\texttt{\_owner}) = \mathsf{L} \sqcup \mathsf{H} = \mathsf{H}$ in the IFS policy according to rule T-Asg. Consequently, SmartIFSyn will automatically insert the statement `require(owner_security_level==H)` at line 5 to enforce security. We remark that `owner_security_level` is a new variable used to track the security level of `owner` at runtime. Such variables and statements for tracking security levels can be instrumented in a straightforward way, thus are omitted in this work.

Declassification for integrity. In Solidity, `msg.sender` can denote an external user's address. Different from other inputs, its value cannot be arbitrarily forged. Thus, when `msg.sender` is required or asserted to be a trusted `storage` variable *var* with the security level L in some `require`

```
1 contract canDeclassify { // (i)
2   address public owner; // owner : L
3   function kill () public {
4     require(msg.sender == owner); // H -> L
5     selfdestruct(msg.sender); // msg.sender : L
6   }
7 }
```

```
1 contract canNotDeclassify { // (ii)
2   address public owner;
3   function kill(address lib, bytes data) public {
4     (bool ok,) = lib.delegatecall(data);
5     // lib(H), data(H) : owner = msg.sender
6     require(msg.sender == owner);
7     selfdestruct(msg.sender); // msg.sender : H
8   }
9 }
```

$$\frac{\begin{array}{c} e = (msg.sender == var) \quad \mathtt{Loc}(var) = \mathtt{storage} \quad \sigma(var) = \mathrm{L} \\ \text{no unimplemented } \mathtt{delegatecall} \text{ with a H-level parameters before} \\ \{u_1, \cdots, u_n\} = \{u \in \mathbb{U}^{in} \mid \mathtt{Loc}(u) \neq \mathtt{storage} \wedge u \notin \mathtt{BlkVars}\} \\ \mathtt{Guard}(e) \text{ in } p \quad \mathrm{c} \vdash p : (\sigma[u_1 \mapsto \mathrm{L}, \cdots, u_n \mapsto \mathrm{L}], \psi) \Rightarrow (\sigma', \psi') \end{array}}{\mathrm{c} \vdash \mathtt{Guard}(e) \text{ in } p : (\sigma, \psi) \Rightarrow (\sigma', \psi')} \text{ T-Dec}$$

```
1  contract canNotDeclassify { // (iii)
2    address public owner; // owner_0 : L
3    function init() public {
4      owner = msg.sender; // owner_1 : H
5    }
6    function kill () public {
7      require(msg.sender == owner); // owner_2 : H
8      selfdestruct(msg.sender); // msg.sender : H
9    }
10 }
```

Fig. 7. Three contracts to show declassification for integrity: (i) canDeclassify contract (top left), (ii, iii) canNotDeclassify contracts (bottom left, bottom right), and the T-Dec rule (top right), where Loc gives a storage type of a variable (`storage` or `non-storage`), Guard denotes a `require`/`assert` statement, and `BlkVars` denotes the set of Solidity predefined variables related to blocks.

or `assert` statement, inputs that are neither `storage` nor Solidity predefined variables related to blocks under the scope of the `require` or `assert` statement can be safely declassified from H to L, unless a `delegatecall` to an unimplemented function with some parameters of H security level occurs before that. This observation is formulated as the new rule T-Dec in Fig. 7.

We note that in Solidity, variables whose values are stored and maintained on the blockchain across transactions are called `storage` variables, while `non-storage` (i.e., `memory` and `calldata`) variables are transient and exist only during the execution of a function. We explicitly constrain `delegatecall` statements as above, because `delegatecall` to unimplemented functions can be compiled and executed at runtime in Solidity, but some H security level parameters may change the security levels of the `storage` variables. We conservatively exclude Solidity predefined variables related to blocks because they can actually be controlled by miners [18].

Example 2. Consider the examples in Fig. 7. In contract (i), the owner's address (`owner`) stored on-chain is trusted (L), whereas the user's address `msg.sender` is untrusted (H). Though an attacker may attempt to invoke `kill` via an arbitrary call, only the legitimate owner can satisfy the `require` statement at line 4; therefore, its value cannot be forged, justifying the declassification of `msg.sender` from H to L. Consequently, the execution of line 5 is secure. In contract, we cannot declassify the security level of `msg.sender` in contracts (ii) and (iii). In contract (ii), an adversary may perform a `delegatecall` where the parameter `data` has H security level, thus may overwrite `owner` within the current execution context using, e.g., `owner = msg.sender`. Thus, though it does not violate the `require` statement at line 6, we cannot safely declassify the security level of `msg.sender`. In contract (iii), we also cannot safely declassify the security level of `msg.sender` in the function `kill`, because the function `init` can directly modify `owner`. Indeed, in the SSA form, the security level of `owner` at line 7 is the least upper bound of the the security levels of `owner_0` and `owner_1`, where `owner_1` at line 4 has security level H due to `msg.sender`.

Lemma 1. The policy inferred by the the system is a local-variable security policy (σ_{loc}).

Proof Sketch. Consider two executions $c_1 \Rightarrow \langle s_1, r_1 \rangle$ and $c_2 \Rightarrow \langle s_2, r_2 \rangle$ and take rule T-Asg as an example for an assignment $e_1 := e_2$ that gives $\mathrm{c} \vdash e_1 := e_2 : (\sigma, \psi) \Rightarrow (\sigma', \psi')$. Suppose $c_1 \simeq_{\mathbb{U}_{\mathrm{L}}^{in}, \sigma'} c_2$, where σ' is the policy inferred by the type system. According to rules T-Asg and T-Upd, we have: $\sigma' = \sigma[e_1 \mapsto \mathrm{c} \sqcup \sigma(e_2)]$, i.e., $\sigma'(e_1) = \mathrm{c} \sqcup \sigma(e_2)$. If $\mathrm{c} \sqcup \sigma(e_2) = \mathrm{L}$, then $\sigma'(e_1) = \mathrm{L}$. As e_2 and c depend solely on inputs of the L security level, we have: $s_1(e_1) = s_2(e_1)$. According to Definition 1, this assignment after type inference satisfies local-variable security. Conversely, if

$c \sqcup \sigma(e_2) = H$, then $\sigma'(e_1) = H$. By Definition 1, the local-variable security is satisfied. Therefore, σ' is σ_{loc}. Missing cases can be found in the supplemental material. □

4.2 Constraint Solving for IFS

When the local-variable security policy σ_{loc} inferred by the type system does not satisfy the global-interaction security, it is not an IFS policy. To resolve this, we use constraint solving to synthesize an IFS policy. However, directly solving the constraint extracted by the type system may produce a policy, where almost all the variables are assigned by the security level L. Though it is an IFS policy, it may declassify many security levels of variables from H to L and is far from the users' annotations. In practice, users typically annotate security levels to crucial variables and expect to preserve their annotations maximally. Therefore, we propose to use MaxSAT [9] to synthesize an IFS policy that is maximally aligned with the users' annotations.

An MaxSAT instance is a pair $(\Psi_{\text{hard}}, \Psi_{\text{soft}})$ where the hard constraint Ψ_{hard} is the conjunction of constraints extracted during type inference, and the soft constraint Ψ_{soft} comprises weighted constraints of the form $\psi : w$. For each user-annotated variable or automatically-annotated input variable x, ψ asserts that the security level of the variable is equal to the annotated security level, and its weight w is the cost of violating ψ. A higher weight w indicates a stronger preference to preserve the annotated security level. In this work, the weight of user-annotated variable is set to 1 while the weight of automatically-annotated input variable is set to 0.5. A solution of the MaxSAT instance is an assignment mapping variables to security levels, i.e., a policy, under which Ψ_{hard} is satisfied and the accumulated cost of violated constraints in Ψ_{soft} is minimized, i.e., the accumulated cost of satisfied constraints in Ψ_{soft} is maximized.

Lemma 2. The policy derived through constraint solving is an IFS policy (σ_{IFS}).

Proof Sketch. Consider two executions $\rho_1 : c_1 \Rightarrow \langle s_1, r_1 \rangle$ and $\rho_2 : c_2 \Rightarrow \langle s_2, r_2 \rangle$ and take rules T-Asg and T-Dst as examples. Suppose $c_1 \simeq_{\mathbb{U}_L^{in,\sigma'}} c_2$, where σ' is the policy obtained by constraint solving. For an assignment $e_1 := e_2$ that gives $c \vdash e_1 := e_2 : (\sigma, \psi) \Rightarrow (\sigma', \psi')$, according to rules T-Asg and T-Upd, we obtained the constraint $c \sqcup \sigma(e_2) \sqsubseteq \sigma'(e_1)$. According to the partial order relation $\sqsubseteq$, this constraint prevents the case where $c \sqcup \sigma(e_2) = H$ and $\sigma'(e_1) = L$. Hence, by Definition 1, σ' is σ_{loc}. Meanwhile, according to rule Asg and Definition 2, since $O(\rho_1) = O(\rho_2) = \langle\rangle$, it follows the Definition 3 always holds. By Definition 3, σ' is also σ_{glb}, and therefore σ' is σ_{IFS} according to Definition 4. For a `selfdestruct` statement that gives $c \vdash$ `selfdestruct`$(id_c) : (\sigma, \psi) \Rightarrow (\sigma, \psi')$, we have: $\sigma' = \sigma$. By rules T-Dst and T-Upd, the constraint is $c \sqcup \sigma'(id_t) \sqcup \sigma'(id_c) \sqcup \sigma'(id_B) = L$. By rule Dst and Definition 2, the variables contained in $O(\rho)$ (i.e., id_t, id_c, and id_B) depend on L-level inputs after constraint solving; it follows that $O(\rho_1) = O(\rho_2)$. Therefore, by Definition 3, σ' is σ_{glb}. Meanwhile, since `selfdestruct` leads to the same termination state $exit$, σ' is σ_{loc} by Definition 1, and is σ_{IFS} by Definition 4. Missing cases can be found in the supplemental material. □

By Lemmas 1 and 2, we obtain

Theorem 1. The policy generated by SmartIFSyn is an IFS policy (σ_{IFS}).

Example 3. Recall the example in Fig. 3 that violates the global-interaction security, as the security level of `msg.sender` cannot be enforced solely using type inference. Our constraint solving, according to rule T-Dst, enforces the security level of `msg.sender` to be L and thus the statement `require(msg_sender_security_level == L)` is inserted at line 14. Similarly, the parameter `keyNumber` in Fig. 4 has the H security level. When the function `play` is invoked, `keyNumber` is recorded in the blockchain record, leading to information leakage. To mitigate this, our constraint solving, according to rule T-Fun, enforces the security level of `keyNumber` to be L and thus the statement `require(keyNumber_security_level == L)` is inserted at line 8.

Table 1. Comparison of related tools.

Method	Contract	Method	Input	Anno.	Open	Detect	Policy Gen
SmartIFSyn	ETH	Type system+MaxSAT	Source	●	●	●	●
AChecker [32]	ETH	Taint analysis+Symbolic execution	Binary	○	●	◐	○
Ethainter [12]	ETH	Taint analysis	Binary	○	●	◐	○
SCIF [87]	ETH	Type system+Code synthesis	Source	●	○	◐	◐
Securify [82]	ETH	Datalog-based pattern checking	Binary	○	●	◐	○
SeRIF [14]	any	Type system+Runtime verification	Source	●	○	◐	◐
SoMo [24]	ETH	Symbolic execution	Source	○	●	◐	○
STC/STV [41]	ETH	Type system+SAT	Source	◐	●	◐	◐

●: Fully supported, ◐: Partially supported, ○: Not supported, **Anno**: support user-defined annotations.

5 Implementation and Evaluation

We implement our approach in a fully automated tool, named SmartIFSyn, based on the Solidity static analyzer Slither [25] and Z3 [19] as the MaxSAT solver. SmartIFSyn currently supports IFS analysis of Solidity of versions 0.4-0.8.

Research questions. We conduct a comprehensive evaluation of SmartIFSyn to answer the following three research questions (RQs):
RQ1: How effective and efficient is SmartIFSyn in detecting information flow vulnerabilities compared to state-of-the-art tools targeting similar problems?
RQ2: How effective and efficient is SmartIFSyn in analyzing real-world smart contracts?
RQ3: How effective and efficient is SmartIFSyn in enforcing the IFS in smart contracts?

SOTA baselines. We consider 7 representative state-of-the-art (SOTA) tools supporting information flow analysis of smart contracts. Table 1 summarizes the comparison of our tool SmartIFSyn with these 7 tools, all of which only consider integrity. Since only 5 of 7 tools are open-sourced, thus we compare SmartIFSyn with them to answer RQ1. Only the open-source tool STC/STV can generate a policy to mitigate detected vulnerabilities, thus we compare STC/STV to answer RQ3.

Datasets. To answer RQ1, we use the benchmarks provided by the baselines AChecker, SoMo, and STC/STV. These contracts are written in Solidity of versions from 0.4 to 0.8 using 6 – 1,728 lines of code. Moreover, they are annotated with vulnerabilities that violate integrity. We note that Ethainter and Securify do not provide publicly available datasets with labeled vulnerabilities. To answer RQ2, we consider two widely-used open-source datasets containing real-world smart contracts: SmartBugs-wild [27] (47,398 contracts published before August 8, 2019) and Sailfish [10] (89,853 contracts published before February 18, 2022). To include more recent contracts, we also randomly sample 20,000 smart contracts from Etherscan [23] that are published between 2022 and June 30, 2025. In total, we get 157,251 real-world contracts and 146,524 real-world unique contracts after de-duplication from Sailfish which is larger. However, conducting experiments on such a larger number of real-world contracts without ground-truth is labor-intensive and resource-expensive. To be affordable, we perform a stratified random sampling [81], where each of 3 datasets is regarded as one stratum from which contracts are sampled proportionally. The sample size n is determined using the standard formula for estimating population proportion $n = \frac{z^2 \cdot p \cdot (1-p)}{e^2}$, where z is the critical value of the standard normal distribution, p is the population proportion, and e is the margin of error. To achieve a 99% confidence level with a ±1% margin of error, we set $z = 2.576$ and $e = 0.01$. We conservatively assume $p = 0.5$. Thus, the total sample size is $n = \frac{(2.576)^2 \cdot 0.5 \cdot (1-0.5)}{(0.01)^2} \approx 16,590$. To answer RQ3, we use all the contracts that are identified as vulnerable in RQ1 and RQ2.

The experiments were conducted on an Ubuntu 23.10 server with 64GB RAM and an AMD EPYC 9754, 128 Core 2.25 GHz CPU. The timeout is set to 120 seconds per contract.

Table 2. Performance comparison of vulnerability detection on different benchmark datasets (RQ1).

Metric	SmartIFSyn	AChecker	Ethainter	Securify	SoMo	STC/STV
Dataset1: AChecker (15 contracts, 15 vulnerabilities)						
Precision	0.94	**1.00**	**1.00**	0.67	**1.00**	0.82
Recall	**1.00**	0.80	0.07	0.13	0.33	0.60
F1-score	**0.97**	0.89	0.13	0.22	0.50	0.69
Average time (s)	**0.93**	12.40	3.26	26.96	1.02	13.17
Dataset2: SoMo (445 contracts, 445 vulnerabilities)						
Precision	0.82	0.83	**0.94**	0.76	0.90	0.81
Recall	**1.00**	0.35	0.30	0.22	0.99	0.40
F1-score	0.90	0.49	0.45	0.34	**0.94**	0.54
Average time (s)	3.71	11.15	4.87	43.86	**2.13**	25.36
Dataset3: STC/STV (110 contracts, 201 vulnerabilities)						
Precision	0.87	0.87	**0.97**	0.77	0.88	0.86
Recall	**1.00**	0.27	0.33	0.12	0.21	**1.00**
F1-score	**0.93**	0.41	0.49	0.21	0.34	0.92
Average time (s)	**1.22**	13.29	5.66	36.60	1.73	15.50

5.1 RQ1: Information Flow Vulnerability Detection with Ground-truth for Integrity

To evaluate the effectiveness and efficiency for detecting information flow vulnerability that violate integrity, we adopt four standard metrics: precision, recall, F1-score [35] and average time per contract. The results are reported in Table 2.

In general, SmartIFSyn outperforms all the SOTA baselines, particularly, SmartIFSyn achieved the best results on both Dataset1 and Dataset3 in terms of recall, F1-score and average time. On all the three datasets, SmartIFSyn did not have any false negatives (FN), namely, achieving 100% recall, because our type system is sound. However, our type system is incomplete, thus, unsurprisingly, false positives (FP) were reported by SmartIFSyn due to over-approximation. AChecker focuses on access control vulnerabilities and thus has limited ability to detect other types of information flow vulnerabilities, such as those arising from the manipulation of Solidity predefined variables related to blocks. On its own dataset (i.e., Dataset1), which only contains access control vulnerabilities, it achieved the second-highest F1-score, with no FPs and only a few FNs. In contrast, on Dataset2 and Dataset3, it generated both FPs and numerous FNs. Ethainter only detects explicitly defined vulnerabilities, such as *Tainted Owner Variable* and *Tainted Delegatecall* [12]. Thus, it achieved very lower recall but higher precision within its scope, resulting in a low F1-score. For Securify, we only consider *violations*, since Securify reports both *violations* and *warnings* [82], where *violations* is more likely to be true positives (TP). This conservative violation strategy leads to several FPs, for instance, any state update after an external call is regarded as a reentrancy vulnerability. This conservative behavior has also been noted by [12, 49, 74]. Securify made many FNs on three datasets because it can only analyze Solidity versions 0.5–0.6, which was also noted by [49]. SoMo mainly targets bypassable modifiers within contracts (i.e., detecting whether an attacker can bypass the modifier to launch an attack) and can only detect modifiers misused vulnerabilities. Thus, while SoMo achieved the best results on its own dataset (i.e., Dataset2), it made a larger number of FNs on Dataset1 and Dataset3. STC/STV performed well on its own dataset (i.e., Dataset3) while very poor on Dataset2, because it does not support contracts in Solidity of versions 0.6–0.8. Nevertheless, it achieved a high recall on supported contracts, because STC/STV is sound.

5.2 RQ2: Analysis of Real-world Contracts for both Integrity and Confidentiality

Effectiveness. The results of SmartIFSyn on the 16,590 real-world contracts are reported in Table 3. SmartIFSyn identified 281 potential information flow vulnerabilities in 258 contracts, among which

Table 3. Results of real-world contracts (RQ2): overall and sample validation.

Overall results				Sample manual validation						
Total	#Secure	#Vuln (#Con_v/#Int_v)	#Failed	Total	#Secure	#Vuln	Total	TP	FP	FN
16,590	16,255	281 (14/267)	77	358	100	281	381	243	38	0

```
1  contract Save { // (i)
2    uint256 private password;
3    uint256 constant num = 49409376313952921;
4    function withdraw(uint256 _password) public {
5      require(uint256(sha3(_password)) % password == num);
6      msg.sender.transfer(...);
7    }  // confidentiality vuln
8    function recovery(uint256 _password) public {
9      require(uint256(sha3(_password)) % password == num);
10     selfdestruct(msg.sender);
11   }
12 }
```

```
1  contract Filter { // (ii)
2    modifier pr() {...}
3    function _getRandomNum() public view returns(uint256) {
4      return uint256(keccak256(abi.encodePacked(
5      blockhash(block.number-1), gasleft())));
6    }
7    function randomOut(address f, uint256 m) internal pr {
8      f.transfer(_getRandomNum() % m); // integrity vuln
9    }
10 }
```

```
1  contract Approve { // (iii)
2    address public owner; // owner : L
3    mapping(address => mapping(address => bool)) approved;
4    uint public value;
5
6    modifier onlyOwner(){
7      require(msg.sender == owner);
8      _;
9    }
10
11   function addApprove(address user1, address user2)
12   public onlyOwner {
13     approved[user1][user2] = true;
14   }
15
16   function approveAndTransfer(address spender) public {
17     if (approved[msg.sender][spender]) {
18       // false positive (FP)
19       spender.transfer(value);
20     }
21   }
22 }
```

Fig. 8. Three real-world smart contracts: (i) contract violates confidentiality (top left), (ii) contract violates integrity (bottom left), and (iii) contract on which SmartIFSyn reported an FP (right).

14 violate confidentiality (#Con_v) and 267 violate integrity (#Int_v). SmartIFSyn also labeled 16,255 contracts as secure and failed on the remaining 77 contracts.

We performed an in-depth analysis of the 77 failures and found that: 12 failures were caused by Slither parsing errors (the same issue occurs when analyzing them the Slither-based tool SoMo [24]), while the remaining 65 failures resulted from timeouts (i.e., 120 seconds per contract). We note that 77 contracts account for only about 0.46% of all the 16,590 real-world contracts.

Since there is no ground-truth of the real-world contracts, to understand the effectiveness of SmartIFSyn, we conducted a further manual validation on the 281 potential information flow vulnerabilities in 258 contracts. We also randomly sampled 100 contracts from the 16,255 contracts that were labeled as secure by SmartIFSyn for re-examination. To avoid bias of manual analysis, two members of the research team independently conducted cross-validation on these 358 contracts. In cases of disagreement, a third member adjudicated. The results show that: 243 out of 281 potential vulnerabilities identified in 258 contracts were confirmed as actual information flow vulnerabilities (12 violate confidentiality and 231 violate integrity) in 223 contracts. Indeed, SmartIFSyn achieved a precision of 0.86, a recall of 1.0, and an F1-score of 0.92, demonstrating the effectiveness of SmartIFSyn for analyzing real-world smart contracts.

Case study. Fig. 8 (i) shows a contract that violates the global-interaction security. SmartIFSyn found a vulnerability that violates confidentiality. Though the contract employs the irreversible hash function SHA3 and a large integer num to prevent brute-force attacks on the password, according to rule Fun, once the contract owner invokes the `withdraw` function, its parameter `_password` (H) will be recorded in plaintext on the blockchain (line 4), resulting in the leakage of `_password`. An attacker can then exploit this to steal funds by invoking the `withdraw` function, obtain the contract balance by invoking the `recovery` function and subsequently destroy the contract (line 10). Fig. 8 (ii) shows a contract that also violates the global-interaction security. SmartIFSyn found a vulnerability that violates integrity in pseudo-random number generation. This vulnerability primarily arises from the controllable input `block.number` (H), which miners can manipulate

through block packing strategies [18], allowing the generated random number to be controlled and potentially enabling transaction manipulation (line 8). Figure 8 (iii) shows a contract on which SmartIFSyn reported an FP of violating integrity. In this contract, when the `addApprove` function is invoked, only calls from the owner (L) can pass the `onlyOwner` modifier (line 12). According to rule T-Dec, both input variables `user1` and `user2` are considered trusted; therefore, all information stored in `approved` is trusted. In the `approveAndTransfer` function, the transfer operation (line 19) is triggered only when `approved[msg.sender][spender] == true`, which is logically secure since only trusted addresses can satisfy this condition. Though an attacker's address cannot be added to `approved`, SmartIFSyn still propagates the least upper bound of the security levels of `approved`, `msg.sender`, and `spender`, where the security levels of `msg.sender` and `spender` are H. As a result, SmartIFSyn misclassified the transfer operation on line 19 as a potential vulnerability that violates integrity.

Vulnerability patterns. Among the 223 vulnerable contracts, 9 were found to hold Ether. In particular, one contract holds approximately 37.51 Ether, two contracts hold about 0.10 Ether each, and another holds about 0.05 Ether. We failed to reach their developers for mitigation due to the decentralized and anonymous nature of smart contracts. Similar observations have also been reported by the teams of teEther [47] and SoMo [24]. Based on a systematic analysis of the 243 confirmed information flow vulnerabilities in these real-world contracts, we identify four prevalent patterns (P1–P4), covering 88.48% of the vulnerabilities:

- P1: Vulnerabilities caused by incorrect assignments or improper initialization (89/243);
- P2: Critical transaction operations that can be controlled by an attacker (66/243);
- P3: Insecure information flows originating from low-level function calls (48/243);
- P4: Private data recorded on the blockchain, leading to information leakage (12/243).

P1–P3 are vulnerability patterns that violate integrity, while P4 is a vulnerability pattern that violates confidentiality. The remaining vulnerabilities (such as unsafe `new` statement), being limited in number and belonging to unique categories, are therefore excluded from the statistical analysis. To address the overlap of vulnerability patterns, we used the following rules: (1) vulnerabilities involving confidentiality are assigned to P4, as information recorded on the blockchain is publicly observable; (2) vulnerabilities involving low-level function calls are assigned to P3; (3) assignment-related and transaction-related vulnerabilities in the remaining cases are assigned to P1 and P2, respectively. P1 often arises from using contract-name-based constructors in early Solidity versions. If misnamed, these functions become publicly callable rather than constructors. P1 may also results from incorrect function visibility, exposing them to unauthorized external calls and thus allowing attackers to impersonate initialization routines and tamper with critical state variables (e.g., the `initOwner` function in Fig. 3). P2 typically results from incorrect or missing access control in transaction operations such as fund transfers or contract destruction. Attackers can exploit unguarded execution paths to manipulate asset flows or control the contract's lifecycle (e.g., the `kill` function in Fig. 4). P3 primarily stems from unsafe low-level function calls. Since low-level function calls bypass type checking and interface validation, insecure calls can introduce vulnerabilities (e.g., the `canNotDeclassify` contract with insecure `delegatecall` in Fig. 7 (ii)). P4 primarily arises from the recording of private data in blockchain records (e.g., the `Save` contract in Fig. 8 (i)).

Efficiency. We use the cumulative distribution function (CDF) [69] to analyze the per-contract analysis time on the 16,590 real-world contracts. The results show that over 95% of contracts were analyzed by SmartIFSyn within 5.40 seconds, and approximately 80% completed in less than 3.55 seconds. In addition, we found that only 2.9% of the contracts (485 in total) took more than 30 seconds. Among them, 65 contracts failed due to timeout, mainly because of a large code size and multiple external libraries.

Table 4. Results of RQ3, where * means newly introduced vulnerabilities.

Method	Analysis time			Variable coverage				Security enforcement		
	t_{ip}	t_{ti}	t_{cs}	#State_var	#Params	#Pre_var	Total	#Spec	#Spec$_v$	Elim$_r$
SmartIFSyn	28.35	2.43	36.18	**7,507**	**9,222**	**4,669**	**21,398**	1074	904	**100%**
STC/STV	99.87	N/A	99.87	2,937	0	0	2,937	432	351 (32*)	38.83%
Total number of variable				7,507	9,222	4,669	21,398	#Vuln		904

5.3 RQ3: Efficacy of Security Enforcement

As aforementioned, we compare with STC/STV [41] to answer RQ3 using all the 793 vulnerable contracts with 904 vulnerabilities (570 contracts from RQ1 and 223 real-world contracts from RQ2), although STC/STV only infers a policy to ensure integrity for local-variable security.

SmartIFSyn successfully generated IFS policies for all the 793 vulnerable contracts, where IFS policies for 184 vulnerable contracts with 216 vulnerabilities are generated via type inference and for the remaining 609 contracts with 688 vulnerabilities are generated via constraint solving. Detailed results are reported in Table 4, where t_{ip} shows the average time of generating an IFS policy for a contract, t_{ti} and t_{cs} respectively only show the average time via type inference and via constraint solving. It is easy to see that SmartIFSyn can quickly generates an IFS policy via type inference, sufficient for contracts that only violates the local-variable security.

In contrast, STC/STV only successfully generated IFS policies for 257 contracts, out of 793 vulnerable contracts. Moreover, it took more time than SmartIFSyn, because STC/STV solely relies upon constraint solving to systematically enumerate all the possible policies, among which the one that satisfies the security constraints and involves the maximum number of trusted (L) variables is selected as the desired IFS policy. The solved number of contracts remains the same even the timeout was extended to 10 miniutes.

We also report the number of input variables involved in the IFS policies generated by SmartIFSyn and STC/STV with the total number of variables in the smart contracts in Table 4, where #State_var, #Params, and #Pre_var denote the numbers of state variables, function parameters, and Solidity predefined variables, respectively. The results show that SmartIFSyn achieved a comprehensive coverage across all variable types, whereas STC/STV primarily focuses on state variables, resulting in a very lower overall coverage because: (1) it does not support more recent Solidity versions, and (2) it analyzes only local-variable security without imposing constraints on external inputs.

Table 4 also reports the number of specifications (#Spec), the number of valid specifications (#Spec$_v$) that successfully eliminate vulnerabilities, and the corresponding elimination rate (Elim$_r$) after inserting the generated `require` statements into the contracts by two tools. The results show that SmartIFSyn typically generates more specifications than the number of identified vulnerabilities due to FPs. Though this may lead to over-defense, it provides a comprehensive protection for securing information flows, successfully eliminating all the vulnerabilities. In contrast, STC/STV only provides a partial protection for securing information flows of local-variable security. Moreover, focusing only on the security of state variables even introduced new vulnerabilities: we found that STC/STV introduced 32 transaction-related vulnerabilities.

Threats to validity. The major internal threat to our evaluation is the correctness of the implementation of SmartIFSyn which relies on open-source analyzer Slither. Although some failures in the experiments were indeed caused by parsing errors in Slither, these accounted for only 0.07%. Moreover, we note that Slither has been widely used for years, which gives us a reasonable confidence in its reliability. Another internal threat is the manual validation which may introduce bias. We conducted a two-member independent cross-validation to minimize the bias and a third member also was involved in cases of disagreement.

Solidity is continuously evolving, and significant changes in newer versions may require updates to maintain compatibility. While the current implementation supports Solidity of versions 0.4 to 0.8, ensuring long-term adaptability remains an ongoing challenge. To address this external threat, we plan to actively monitor Solidity updates and adjust the implementation of our tool accordingly.

6 Related Work

Smart contract security has been extensively studied [43, 89]. This section focuses on code synthesis and information flow analysis, which are the most relevant to our work. We also briefly review other techniques to better position our contribution in the broader smart contract security landscape.

Code synthesis aims to automatically generate programs or specifications that meet given requirements [2, 26, 29, 76]. Several code synthesis approaches have been proposed to generate correct- or secure-by-design smart contracts [11, 28, 48, 62, 63, 66, 67], but they primarily target common vulnerabilities such as integer overflows, reentrancy, and logic errors. For instance, VeriSolid [63] generates secure-by-design smart contracts based on finite state machines, and thus enforces security at the design level. SGUARD [67] focuses on detecting vulnerabilities such as reentrancy and arithmetic overflow through runtime verification of bytecode by generating secure contracts accordingly. While these approaches enhance security, they do not systematically address information flow security. Although some efforts (e.g., [63, 66, 67]) may partially mitigate certain information flow vulnerabilities, they generally lack a systematic methodology for identifying and eliminating such vulnerabilities.

Information flow analysis ensures secure control and data flow across different security levels [15, 20, 21, 33, 34, 42, 50, 51, 71, 75, 83]. More recently, its principles have been adapted to smart contract security. For example, SeRIF [14] and STC/STV [41] use type systems to enforce integrity, while SCIF [87] detects vulnerabilities such as reentrancy and confused deputy attacks based on compositional information flow principles. Ethainter [12] employs taint analysis to track tainted data and detect composite vulnerabilities. SoMo [24] identifies bypassable modifiers via modifier dependency graphs and symbolic verification of access paths. AChecker [32] uses symbolic execution on bytecode to detect access control vulnerabilities. Securify [82] detects information-flow-related vulnerabilities by checking whether the contract complies with or violates predefined patterns. However, these tools generally overlook the enforcement of confidentiality and fall short of ensuring comprehensive global-interaction security, as evaluated in Section 5.

Other techniques for smart contract security include AI-based approaches [17, 54, 55, 77, 88], fuzzing [30, 44, 46, 52, 86], model checking [1, 5, 59, 65, 66], symbolic execution [16, 38, 45, 47, 56, 64, 68], and theorem proving [3, 7, 37, 39, 40, 60, 61, 70, 79]. These techniques have been widely used to detect various vulnerabilities and are effective in improving contract security and verifying correctness. However, they cannot synthesize enforceable IFS policies, which is the main focus of this work.

7 Conclusion

In this work, we have proposed SmartIFSyn, a novel automated approach for synthesizing IFS policies to secure smart contracts. To achieve this, we formalized the semantics of Solidity including record-related statements for the first time and defined IFS in smart contracts from two key perspectives: local-variable security and global-interaction security. Our approach integrates type inference and constraint solving to ensure both local-variable security and global-interaction security and support both confidentiality and integrity. The experimental results on a large number of real-world contracts demonstrate the effectiveness and efficiency of SmartIFSyn, detecting 243 vulnerabilities in 223 real-world Ethereum smart contracts and outperforming the SOTA tools.

Data Availability

The complete source code of our tool SmartIFSyn is available in an anonymous repository at https://github.com/anonymous-user-for-submission/SmartIFSyn. The datasets of smart contracts used in our experiments are also provided in an anonymous repository at https://github.com/anonymous-user-for-submission/SmartIFSyn-dataset.

References

[1] Tesnim Abdellatif and Kei-Leo Brousmiche. 2018. Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 1–5. doi:10.1109/NTMS.2018.8328737

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. doi:10.1109/FMCAD.2013.6679385

[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*. 66–77. doi:10.1145/3167084

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust*. 164–186.

[5] Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. 2019. Developing secure bitcoin contracts with BitML. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1124–1128. doi:10.1145/3338906.3341173

[6] Massimo Bartoletti and Livio Pompianu. 2017. An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In *Financial Cryptography and Data Security*. 494–509.

[7] 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*. 91–96. doi:10.1145/2993600.2993611

[8] Kenneth J Biba et al. 1977. Integrity considerations for secure computer systems. (1977).

[9] Nikolaj S. Bjørner and Anh-Dung Phan. 2014. *νZ* - Maximal Satisfaction with Z3. In *Proceedings of the 6th International Symposium on Symbolic Computation in Software Science (SCSS)*, Temur Kutsia and Andrei Voronkov (Eds.), Vol. 30. 1–9.

[10] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. 161–178. doi:10.1109/SP46214.2022.9833721

[11] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. 2018. Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts. In *27th USENIX Security Symposium*. 1335–1352.

[12] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469. doi:10.1145/3385412.3385990

[13] Zhuo Cai, Soroush Farokhnia, Amir Kafshdar Goharshady, and S. Hitarth. 2023. Asparagus: Automated Synthesis of Parametric Gas Upper-Bounds for Smart Contracts. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 253 (Oct. 2023). doi:10.1145/3622829

[14] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. 2021. Compositional Security for Reentrant Applications. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1249–1267. doi:10.1109/SP40001.2021.00084

[15] Hongxu Chen, Alwen Tiu, Zhiwu Xu, and Yang Liu. 2018. A Permission-Dependent Type System for Secure Information Flow Analysis. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 218–232. doi:10.1109/CSF.2018.00023

[16] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 442–446. doi:10.1109/SANER.2017.7884650

[17] Yizhou Chen, Zeyu Sun, Zhihao Gong, and Dan Hao. 2024. Improving smart contract security with contrastive learning-based vulnerability detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–11. doi:10.1145/3597503.3639173

[18] ConsenSys. 2020. Smart contract weakness classification registry. https://github.com/SmartContractSecurity/SWC-registry

[19] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS*. 337–340.

[20] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (may 1976), 236–243. doi:10.1145/360051.360056

[21] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (jul 1977), 504–513. doi:10.1145/359636.359712

[22] Ethereum. 2025. 2025 Solidity documentation. https://docs.soliditylang.org/en/latest/

[23] Etherscan. 2025. https://etherscan.io/

[24] Yuzhou Fang, Daoyuan Wu, Xiao Yi, Shuai Wang, Yufan Chen, Mengjie Chen, Yang Liu, and Lingxiao Jiang. 2023. Beyond "Protected" and "Private": An Empirical Security Analysis of Custom Function Modifiers in Smart Contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1157–1168. doi:10.1145/3597926.3598125

[25] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. doi:10.1109/WETSEB.2019.00008

[26] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 420–435. doi:10.1145/3192366.3192382

[27] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2021. SmartBugs: a framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1349–1352. doi:10.1145/3324884.3415298

[28] Bernd Finkbeiner, Jana Hofmann, Florian Kohn, and Noemi Passing. 2023. Reactive Synthesis of Smart Contract Control Flows. In *Automated Technology for Verification and Analysis*. 248–269.

[29] Bernd Finkbeiner, Niklas Metzger, and Yoram Moses. 2022. Information Flow Guided Synthesis. In *Computer Aided Verification*. 505–525.

[30] Ying Fu, Meng Ren, Fuchen Ma, Xin Yang, Heyuan Shi, Shanshan Li, and Xiangke Liao. 2019. EVMFuzz: Differential fuzz testing of Ethereum virtual machine. *Journal of Software: Evolution and Process* 36, 4 (2019), e2556. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2556 doi:10.1002/smr.2556

[31] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2022. eTainter: detecting gas-related vulnerabilities in smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 728–739. doi:10.1145/3533767.3534378

[32] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 945–956. doi:10.1109/ICSE48619.2023.00087

[33] Roberto Giacobazzi and Isabella Mastroeni. 2018. Abstract Non-Interference: A Unifying Framework for Weakening Information-flow. *ACM Trans. Priv. Secur.* 21, 2, Article 9 (Feb. 2018). doi:10.1145/3175660

[34] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. doi:10.1109/SP.1982.10014

[35] Cyril Goutte and Eric Gaussier. 2005. A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation. In *Advances in Information Retrieval*. 345–359.

[36] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 116 (Oct. 2018). doi:10.1145/3276486

[37] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *Principles of Security and Trust*. 243–269.

[38] Ningyu He, Ruiyi Zhang, Lei Wu, Haoyu Wang, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2020. Security Analysis of EOSIO Smart Contracts. *CoRR* abs/2003.06568 (2020). arXiv:2003.06568

[39] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 204–217. doi:10.1109/CSF.2018.00022

[40] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security*. 520–535.

[41] Xinwen Hu, Yi Zhuang, Shang-Wei Lin, Fuyuan Zhang, Shuanglong Kan, and Zining Cao. 2021. A security type verifier for smart contracts. *Computers & Security* 108 (2021), 102343. doi:10.1016/j.cose.2021.102343

[42] Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. *ACM SIGPLAN Notices* 41, 1 (2006), 79–90.

[43] Nikolay Ivanov, Chenning Li, Qiben Yan, Zhiyuan Sun, Zhichao Cao, and Xiapu Luo. 2023. Security Threat Mitigation for Smart Contracts: A Comprehensive Survey. *ACM Comput. Surv.* 55, 14s, Article 326 (jul 2023). doi:10.1145/3593293

[44] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 259–269. doi:10.1145/3238147.3238177

[45] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts. In *Ndss*. 1–12.

[46] Ziqiao Kong, Cen Zhang, Maoyi Xie, Ming Hu, Yue Xue, Ye Liu, Haijun Wang, and Yang Liu. 2025. Smart Contract Fuzzing Towards Profitable Vulnerabilities. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE008 (June 2025). doi:10.1145/3715720

[47] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium*. 1317–1333.

[48] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 438–453. doi:10.1145/3385412.3385982

[49] Kaixuan Li, Yue Xue, Sen Chen, Han Liu, Kairan Sun, Ming Hu, Haijun Wang, Yang Liu, and Yixiang Chen. 2024. Static application security testing (sast) tools for smart contracts: How far are we? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1447–1470.

[50] Peixuan Li and Danfeng Zhang. 2017. Towards a Flow- and Path-Sensitive Information Flow Analysis. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 53–67. doi:10.1109/CSF.2017.17

[51] Ximeng Li, Flemming Nielson, and Hanne Riis Nielson. 2016. Future-dependent Flow Policies with Prophetic Variables. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. 29–42. doi:10.1145/2993600.2993603

[52] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 65–68. doi:10.1145/3183440.3183495

[53] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen est omen: exploring and exploiting similarities between argument and parameter names. In *Proceedings of the 38th International Conference on Software Engineering*. 1063–1073. doi:10.1145/2884781.2884841

[54] Zhenguang Liu, Peng Qian, Xiaoyang Wang, Yuan Zhuang, Lin Qiu, and Xun Wang. 2023. Combining Graph Neural Networks With Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Transactions on Knowledge and Data Engineering* 35, 2 (2023), 1296–1310. doi:10.1109/TKDE.2021.3095196

[55] Feng Luo, Ruijie Luo, Ting Chen, Ao Qiao, Zheyuan He, Shuwei Song, Yu Jiang, and Sixing Li. 2024. SCVHunter: Smart Contract Vulnerability Detection Based on Heterogeneous Graph Attention Network. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–13. doi:10.1145/3597503.3639213

[56] 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*. 254–269. doi:10.1145/2976749.2978309

[57] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. 2015. Demystifying Incentives in the Consensus Computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 706–719. doi:10.1145/2810103.2813659

[58] Fuchen Ma, Meng Ren, Fu Ying, Wanting Sun, Houbing Song, Heyuan Shi, Yu Jiang, and Huizhong Li. 2023. V-Gas: Generating High Gas Consumption Inputs to Avoid Out-of-Gas Vulnerability. *ACM Trans. Internet Technol.* 23, 3, Article 40 (Aug. 2023). doi:10.1145/3511900

[59] Gabor Madl, Luis Bathen, German Flores, and Divyesh Jadav. 2019. Formal Verification of Smart Contracts Using Interface Automata. In *2019 IEEE International Conference on Blockchain (Blockchain)*. 556–563. doi:10.1109/Blockchain.2019.00081

[60] Diego Marmsoler and Achim D Brucker. 2021. A denotational semantics of Solidity in Isabelle/HOL. In *Software Engineering and Formal Methods*. Springer, 403–422.

[61] Diego Marmsoler and Billy Thornton. 2023. SSCalc: A Calculus for Solidity Smart Contracts. In *International Conference on Software Engineering and Formal Methods*. Springer, 184–204.

[62] Anastasia Mavridou and Aron Laszka. 2018. Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In *Financial Cryptography and Data Security*. 523–540.

[63] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security*. 446–465.

[64] B Mueller. 2017. Mythril-Reversing and bug hunting framework for the Ethereum blockchain.

[65] Zeinab Nehaï, Pierre-Yves Piriou, and Frédéric Daumas. 2018. Model-Checking of Smart Contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom)*

and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). 980–987. doi:10.1109/Cybermatics_2018.2018.00185

[66] Keerthi Nelaturu, Anastasia Mavridoul, Andreas Veneris, and Aron Laszka. 2020. Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 1–9. doi:10.1109/ICBC48266.2020.9169428

[67] Tai D. Nguyen, Long H. Pham, and Jun Sun. 2021. SGUARD: Towards Fixing Vulnerable Smart Contracts Automatically. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1215–1229. doi:10.1109/SP40001.2021.00057

[68] Ivica Nikolić, 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*. 653–663. doi:10.1145/3274694.3274743

[69] Athanasios Papoulis. 1965. *Random variables and stochastic processes*. McGraw Hill.

[70] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. 2018. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 912–915. doi:10.1145/3236024.3264591

[71] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Type-driven repair for information flow security. *CoRR abs/1607.03445* (2016).

[72] Michael Pradel and Koushik Sen. 2018. DeepBugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (Oct. 2018). doi:10.1145/3276517

[73] Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting argument selection defects. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 104 (Oct. 2017). doi:10.1145/3133928

[74] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*.

[75] A. Sabelfeld and A.C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19. doi:10.1109/JSAC.2002.806121

[76] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 404–415. doi:10.1145/1168919.1168907

[77] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639117

[78] Nick Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (Sep. 1997). doi:10.5210/fm.v2i9.548

[79] Bryan Tan, Benjamin Mariano, Shuvendu K. Lahiri, Isil Dillig, and Yu Feng. 2022. SolType: refinement types for arithmetic overflow in solidity. *Proc. ACM Program. Lang.* 6, POPL, Article 4 (Jan. 2022). doi:10.1145/3498665

[80] Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow as a Safety Problem. In *Proceedings of the 12th International Symposium on Static Analysis*, Vol. 3672. 352–367. doi:10.1007/11547662_24

[81] Steven K Thompson. 2012. *Sampling*. Vol. 755. John Wiley & Sons.

[82] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82. doi:10.1145/3243734.3243780

[83] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187.

[84] Shuai Wang, Liwei Ouyang, Yong Yuan, Xiaochun Ni, Xuan Han, and Fei-Yue Wang. 2019. Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49, 11 (2019), 2266–2277. doi:10.1109/TSMC.2019.2895123

[85] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[86] Valentin Wüstholz and Maria Christakis. 2019. Harvey: A Greybox Fuzzer for Smart Contracts. *CoRR* abs/1905.06944 (2019). arXiv:1905.06944

[87] Siqiu Yao, Haobin Ni, Andrew C. Myers, and Ethan Cecchetti. 2024. SCIF: A Language for Compositional Smart Contract Security. doi:10.48550/ARXIV.2407.01204

[88] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence*. 3283–3290.

[89] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2021. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2084–2106. doi:10.1109/TSE.2019.2942301