SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis

ZEQIN LIAO, Sun Yat-sen University, China
YUHONG NAN, Sun Yat-sen University, China
HENGLONG LIANG, Sun Yat-sen University, China
SICHENG HAO, Sun Yat-sen University, China
JUAN ZHAI, University of Massachusetts, Amherst, USA
JIAJING WU, Sun Yat-sen University, China
ZIBIN ZHENG*, Sun Yat-sen University, China and GuangDong Engineering Technology Research Center of Blockchain, China

With the increasing popularity of blockchain, different blockchain platforms coexist in the ecosystem (e.g., Ethereum, BNB, EOSIO, etc.), which prompts the high demand for cross-chain communication. Cross-chain bridge is a specific type of decentralized application for asset exchange across different blockchain platforms. Securing the smart contracts of cross-chain bridges is in urgent need, as there are a number of recent security incidents with heavy financial losses caused by vulnerabilities in bridge smart contracts, as we call them Cross-Chain Vulnerabilities (CCVs). However, automatically identifying CCVs in smart contracts poses several unique challenges. Particularly, it is non-trivial to (1) identify application-specific access control constraints needed for cross-bridge asset exchange, and (2) identify inconsistent cross-chain semantics between the two sides of the bridge.

In this paper, we propose SmartAxe, a new framework to identify vulnerabilities in cross-chain bridge smart contracts. Particularly, to locate vulnerable functions that have access control incompleteness, SmartAxe models the heterogeneous implementations of access control and finds necessary security checks in smart contracts through probabilistic pattern inference. Besides, SmartAxe constructs cross-chain control-flow graph (xCFG) and data-flow graph (xDFG), which help to find semantic inconsistency during cross-chain data communication. To evaluate SmartAxe, we collect and label a dataset of 88 CCVs from real-attacks cross-chain bridge contracts. Evaluation results show that SmartAxe achieves a precision of 84.95% and a recall of 89.77%. In addition, SmartAxe successfully identifies 232 new/unknown CCVs from 129 real-world cross-chain bridge applications (i.e., from 1,703 smart contracts). These identified CCVs affect a total amount of digital assets worth 1,885,250 USD.

CCS Concepts: • **Software and its engineering → Software creation and management**; • **Software creation and management → Software verification and validation**;

Additional Key Words and Phrases: Smart Contract, Static Analysis, Cross-chain Bridge, Bug Finding

*Corresponding Author

Authors' addresses: Zeqin Liao, Sun Yat-sen University, Zhuhai, China, liaozq8@mail2.sysu.edu.cn; Yuhong Nan, Sun Yat-sen University, Zhuhai, China, nanyh@mail.sysu.edu.cn; Henglong Liang, Sun Yat-sen University, Guangzhou, China, lianghlong@mail2.sysu.edu.cn; Sicheng Hao, Sun Yat-sen University, Zhuhai, China, haosch@mail2.sysu.edu.cn; Juan Zhai, University of Massachusetts, Amherst, USA, juanzhai@umass.edu; Jiajing Wu, Sun Yat-sen University, Zhuhai, China, wujiajing@mail.sysu.edu.cn; Zibin Zheng, Sun Yat-sen University, Zhuhai, China and GuangDong Engineering Technology Research Center of Blockchain, Zhuhai, China, zhzibin@mail.sysu.edu.cn.

ACM 2994-970X/2024/7-ART12
https://doi.org/10.1145/3643738

ACM Reference Format:
Zeqin Liao, Yuhong Nan, Henglong Liang, Sicheng Hao, Juan Zhai, Jiajing Wu, and Zibin Zheng. 2024. SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis. *Proc. ACM Softw. Eng.* 1, FSE, Article 12 (July 2024), 22 pages. https://doi.org/10.1145/3643738

1 INTRODUCTION

The rise of blockchain has prompted a wide range of blockchain platforms (e.g., Ethereum [Ethereum 2015], BNB [Binance 2022]) and crypto-assets (e.g., Bitcoin [Bitcoin 2009], Non-Fungible Token [Wikipedia 2023]). Given such a highly diverse and fragmented ecosystem, there is a strong need for data communication across different blockchain platforms (e.g., exchanging Ether to Bitcoin).

Cross-chain bridge is a specific type of application, working as an intermediary for information exchange (e.g., digital assets) across different blockchains. For example, Polygon network bridge [Polygon 2023], the most popular cross-chain bridge holding more than 2 billion USD, allows users to transfer tokens between Polygon and Ethereum blockchains without other un-trusted third parties. While gaining a market cap of billions of dollars [Chainspot 2023], cross-chain bridges face emerging security issues. Our investigation showed that in the recent two years, cross-chain bridges have suffered more than 29 security incidents. A large portion of security incidents in cross-chain bridges are caused by vulnerabilities that reside in their smart contracts. For instance, PolyNetwork [Network 2020] was exploited by an access-control vulnerability, leading to a total loss of 600 million USD [Wikipedia contributors 2022].

In this paper, we call vulnerabilities that are specific to cross-chain bridge smart contracts as *Cross-Chain Vulnerability (CCV)*. CCVs are unique to the cross-chain scenario (i.e., asset exchange) implemented by smart contracts. For example, while the root cause of a CCV could be a lack of fine-grained access control, such access control enforcement is not necessary in other traditional smart contracts. As another example, a CCV may caused by inconsistent semantics between the two sides of the bridge (see Section 2.2 for more details).

Given the severe impact of CCVs, there has been very limited research focusing on analyzing CCVs, not to mention a systematic detection framework for securing cross-chain bridges. More specifically, prior research [Duan et al. 2023; Lee et al. 2023] performed studies to understand the key patterns of cross-chain attacks. Based on their observation and findings, a set of mitigation suggestions were proposed to cross-chain developers. Unfortunately, these suggestions can not be directly applied, or enforced by existing cross-chain bridges. The most relevant work in terms of detecting CCV is Xscope [Zhang et al. 2022], which can identify a subset of CCVs via anomaly detection. However, Xscope requires analyzing collected statistics of on-chain transactions. With such a prerequisite, it can only detect CCVs that have already been exploited by attackers.

Our Work. In this paper, we propose SmartAxe, a new static analysis framework to detect CCVs for cross-chain bridge smart contracts. To the best of our knowledge, SmartAxe is the first of its kind to detect CCVs via program analysis at the bytecode level. In this capability, SmartAxe enables automatic security vetting for a variety of cross-chain bridge applications before their deployment, and hence, improves their security and amend potential risks that may cause severe damage (e.g., financial losses).

The root cause of CCVs lies in two aspects: 1) access control incompleteness and 2) cross-bridge semantic inconsistency. Identifying CCVs in bridge contracts faces the following two unique challenges in terms of static analysis.

- Firstly, identifying access control incompleteness relies on the precise extraction of access control constraints. Access control constraints consist of appropriate security checks over specific resources (e.g., a critical state variable in smart contract). Unfortunately, such security checks are implemented heterogeneously in bridge contracts. Moreover, associating related resources with such security checks introduces significant complexity.

- Secondly, identifying cross-bridge semantic inconsistency heavily relies on analyzing the fine-grained contextual information related to cross-chain communication. For example, inspecting the control-flow and data-flow dependency across multiple blockchains. These information are mostly overlooked by prior frameworks [Liao et al. 2023; Liu et al. 2021; Tsankov et al. 2018] when detecting smart contract vulnerabilities.

To this end, SmartAxe integrates two key designs in its static analysis process for CCV detection. Firstly, to identify necessary security checks in cross-chain bridges, SmartAxe models the heterogeneous implementations of access control in bridge contracts to a canonical form. In the meantime, SmartAxe pinpoints cross-chain related resources that need security checks through probabilistic pattern inference (Section 4.2). In this way, SmartAxe can effectively find out those vulnerable functions which have access control incompleteness. Secondly, to model the context information, SmartAxe aligns the control flow and data flow between two sides of the bridge, and further constructs the cross-chain control-flow graph (xCFG) and data-flow graph (xDFG) (Section 4.3). With the constructed graphs, SmartAxe locates vulnerable functions containing cross-bridge semantic inconsistency during cross-chain data communication. Lastly, SmartAxe analyzes the accessibility (i.e., entry point) and subversiveness (i.e., affected state variables) for vulnerable functions, and reports the vulnerability trace (Section 4.4).

To evaluate the effectiveness of SmartAxe, we first construct a manually-labeled dataset (as D_{manual}) based on 22 public reports discussing CCVs. The dataset is composed of 16 cross-chain bridge applications, with 88 CCVs from 203 smart contracts. Our experiments show that SmartAxe is effective in CCV detection with a precision of 84.95% and a recall of 89.77% over this dataset. The results show that SmartAxe effectively detect the majority of CCVs that cause real-world damage.

Detecting CCVs in the wild. With the help of SmartAxe, we perform a large-scale security vetting of 1,703 smart contracts (from 129 real-world cross-chain bridge applications). To the best of our knowledge, this is the most comprehensive collection of cross-chain bridge smart contracts in the wild. Finally, SmartAxe reported 232 new CCVs which have not been identified by previous research. The total assets affected by these CCVs reached 1,885,250 USD.

In summary, the contributions of this paper are as follows:

- We highlight the root causes of cross-chain vulnerabilities (Section 2.2), including access control incompleteness and cross-bridge semantic inconsistency.
- We propose SmartAxe, the first static analysis framework to detect CCVs for cross-chain bridge smart contracts.
- We perform an extensive evaluation to show the effectiveness of SmartAxe. In addition, by performing a large-scale study over 1,703 cross-chain bridge smart contracts in the wild, SmartAxe identified 232 new CCVs in the real world.
- We build the first manual-labeled dataset of cross-bridge vulnerabilities, as well as the most comprehensive dataset of cross-chain bridge applications/smart contracts. To benefit future research, we release the artifact of SmartAxe, as well as the datasets [1].

[1] https://github.com/InPlusLab/FSE24-SmartAxe

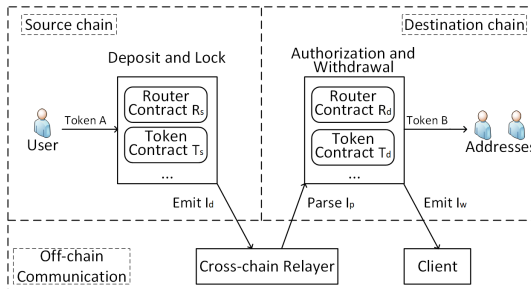


Fig. 1. Workflow of cross-chain bridge.

2 BACKGROUND AND MOTIVATION

2.1 Smart Contract and Cross-Chain Bridge

Smart contract is a specific type of program running on the blockchain. These programs support various functionalities to implement new business models [Zheng et al. 2020], such as decentralized finance (DeFi), decentralized gaming (GameFi), and cross-chain bridges.

Cross-chain bridge works as an intermediary for asset exchange on different blockchain platforms. Figure 1 shows the key architecture of the cross-chain bridge. A typical cross-chain bridge can be divided into three parts: source chain, cross-chain relayer, and destination chain. The cross-chain bridge deploys smart contracts on the source chain and destination chain. The relayer is designed to support information exchange between the source chain and the destination chain. With the cross-chain bridge, users can deposit assets on the source chain and withdraw assets on the destination chain. For example, exchange Token A with Token B in Figure 1. To detail, this process includes the following three steps:

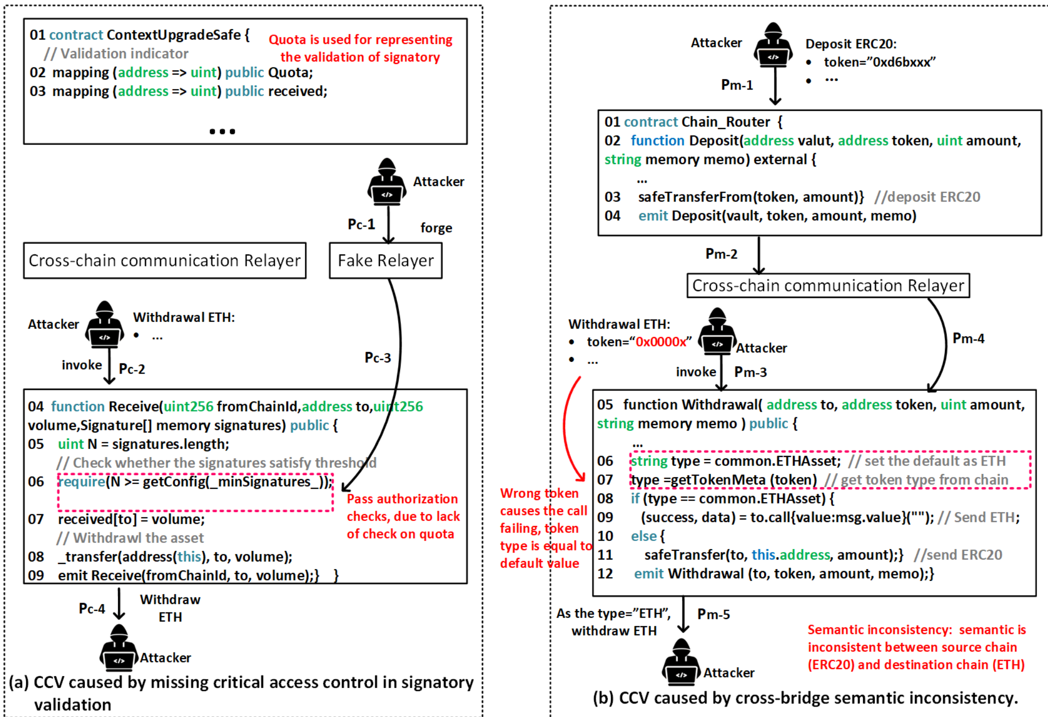
(1). **Asset deposit on source chain.** When receiving the asset exchange request from the user, the *Router contract* R_s of the source chain invokes the *Token contract* T_s to lock *Token A*. Then R_s emits a deposit event I_d as the confirmation of locked assets, which contains the detailed deposit information (e.g., the type and amount). After that, the asset of users would be transferred to the *Router contract* R_s

(2). **Cross-chain communication via off-chain relayers.** Once a deposit event I_d is emitted, the off-chain relayer verifies whether the deposit is valid on the source chain. If the verification gets passed, the relayer transmits the informed information I_p to *Router contract* R_d of the destination chain.

(3). **Asset withdrawal on destination chain.** Once I_p is delivered to the destination chain, the *Router contract* R_d validates various proofs from I_p for authorization. After the validation, R_d emits a withdraw event I_w and invokes *Token contract* T_d to withdraw Token B to user-specified addresses on the destination chain.

2.2 Definition and Problem Statement

Cross-Chain Vulnerability. Cross-chain vulnerability (CCV) is a specific type of vulnerability in cross-chain bridge smart contracts. In most cases, cross-chain bridge contract unexpectedly introduces incomplete access control or inconsistent cross-bridge semantics when exchanging assets between the two blockchains. In the following, we use two motivating examples (Figure 2) to illustrate CCV. The examples are collected from two real-world bridge contracts that have been exploited by attackers (i.e., ChainSwap [Sam Cooling 2021] and ThORChain [Sebastian Sinclair 2021]). We re-organized the original contract code for better illustration.



(a) CCV caused by missing critical access control in signatory validation

(b) CCV caused by cross-bridge semantic inconsistency.

Fig. 2. Two motivating examples of CCV.

- **Access Control Incompleteness.** Cross-chain bridge contracts may omit critical security checks, or contain incorrect access control implementations. As shown in Figure 2(a), variable *Quota* in Contract *ContextUpgradeSafe* represents the validation of signatory (line 2), which should be checked before asset authorization and withdrawal. However, while Function *Receive* is designed for asset authorization and withdrawal, it only checks the signatures without validating the signatory (i.e., checking *Quota*, see the red dotted box (line 6 and 7)). Therefore, the contract introduces an exploitable CCV. As shown in path P_c, to attack this cross-chain bridge, the attacker can forge fake relayers (i.e., signatory) for signatures (P_c-1) and bypass authorization check due to the lack of signatory validation (P_c-3), and finally withdraw asset (P_c-4). In practice, this vulnerability can be avoided by adding statements such as *"require (Quota[signatory] > 0)"* between lines 6 and 7.
- **Cross-bridge Semantic Inconsistency.** Ideally, the program semantics of the source chain and destination chain should be aligned with each other, such as type and amount of the exchanged assets. Figure 2 (b) shows an example of semantic inconsistency caused by incorrect parsing of token type. For contract *Chain_Router*, function *Deposit* emits a record (line 4) of "ERC-20" on the source chain. However, the default token expected in function *Withdrawal* is "ETH" (line 6). While the actual token type can be updated by *getTokenMeta(token)*, the smart contract on the destination chain does not carefully handle and inspect exceptions. Particularly, an adversary can intentionally pass a wrong token address (i.e., 0x0000x) here, making the token type incorrectly parsed as the default value ("ETH") regardless of the actual token type on the source chain. Since ETH is more expensive than ERC-20, the attacker can benefit from exchanging assets with such value differences.

```
1 contract Radar {
2   function bridgeTokens(uint256 amount,bytes32 destChain,address destAddress) external {
3     require(balanceOf(msg.sender) >= _amount); // check the balance for ensuring deposit sucess
4     ...
5     safeTransferFrom(msg.sender, address(this), amount);
6     emit Deposit(amount, destChain, destAddress);}}
7
8 contract Polkabridge {
9   mapping(address => uint) public balanceOf;
10  function mint(address to) external {
11    ...
12    require(liquidity > 0); // check the liquidity for ensuring deposit sucess
13    balanceOf[to] = balanceOf[to] + liquidity;
14    emit Deposit(address(0), to, liquidity);}}
```

Fig. 3. An example of the diversity of security checks for deposit success (i.e., lines 3 and 12), which is caused by different implementations and non-standard check, and cannot be addressed by Ethainter [Brent et al. 2020], SPCon [Liu et al. 2022], AChecker [Ghaleb et al. 2023].

Existing Work and Limitations. Despite that CCVs have been extensively exploited, there is limited work on identifying this type of vulnerability in advance and further eliminating such losses. To the best of our knowledge, the most relevant work on identifying cross-chain attack is Xscope [Zhang et al. 2022]. However, Xscope is an anomaly detection tool to analyze cross-chain transactions, and it does not support CCV detection at the smart contract level.

Further, as smart contracts are difficult to support patches after deployment, such a framework cannot avoid the attack and further eliminate economic losses. Additionally, Duan et al. [Duan et al. 2023] and Lee et.al [Lee et al. 2023] conduct the survey on cross-chain attacks and propose advice on designing cross-chain systems, so both studies cannot fundamentally detect CCV.

2.3 Scope of Our Work

SmartAxe is designed to be a generalized framework for detecting CCVs caused by vulnerable smart contracts in cross-chain bridges. SmartAxe targets the upper-level application of bridge (i.e., bridge Dapp), rather than the underlying protocol of bridge (e.g., Inter-blockchain communication (IBC) protocol). As a program analysis framework, SmartAxe can cover 20 CCV attacks of 29 cross-chain security attacks, but does not cover other 9 cross-chain exploits irrelevant to smart contract code, such as private key leakage [Rubic 2022], DNS hijacking [CelerNetwork 2022] and trusted root leakage [Behnke 2022]. Given a set of smart contracts in the cross-bridge application, SmartAxe performs static analysis on the bytecode, and further reports whether the smart contract contains CCVs. Since SmartAxe conducts analysis on the level of bytecode instead of source code, it is applicable for a number of security vetting scenarios such as large-scale third-party auditing.

3 DESIGN OF SMARTAXE

3.1 Challenges and Solutions

With the increasing complexity of cross-chain smart contracts, identifying CCVs is by no means trivial. While prior works conducted analysis on access control vulnerability for smart contracts (e.g., Ethainter [Brent et al. 2020], SPCon [Liu et al. 2022], AChecker [Ghaleb et al. 2023]), these works never consider the security assumption of cross-bridge scenario, therefore, they can hardly detect CCVs. Below we list the challenges encountered by SmartAxe as well as the corresponding solutions.

C1: Extracting access control constraints. A typical access control constraint commonly consists of security checks over specific resources in smart contracts. Specifically, security checks can be conditional or compared statements (e.g., `require`, `assert`, and `if`), the resource can be the

```
1 contract BaseBridge {
2   mapping (address => uint) public authorization;
3   function withdrawal(uint256 SrcChain,address to,uint256 amount,Signature[] memory signatures) public {
4     ...
5     if(signatures.length > _minSignatures); {  //Security Check 1, check the number of signatures
6       for(uint i=0; i<signatures.length; i++) {
7         address _signatory = getSignatory(signatures[i]);
8         require(_signatory == signatures[i].signatory);  //Security Check 2, check the validation signatory
9         authorization[to]=authorization[to]+1;
10        emit Authorize(SrcChain, to, amount, signatory);}}
11    require(authorization[to] == signatures.length);
12    _transfer(address(this), to, volume);  // Resource 1, withdrawal the asset
13    emit Receive(fromChainId, to, volume);}}
```

Fig. 4. An example to show the complexity in linking resources with security checks.

important operation statement (e.g., read and write on state variables, method invocation). From this perspective, the challenge of extracting access control constraints can be divided into two aspects. The first aspect is the diversity of security checks which is caused by different implementations and non-standard checks. For instance, Figure 3 shows the diverse implementation of security checks between different cross-chain bridges. As can be seen, to check the deposit success, contract *Radar* compares the balance of the user with the deposit amount (line 3), contract *Polkabridge* compares the liquidity of the bridge with the threshold (line 12). However, prior research (i.e., Ethainter [Brent et al. 2020], SPCon [Liu et al. 2022], AChecker [Ghaleb et al. 2023]) cannot address this diversity, as their detection patterns are incomplete without considering the semantic of the bridge.

The second aspect is the inherent complexity in linking resources to security checks. We take Figure 4 as an example. For the resource of line 12, prior works [Brent et al. 2020; Ghaleb et al. 2023; Liu et al. 2022] link this resource to the security checks on which the resource has control flow dependency, i.e., only check of line 5 without check of line 8, which causes the mistake. Actually, by analyzing the semantics on them, we can find that the resource of line 14 transfers the authorized withdrawal, the check of line 5 checks the authorizing signatures, and the check of line 7 checks the authorizing signatory, so all of the checks of lines 5 and 7 should be linked with resource 12. Alternatively, by analyzing the data dependency of lines 8, 9, and 11, we can also determine check of line 7 should be linked with resource 12. However, automatically identifying these is by no means trivial, as it requires analyzing the complex patterns of semantic and data dependency.

To overcome this challenge, for identifying the security checks, we review the documentation and program code of the top 100 cross-chain bridges, and model the access control of cross-chain bridge and normalizes diverse checks to a canonical form (see details via Table 1 and Section 4.2). To link the resources to security checks, SmartAxe utilizes a probabilistic pattern inference method (see detail via Table 2 and Section 4.2). Specifically, SmartAxe utilizes a set of predefined patterns that consider the dependency of control flow, data flow, and semantics between the resource and security check to determine the association relationship between them.

Further, based on the extracted access control constraint, SmartAxe identifies the incompleteness of access control for the bridge contract.

C2: Identifying cross-bridge semantic inconsistencies. Different from the vulnerability analysis on a single blockchain, identifying CCVs of cross-bridge semantic inconsistency heavily relies on modeling the contextual information (e.g., emitting I_d, relayer, informing I_p) during cross-chain data transmission. Identifying the contextual information is by no means trivial, as it requires the accurate alignment of control flow and data flow between two sides of the cross-chain bridge, which lack of prior work can support. Further, it is difficult to identify the alignment sites of control flow and data flow, because locating alignment sites requires fine-grained semantic and control flow analysis.

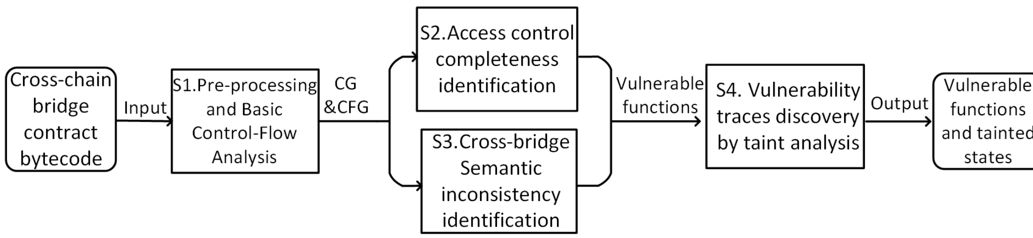


Fig. 5. The workflow of SmartAxe.

To overcome the second challenge, by the fine-grained semantic and control flow analysis, SmartAxe identifies two types of functions as the alignment site, i.e., (1) functions that implement deposit and lock and (2) functions that implement authorization and withdrawal. Then, SmartAxe aligns the corresponding alignment sites and constructs the cross-chain control flow graph (xCFG). To facilitate the detection for cross-bridge semantic consistency of CCVs, SmartAxe performs the data flow analysis on the xCFG to construct the cross-chain data-flow graph (xDFG).

Further, based on the constructed xCFG and xDFG, SmartAxe identify the CCV of cross-bridge semantic inconsistency for bridge contracts.

3.2 Workflow of SmartAxe

SmartAxe takes the bytecode of cross-chain bridge contracts as its input, and finally reports whether it contains CCVs, as well as the corresponding vulnerable traces. A vulnerable trace contains function calls from the vulnerable function to tainted state variable(s) that can be affected by external call(s). Figure 5 demonstrates the workflow of SmartAxe.

S1. **Basic control-flow analysis.** As a typical process of static analysis, SmartAxe separately recovers the control flow of smart contracts on each side of the cross-chain bridge as the pre-processing step, through the existing analyzer (e.g., SmartDagger [Liao et al. 2022] in our research).

S2. **Access control completeness identification.** In the second step, SmartAxe identifies all the access control constraints based on the basic control-flow facts. Particularly, SmartAxe models the access control of the bridge contracts and normalizes diverse checks to a canonical form. Then, SmartAxe conducts the probabilistic pattern inference to associate the resources with security checks. Based on the extracted access control constraints, SmartAxe identifies the vulnerable functions containing incompleteness of access control.

S3. **Cross-bridge semantic inconsistency identification.** SmartAxe aligns the control flow of smart contracts between the source chain and destination chain to construct xCFG. Further, SmartAxe performs data flow analysis to construct xDFG. Based on the constructed graphs, SmartAxe identifies the vulnerable functions containing semantic inconsistency.

S4. **Vulnerable traces discovery by taint analysis**. Lastly, based on the vulnerable functions reported by S3 and S4, SmartAxe identifies all the vulnerable traces via taint analysis.

4 APPROACH DETAILS

In this section, we demonstrate the details of each step in SmartAxe. Meanwhile, we utilize a running example (i.e., Figure 2) to elaborate how SmartAxe accurately identifies CCVs.

4.1 Basic Control-Flow Analysis

SmartAxe separately constructs the basic control flow from the smart contract bytecode on each side of a given cross-chain bridge by utilizing the state-of-the-art static analysis tool SmartDagger. SmartAxe utilizes SmartDagger which is developed for identifying cross-contract vulnerability, as SmartDagger can construct a more complete control flow for cross-function (contract) invocation compared with other static analysis tools (e.g., Slither [Feist et al. 2019] and Mythril [Consensys 2017]). Specifically, for each side of cross-chain bridge, SmartDagger constructs the control flow graph from the bytecode of the bridge contracts.

4.2 Access Control Incompleteness Identification

In this subsection, SmartAxe extracts the access control constraints by modeling the security checks of access control and associating the resources with security checks. Based on extracted access control constraints, SmartAxe identifies the incompleteness of access control.

Modeling heterogeneous security checks. As mentioned earlier, the workflows of cross-chain bridges consist of three steps, 1) asset deposit and lock, 2) cross-chain communication, and 3) asset authorization and withdrawal. The security checks of bridge contracts essentially enforce the permission of users according to such three workflows. However, the implementation for security checks is actually much more diverse in the cross-chain bridge. This is because different cross-chain bridges generally utilize different kinds of security features that are associated with these workflows to realize the security check of access control. Hence, we have to identify, model, and normalize all these access control checks in various forms in order to perform a comparison.

Modeling security checks is based on the fact that, while most of security checks are different in syntactic form, they are semantically equivalent in terms of the protection they provide. SmartAxe adopt the same definition of equivalence proposed in previous research [Aafer et al. 2018]. Our definition of equivalence is regarding the protection enabled by the security checks, that is, the kind of malicious behaviors precluded by the security checks. We turn to Figure 3 to illustrate this, checks of line 3 and 12 are different from the perspective of syntactic form. However, they are equivalent to each other in terms of the protection they provide, as the check of line 3 ensures sufficient balance for the deposit success and the check of line 12 ensures sufficient liquidity for the deposit success.

To establish requisite access control mechanisms in cross-bridge smart contracts, we collect the top-100 bridge Dapps from Chainspot [Chainspot 2023], a website for blockchain bridge aggregation. Our domain experts meticulously scrutinize these Dapps to discern all security checks embedded within their smart contracts. According to the workflows of bridges, our domain experts categorize these security checks accordingly. For each category, they further stratify them into distinct perspectives based on the targeted protection areas. Subsequently, they provide a comprehensive summary of the corresponding security features and their respective applications associated with each perspective.

Table 1 shows our summarized security check model of access control. SmartAxe divides the security checks of access control into three categories: (1) category that is specific to asset deposit and lock, (2) category that is specific to cross-chain router, and (3) category that is specific to asset authorization and withdrawal (i.e., the first column). Therefore, we propose to model the security checks of bridge contracts as follows.

$$BridgeCheck := [DepositandLock, CrosschainRouter, Authorizationandwithdrawal] \tag{1}$$

Category-1 - asset deposit and lock: This category includes a success check for the deposit and a validation check for arguments passed by users (i.e., the first two entries of the second column).

Table 1. Security check model of access control for cross-chain bridge contract.

Category	Perspective	Security feature	Example of usage
C1. Asset deposit and locking	P1.Success check for the deposit	Balance of bridge after deposit	Comparison with balance before deposit
		Balance of user	Comparison with the deposit amount
		Liquility of bridge	Comparison with deposit threshold
	P2.Validation check for arguments of user	Arguments of public function	Comparison with logic condition
		Arguments of user message	Comparison with logic condition
C2.Cross-chain router	P3.Correctness check for cross-chain router	Bridge-supported token/chain	Comparison with ID of destination
		Address of external invocation	Comparison with 0 address
C3. Asset authorization and withdrawal	P4.Validation check for verification	Signature and Signatory	Comparison with cross-chain message
		Timeout of signature	Comparison with on-chain time status (e.g., timestamp, blocknumber)
	P5.Check for repetitive withdrawal	List recording the withdrawal	Consultation on the lists (i.e., mapping variables)
	P6.Correctness check for releasing	Recevier address	Comparison with user-specified address or 0 address

Specifically, a success check for the deposit is used to confirm the deposit is transferred to the cross-chain bridge and thus prevent fake deposits. The validation check for arguments passed by users is used to prevent users from passing into malicious arguments that cause malicious modification on the contract state. As shown in the first three entries of the third column, security checks of each perspective can adopt various forms, such as comparing the balance of the bridge before and after the deposit, comparing the user balance with the deposit amount, or comparing the liquidity of the bridge with the deposit threshold. Despite the three types of security checks being different in syntactic form, they are semantically equivalent in terms of confirming the success of the deposit, so there is a disjunction relationship between them. In addition, in the fourth and fifth entries of the third column, validation checks for arguments can be implemented by checking the arguments of public function and message invocation (e.g., *msg.sender*, *msg.value*). Similarly, these two types of security checks are semantically equivalent, so there is a disjunction relationship between them. Further, we formulate the above-illustrated understanding as follows.

$$\mathit{DepositandLock} :=[\mathit{DepositSuccess}, \mathit{Argument}] \tag{2}$$

$$\mathit{DepositSuccess} :=\mathit{check}(\mathit{BridgeBalance}) \wedge \mathit{check}(\mathit{UserBalance}) \wedge \mathit{check}(\mathit{AssetOwnership}) \tag{3}$$

$$\mathit{Argument} :=\mathit{check}(\mathit{FunctionArgument}) \wedge \mathit{check}(\mathit{MessageArgument}) \tag{4}$$

Category-2 - cross-chain routers: This category mainly checks the correctness of cross-chain routers, as shown in the third entry of the second column. A correctness check for the cross-chain router is used to prevent unexpected logic or errors in the cross-chain data transmission. This type of check can be implemented by (1) checking the supports of the bridge (e.g., Token ID, chain ID) and (2) checking the error of external invocation (e.g. 0 address can cause the external invocation to fail). Obviously, these two types of checks are not semantically equivalent to each other, so SmartAxe adopts a conjunction relationship between them. We formulate understandings as follows.

$$\mathit{CrosschainRouter} :=[\mathit{Correctness}] \tag{5}$$

$$\mathit{Correctness} :=\mathit{check}(\mathit{Support}) \vee \mathit{check}(\mathit{ExternalAddress}) \tag{6}$$

Category-3 - asset authorization and withdraw: This category includes a validation check for authorization, the repetitiveness check, and the correctness check for withdrawal (as shown in the fourth, fifth, and sixth entries of the second column). Specifically, the validation check for authorization is used to ensure that the cross-chain transaction has been signed and proved by relayers. To this end, such validation checks are implemented by checking the correctness

Table 2. Probabilistic inference pattern for associating resources with security checks.

Pattern	Condition	Probabilistic assignment
P1	$ControlFlowDependency(c, \{r\})$	$Association(c, p, r) = true(0.95)$
P2	$ControlFlowDependency(c, R) \lor r \in R$	$Association(c, p, r) = true(0.60)$
P3	$SameBlock(r_1, r_2)$	$Association(c, p_1, r_2) \xrightarrow{0.60} Association(c, p_1, r_1)$
P4	$SemanticCorrelation(r_1, r_2)$	$Association(c, p_2, r_2) \xrightarrow{0.70} Association(c, p_1, r_1)$
P5	$DataFlowDependency(r_1, r_2)$	$Association(c, p_2, r_2) \xrightarrow{0.80} Association(c, p_1, r_1)$

and timeout for the signature or signatory. Due to the inequivalence between them, we utilize a conjunction operation to measure them. The check for repetitive withdrawal is used to avoid users receiving tokens from cross-chain bridge repetitively. For this purpose, this type of check is implemented by inspecting the specific lists (e.g. *mapping variable*) that are used to record the withdrawal. Further, a correctness check for withdrawal is used to ensure that the transfer targets are valid. This type of security check can be implemented by inspecting the receiver address.

$$\begin{aligned} Authorizationandwithdrawal :=& [AvoidanceofRepetition, AuthorizationVerification, \\ & withdrawalCorrectness] \end{aligned} \tag{7}$$

$$AvoidanceofRepetition := check(RecordedList) \tag{8}$$

$$AuthorizationVerification := check(Signature) \lor check(Timeout) \tag{9}$$

$$withdrawalCorrectness := check(RecevierAddress) \tag{10}$$

Further, by comparing the control flow with our summarized security check model (Table 1), SmartAxe extracts all the security checks of access control for the bridge contracts.

Associating resources with security checks. After the extraction for security checks, SmartAxe identifies the resources of access control constraints, and associates the resources with the corresponding security checks through the probabilistic pattern inference method.

SmartAxe considers four types of resources for access control constraints: (1) FieldAccess, denoted as f, (2) Internal method, represented as m, (3) application binary interface (ABI), denoted as a, and (4) event emitting statement, denoted as e. Further, f represents the statements that read or write on state variables (i.e., global variables of smart contracts). m represents the statements that invoke internal methods (e.g. private function) in cross-chain bridge contracts. a refers to the interface of external calls (e.g. public function) in contracts. e refers to the event emitting statements that record the cross-chain data transmission (e.g., deposit record I_d). For these four types of resources, the first three types of resources have been well discussed in the existing works [El-Rewini et al. 2022]. Unlike these studies, SmartAxe introduces the dedicated type of resource (i.e., event emitting statement) for cross-chain bridge scenario, as the most important information of cross-chain transaction is recorded by such event emitting statement.

Before associating the resources with security checks, SmartAxe first collects basic facts. Specifically, given the resources in bridge contracts $R = \{r_1, r_2, ..., r_{n1}\}$, for each resource r_i, SmartAxe utilizes path-sensitive analysis to compute all reachable paths $P = \{p_1, p_2, ..., p_{n2}\}$. Then, for each reachable path p_j, SmartAxe finds out the security checks $C = \{c_1, c_2, ..., c_{n3}\}$ of access control constraints along the path. Hence, An association between resource r_i and security check c_k on path p_j can be denoted as $Association(c_k, p_j, r_i)$.

Then, SmartAxe introduces the prior probability to represent the confidence level of the association $Association(c_k, p_j, r_i)$. A prior probability is a value between 0 and 1 representing our degree of belief in the association $Association(c_k, p_j, r_i)$. Furthermore, SmartAxe determines the prior probability by analyzing access control properties, e.g., control flow, data flow, and semantics between resources and access control checks. Inspired by prior work [El-Rewini et al. 2022], we summarize the dedicated association patterns between resources and security checks for cross-chain bridge contracts, as well as their corresponding prior probability, as illustrated in Table 2.

Detecting access control incompleteness. Given the extracted access control constraints of a certain bridge, SmartAxe identifies the bridge contracts containing CCVs of access control incompleteness and outputs related vulnerable functions, if one of the following aspects is detected.

(1). **Access control omission.** SmartAxe detects access control omission by comparing the extracted access control constraints with the security check model (Table 1). If SmartAxe discovers the omission on the security checks of a certain category (or perspective), SmartAxe reports the corresponding vulnerable functions.

(2). **Access control violation paths.** Similar to prior works [Shao et al. 2016] which are designed for identifying inconsistency access control policies of Android applications, this aspect is used for identifying access control violation paths that allow users without sufficient permission to access sensitive resources. For the entry points of bridge contracts, SmartAxe compares their sub-control-flow graphs pairwise to identify possible paths that can reach the same sensitive resources but enforce different security checks (e.g., one with checks and the other without). SmartAxe outputs all the vulnerable functions on the access control violation paths.

4.3 Cross-Bridge Semantic Inconsistency Identification

In this subsection, SmartAxe constructs the xCFG and xDFG by aligning and connecting the single-chain control flow graphs. Further, based on the constructed graph, SmartAxe identifies the CCV of semantic inconsistency.

Graph Construction. The xCFG constructed by SmartAxe can be denoted as $G_c = (N_c, E_c, X_e)$. Specifically, SmartAxe encodes the following information: (1) the nodes of xCFG is a set of basic block nodes that represent the operations of the program, a relayer node that represents cross-chain data transmission, and a client node that represents the client of the cross-chain bridge. Here, N_b denotes the basic block nodes, N_r denotes the relayer node, and N_l denotes the client node. Therefore, we have $N_c := \{N_b \cup N_r \cup N_l\}$; (2) the edges of xCFG are composed of control flow edges E_f, emitting edges E_e and informing edge E_i. Here, E_e denotes the information flows that the relayer and client watch the emitted events, i.e., the relayer watches the deposit event emitted on the source chain or the client watches the withdrawal event on the destination chain. And E_i denotes the information flows that the relayers inform the contracts of the destination chain to perform authorization and withdrawal. Similarly, we also have $E_c := \left\{E_f \cup E_e \cup E_i\right\}$; (3) $X_e(E_c) \rightarrow \{CF, Emitting, Informing\}$ is a labeling function that maps an edge to one of the three types.

In addition, SmartAxe constructs the xDFG by performing the data flow analysis on the xCFG. Hence, in terms of data structure, xDFG constructed by SmartAxe is similar to the traditional data flow graph. The xDFG can be denoted as $G_d = (N_d, E_d)$. Here, N_d denote the different program operation of bridge contracts, and E_d refers to the data dependencies between program operations.

To construct the above-illustrated graphs, SmartAxe utilizes two key steps for cross-chain inter-procedure analysis.

SmartAxe constructs xCFG by adding emitting edges and informing edges to single-chain control flow graphs. Specifically, when adding emitting edges, to represent that the relayer watches the deposit event emitted on the source chain, SmartAxe searches for the emitting statement of deposit and lock event on the source-chain control flow graph as the source of emitting edge, makes the relayer nodes as the target of emitting edge, and connect them with the directed edge. Similarly, to represent that the client watches the withdrawal event on the destination chain, SmartAxe connects another emitting edge. For such an edge, the source is the emitting statement of authorization and withdrawal event on the destination-chain control flow graph, and the target is the client nodes. When adding the informing edge, to represent that the relayers inform the contracts of the destination chain for authorization and withdrawal, SmartAxe makes the relayer node as the source, further searches for the authorization statement as the target of the informing edge, and connects them together with the directed edge.

Then, SmartAxe constructs the xDFG through our proposed dedicated data flow analysis. Similar to traditional data flow analysis, SmartAxe performs the forward data flow analysis for control flow edges. Unlike the traditional data flow analysis, for emitting edges, only the arguments of the event can propagate through the emitting edges forward, as only these arguments are recorded in cross-chain data that transmits to the relayer. In terms of informing edges, only the arguments invoked by the authorization method can propagate through the informing edge forward.

Cross-bridge Semantic Inconsistency Detection. Given the constructed xCFG and xDFG of a certain bridge, SmartAxe identifies the bridge contracts containing CCVs of semantic inconsistency. and outputs related vulnerable functions, if one of the following aspects is detected.

(1). **Semantic granularity check.** The granularity insufficiency of deposit events causes destination-chain contracts cannot distinguish the different types of deposit and enter the same withdrawal logic. On xCFG, SmartAxe detects this aspect by comparing the cross-chain paths pairwise to identify possible paths that converge to the same withdrawal logic on the destination chain but enforce different deposit logic on the source chain. SmartAxe outputs all the vulnerable functions on the paths containing insufficiently-grained deposit events.

(2). **Semantic integrity check.** The feature of parse error is that the amount or type of withdrawal depends on the withdrawal function itself rather than the source-chain deposit. On xDFG, SmartAxe detects such an aspect by identifying whether the state variables of withdrawal have the data-flow dependency on state variables of deposit. If SmartAxe finds the lack of data-flow dependency between them, SmartAxe reports corresponding functions as vulnerable.

4.4 Vulnerability Trace Discovery

In this subsection, SmartAxe model the vulnerable functions of access control incompleteness and cross-bridge semantic consistency as CCV indicators, and analyze the accessibility (i.e., entry trace) and subversiveness (i.e., affected state variables) for CCV indicators. Finally, SmartAxe reports vulnerability traces which contain function calls from the vulnerable function to tainted state variable(s).

While vulnerable functions are well identified by Section 4.2 and 4.3, SmartAxe still require satisfying the following condition for locating CCVs.

Condition-1: Finding the entry trace of external call. With the CCV indicator, SmartAxe identifies the entry traces for each CCV indicator. This process is modeled as a process that SmartAxe utilizes taint propagation to detect whether the CCV indicators can be tainted by external attackers. Specifically, SmartAxe makes the taint propagate from the entry point (e.g., public function) of the cross-chain bridge contracts, and further detects whether the taint can reach the CCV indicators.

Condition-2: Finding the state variables affected by CCVs. After finding the entry trace of external attackers, SmartAxe continues to conduct the forward propagation on xDFG, and identify the state variables affected by the CCV. The reason for this step is to find out the state variables that are affected by the subverted flows of the CCV. Lastly, SmartAxe reports the tainted functions and state variables as various vulnerable traces that reveal how the attackers can leverage the CCV.

SmartAxe performs taint propagation via the method proposed in SmartState [Liao et al. 2023]. The taint sources can be divided into two types: the parameters passed by contract callers and the

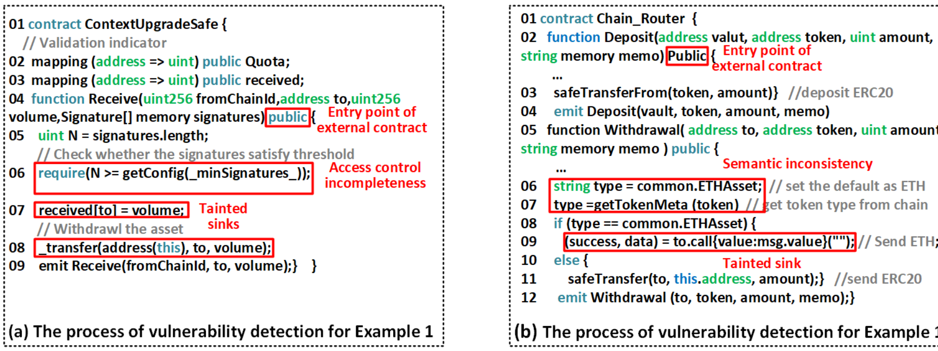


Fig. 6. The process of vulnerability detection for motivating examples in Figure 2.

parameters of public functions. The taint sinks of SmartAxe consist of either those external calls or state variables of the smart contracts (including the CCV indicators and client node). More detailed information regarding the taint sources and sinks is summarized in table 3.

Again, we take the motivating example of Figure 2 as an instance to show the CCV discovery, and the process is shown in Figure 6. For the contract in the left example, SmartAxe identifies the function *Receive* contains access control incompleteness as the CCV indicator, because *Receive* omits the validation check for signatory (line 6 and 7). Then, SmartAxe searches for the Condition-2, and identifies that function *Receive* is the entry point of the execution paths, as *Receive* is a public function that can be accessed by an external attacker. Lastly, SmartAxe investigates the Condition-3 by screening for state variables affected by the vulnerability through taint analysis. As a result, SmartAxe reports the vulnerability in contract *ContextUPgradeSafe* as follows, $Receive \rightarrow \_transfer \rightarrow \{received, balance\}$.

Similarly, SmartAxe identifies vulnerability in contract *Chain_Router* of right example as follows, $Deposit \rightarrow Withdrawal \rightarrow \{ETHbalance\}$

Table 3. EVM instructions defined as taint sources and taint sinks by SmartAxe.

	Type	EVM Instruction or Keyword or Statement
Source	(1) Parameter passed by user	CALLDATALOAD, CALLDATACOPY, CALLER, ORIGIN, CALLVALUE, CALLDATASIZE
	(2) Parameter of public function	Public, External
Sink	(1) External calls	CALL, CALLCODE, STATICCALL, DELEGATECALL
	(2) State variables	SSTORE, BALANCE, ADDRESS, CCV indicators, Client node

5 EVALUATION

In this section, we first introduce the evaluation setup and two datasets used for evaluation (i.e., manually-labeled CCV dataset and large-scale dataset). Then, we show the effectiveness of SmartAxe and its individual components by evaluating SmartAxe in terms of precision and recall on the manually-labeled CCV dataset. Lastly, we evaluate the real-world performance of SmartAxe by conducting the analysis on the large-scale dataset, and identified new CCVs in the wild.

5.1 Implementation and Evaluation Setup

We implement SmartAxe with around 3200 Line-of-Code in Python 3.8.10. Then, we conduct all the evaluation experiments for SmartAxe on a Ubuntu 20.04 server, which is equipped with the Intel i9- 10980XE CPU (3.0GHz), RTX3090 GPU, and 250 GB RAM.

Dataset and Ground-truth Establishment. We collect the following datasets for the evaluation experiment.

Manually-labeled CCV dataset (D_{manual}). We establish the ground truth for evaluating the effectiveness of SmartAxe in this dataset. Specifically, we collect 16 vulnerable cross-chain bridge applications (i.e., a total of 203 smart contracts) according to 20 real CCV attacks reported by public news and reports. Then, we annotate CCVs in the bridge contract by reviewing such attack reports. Specifically, each CCV is determined by two essential parts, (1) a vulnerable smart contract and (2) one or multiple vulnerable traces trigger the vulnerability. Here, a vulnerable trace contains function calls from the vulnerable function to tainted state variable(s) that can be affected by external call(s). In this way, we annotate a total of 88 CCV traces for 16 vulnerable bridge applications. To avoid bias, we invite three domain experts for the manual annotation, each expert separately performs the annotation. Only the vulnerability agreed by all three experts is confirmed as a valid CCV. To the best of our knowledge, this is the most comprehensive collection of CCVs from public sources.

Large-scale dataset (D_{large}). We collect the large-scale dataset to show the effectiveness of SmartAxe in identifying CCVs in the wild. The large-scale dataset contains 129 cross-chain bridge applications (i.e., a total of 1703 smart contracts). Specifically, we searched for the cross-chain bridge project in the community and Internet exhaustively, and finally looked up a total of 148 cross-chain bridge applications. Further, we manually confirm these applications, and find out both the source code and bytecode for 129 cross-chain bridge applications among them (i.e., the coverage rate is over 87%). Finally, we comb the source code and bytecode of these 129 cross-chain bridge applications and obtain the large-scale dataset. Our collection actually covers the mainstream of cross-chain bridges. For example, 86 of 129 cross-chain bridges are contained in the top 100 cross-chain bridges ranked by the Chainspot [Chainspot 2023]. To the best of our knowledge, this is the most comprehensive collection of cross-chain bridge contracts from public sources.

Evaluation Metrics. We lay out the following research questions (RQs) for the evaluation experiments.

- RQ1. How does SmartAxe perform in detecting CCVs?
- RQ2. How effective is SmartAxe in finding access control incompleteness?
- RQ3. How effective is SmartAxe in finding cross-bridge semantic inconsistencies?
- RQ4. Can SmartAxe detect CCVs from real-world cross-chain bridge applications?

5.2 Effectiveness of SmartAxe

To answer RQ1, we run SmartAxe over the manually-labeled CCV dataset and evaluate its precision and recall rate. Particularly, we give the same time budget (i.e., 10-mins timeout following prior works [Liao et al. 2023, 2022]) for every smart contract in the dataset. More specifically, we compute the precision and recall rate by manually comparing the results reported by SmartAxe with the ground truth of D_{manual} (i.e., 88 CCVs in 203 cross-chain bridges). Note that we do not conduct comparison experiments, as SmartAxe is the first CCV detection approach for cross-chain bridge contracts.

Table 4. Overall effectiveness for SmartAxe on the Manual-labelled CCV Dataset (D_{manual}).

CCV	Precision			Recall		
	TP	FP	rate	TP	FN	rate
Access control incompleteness	54	10	84.38%	54	5	91.53%
Semantic inconsistency	25	4	85.71%	25	4	86.21%
Total	79	14	84.95%	79	9	89.77%

Table 4 demonstrates the precision and recall of SmartAxe. As can be seen, SmartAxe achieves high precision (i.e., 84.95%) and recall (i.e., 89.77%) for CCV detection. To this end, we can conclude that SmartAxe can identify CCVs for cross-chain bridge contracts effectively.

False Positives and False Negatives. We manually investigate all the false positives and false negatives reported by SmartAxe. Our investigation result shows that most of the 14 false positives are caused by the limitation of basic facts produced by SmartDagger (i.e., the basic control-flow analyzer in SmartAxe). For instance, SmartDagger is insufficiently precise in recovering the type and semantics of state variables for smart contracts, which causes SmartAxe to report false positives. To overcome these false positives, SmartAxe can integrate a more advanced analyzer to improve the effectiveness of type and semantic recovery. In terms of false negatives, the reason for them is that SmartAxe ignores a tiny amount of access control that relies on the on-chain query (e.g., checking ownership of on-chain asset). Actually, such problems are unable to be solved through a static analysis approach like ours, because they require real-time on-chain data.

5.3 Impact of Security Check Modeling and Resource Association

To answer RQ2, we evaluate the effectiveness of the security check modeling and resource association respectively.

Effectiveness of Security Check Modeling. As illustrated in Section 4.2, security check modeling is the important advantage of SmartAxe. The security check modeling helps for covering the diverse security checks and facilitates the discovery for access control omission effectively, so SmartAxe can utilize such advantage to identify more vulnerable traces. The security check modeling ensures soundness (i.e., avoiding false negatives) for vulnerability analysis. To evaluate this, we compare the recall rate of SmartAxe with two state-of-the-art tools (i.e. Ethainter [Brent et al. 2020], Achecker [Ghaleb et al. 2023]). Note that we do not compare SmartAxe with SPCon, as AChecker has been proven to perform better than SPCon in prior work [Ghaleb et al. 2023]. We ran SmartAxe, Ethainter, and AChecker over the manually-labeled dataset D_{manual} to compare their recall rate.

Table 5. Effectiveness of security check modeling with other SOTA static analyzers over D_{manual}.

Approach	Ethainter [Brent et al. 2020]			AChecker [Ghaleb et al. 2023]			SmartAxe		
	TP	FN	Recall	TP	FN	Recall	TP	FN	Recall
CCV	6	82	6.82%	3	85	3.41%	79	9	89.77%

Table 6. Comparison results between SmartAxe and SmartAxe without resource allocation on the Manual-labelled CCV Dataset (D_{manual}), for evaluating the effectiveness of resource allocation

Approach	SmartAxe w/o resource allocation			SmartAxe		
	TP	FP	Precision	TP	FP	Precision
CCV	79	23	77.45%	79	14	84.95%

As can be seen in table 5, the recall rate of SmartAxe is much higher compared with the other two tools. To identify why SmartAxe performs better, we manually inspect all the false negatives for the other two state-of-the-art tools. Particularly, the manual inspection results demonstrate that, most false negatives are caused by the limitation of these state-of-the-art tools, as they cannot overcome the diversity of security checks in cross-chain bridge contracts. In contrast, SmartAxe can utilize the proposed normalized access control model (i.e., Table 1) to avoid these false negatives. To illustrate, we take the motivating example of Figure 2 (a) as an instance again, which has been discussed in Section 2.2 earlier. Noting that Ethainter and AChecker produce false negatives for Figure 2, but SmartAxe can avoid such false negatives by security check modeling. With the security check model of Table 1, SmartAxe identifies that function *Receive* omits the validation check on signatory by comparing function *Receive* with perspective P4 of Table 1, and reports a CCV of access control incompleteness.

Effectiveness of Resource Allocation. Another advantage of SmartAxe is that SmartAxe reduces the false positives by associating resources with the security checks. To evaluate this, we compare the SmartAxe with SmartAxe without resource allocation, i.e., SmartAxe without resource association. To evaluate the effectiveness of resource association. We ran the SmartAxe without resource allocation and SmartAxe over manually-labeled dataset D_{manual}.

Table 6 presents the precision of SmartAxe and SmartAxe without resource allocation. Due to unloading the resource allocation method, the precision of the SmartAxe without resource allocation is only 77.45%, and the SmartAxe without resource allocation reports more false positives (i.e., 11 new false positives). To this end, we can summarize that the resource allocation method helps SmartAxe improve the precision for CCV detection.

Further, we manually inspect each false negative reported by the SmartAxe without resource allocation. The inspection results demonstrate that 11 of 23 false positives (i.e., 47.83%) can be eliminated by utilizing the resource allocation method, which are missed by the SmartAxe without resource allocation. To illustrate, we take the motivating example of Figure 4 as an instance again, which has been discussed in Section 3.1 earlier. SmartAxe without resource allocation produces false positives for Figure 4, and report function *withdrawal* omits validation check for signatory, as it cannot identify the association between the security check of line 8 and resource of line 12. With the probabilistic inference pattern of resource allocation in Table 2, SmartAxe identifies that security checks of line 5 and 8 should be associated with the resource of line 12, and further eliminate such false positives.

5.4 Impact of Graph Construction

As illustrated in Section 4.3, another advantage possessed by SmartAxe is our constructed comprehensive xCFG and xDFG, which facilitate the detection for CCV of semantic inconsistency.

Hence, the constructed graphs help for performing more semantic inconsistency discovery and taint tracking, so that it can identify more vulnerability traces. Therefore, the effectiveness of graph construction reflects on the recall rate. Similarly, we compare the SmartAxe with SmartAxe without xCFG and xDFG construction, to evaluate the effectiveness of graph construction. We ran all of these tools on the manually-labeled dataset (D_{manual}) to evaluate their recall.

Table 7 shows the comparison results between SmartAxe and the SmartAxe without graph construction. Due to ignoring graph construction, the recall rate of the SmartAxe without graph construction is only 65.91% and drops rapidly. In contrast, SmartAxe presents a better performance (i.e., 89.77%). In conclusion, the constructed xCFG and xDFG help SmartAxe improve the recall of CCV detection effectively.

Table 7. Impact of graph construction over D_{manual}.

Approach	SmartAxe w/o xACG and xDFG			SmartAxe		
	TP	FN	Recall	TP	FN	Recall
CCV	56	32	65.91%	79	9	89.77%

In addition, we manually investigate all the false negative results for SmartAxe without graph construction. The manual analysis results show that 23 of 32 false negatives actually can be eliminated through performing end-to-end analysis on xCFG and xDFG, which are missed by the SmartAxe without graph construction.

5.5 Cross-Chain Vulnerability in Real world

To answer RQ4, we run SmartAxe over the large-scale dataset (D_{large}) to evaluate the real-world performance of SmartAxe. Our domain experts manually inspected all the reported results via majority voting, and finally confirmed that SmartAxe detects 232 new CCV from 129 cross-chain bridge applications (i.e., affecting 126 smart contracts). To detail, SmartAxe outputs 324 warnings. Among them, our manual investigation showed that 278 are true positives and the rest 46 are false positives. By inspecting the amount of assets affected by these CCV, we find that these 232 new CCVs affect a total asset of 1,885,250 USD as of our paper submission. Figure 8 presents the top 5 cross-chain bridges in terms of assets affected by CCV. For more detail, we discuss two case studies for illustration.

Case study 1. at *0x915861959D2feBCCF37795Fd93c6094DdeBf34Bd*. This smart contract is from a real-world bridge ranking in the top 40 cross-chain bridges [Chainspot 2023]. However, such a smart contract contains a CCV of access control incompleteness. Specifically, this contract implements the authorization and withdrawal via two functions, i.e., function *saveWithdrawNative* for native tokens of the bridge and function *saveWithdrawAlien* for alien tokens outside the bridge. Unfortunately, both two functions omit the security checks on token type (i.e., native token or alien token). Hence, the deposit record of the native token can unexpectedly pass the authorization of function *saveWithdrawAlien*, and cause the alien token to be withdrawn, and it is the same the other way around. For this case, by comparing functions (i.e., *saveWithdrawNative* and *saveWithdrawAlien*) with perspective P3 of Table 1, SmartAxe effectively identifies them as vulnerable functions and utilizes the taint analysis to determine that state variable *balance* is manipulated by the CCV.

Case study 2. at *0x2d6775C1673d4cE55e1f827A0D53e62C43d1F304*. This smart contract is from a real-world bridge, which ranks in the top 30 cross-chain bridges [Chainspot 2023]. Particularly, this contract also involves a CCV of access control incompleteness. Specifically, such a contract implements the deposit of liquidity via function *preFill*. Particularly, *preFill* receives a calldata argument *_message* which contains the sender, destination, and amount of the deposit. Unfortunately, SmartAxe omits the validation check on such an argument. Hence, an attacker can construct the fake *_message* to obtain a malicious deposit. For this case, by comparing functions (i.e., *preFill*) with perspective P2 of Table 1 , SmartAxe effectively identifies function *preFill* as vulnerable functions and utilizes the taint analysis to determine that state variable *balance* and *liquidityProvider* is manipulated by the CCV.

Table 8. Top 5 cross-chain bridge in terms of asset affected by CCV

Bridge	Hop.Exchanxx bridge	Terxx Bridge	Sifchxxx Bridge	RenBridxx	Ocuxx Bridge
Vulnerability number	4	6	1	3	3
Affected asset	1445827	28038.67	16743.31	12896.52	8093.08

5.6 Discussion and Limitation

SmartAxe shares the following advantages in CCV detection: (1) As demonstrated in evaluation, SmartAxe is obviously effective in locating CCVs for cross-chain bridge contracts, which lack of prior works can support [Zheng et al. 2023]; (2) SmartAxe establishes comprehensive extraction for access control constraints, and cover the diverse and complex access control for cross-chain bridge contract effectively, which overcome the limitation of prior works [Brent et al. 2020; Ghaleb et al. 2023; Liu et al. 2022]; (3) SmartAxe propose unique a graph construction (i.e., xCFG and xDFG) to locate the CCVs of semantic inconsistency effectively, which lack of prior works can support. With the above-illustrated advantages, SmartAxe can identify CCV precisely and comprehensively. All of the developers, participants, and third-party authorities can utilize SmartAxe to investigate the security of cross-chain bridge contracts.

While SmartAxe leverages predefined patterns (Table 1) and probability values (Table 2), it remains a highly generalizable framework for CCV detection. This is attributed to the following reasons: (1) The predefined patterns are derived from three fundamental workflows of the bridge, namely asset deposit, cross-chain communication, and asset withdrawal. These security checks are essential and universally applicable across all bridges. (2) In terms of comprehensiveness, our investigation covers the top 100 bridges, which represent 67.57% (100/148) of bridges in existence, thereby affecting over 90% of all cross-chain transactions. Given the prevalence and significance of these top bridges, the proposed patterns adequately address most CCVs. (3) The probability values are derived from the control-flow and data-flow dependencies inherent in the bridge contract. As all bridges rely on similar dependencies (such as reachability and variable dependency) to facilitate resource protection, the probability values hold true across all bridges.

Responsible disclosure. To prevent real economic damage to the bridge contracts, we anonymize the reported results in this paper and only showcase the Top 5 cross-chain bridges in terms of assets affected by our confirmed CCVs, as illustrated in Table 8.

We are gradually reporting the identified CCVs to bridge developers, a process that is time-consuming and nontrivial. Confirming such CCVs ourselves poses significant challenges: (1) Conducting on-chain testing for vulnerability confirmation may lead to economic loss or damage. (2) Off-chain simulation for confirming CCVs necessitates setting up the entire cross-bridge infrastructure (e.g., relayers and underlying protocols), demanding substantial engineering effort. Presently, all reports are awaiting acknowledgment by the corresponding parties such as project developers.

Threats to validity. Below we discuss the soundness and completeness of individual components of SmartAxe. (1) For security check extraction, the completeness of SmartAxe is threatened by the insufficiently precise basic fact (produced by SmartDagger); (2) For the resource allocation, the soundness of SmartAxe is influenced and subverted by the uncertainty of the probabilistic method. (3) The semantic inconsistency identification and vulnerability trace location part of SmartAxe are sound and complete, because they neither introduce false information nor miss valid information.

Note that SmartAxe currently lacks support for analyzing non-EVM chains, as its core control-flow analyzer, SmartDagger, is specifically designed for EVM chains. Nevertheless, SmartAxe offers the flexibility to integrate alternative analyzers tailored for non-EVM chains to extend its capabilities in supporting such networks. Importantly, the effectiveness of SmartAxe remains unaffected by this limitation for the following reasons: (1) The vast majority of bridge Dapps predominantly operate on or support EVM chains, as evidenced by prior research [Lee et al. 2023]. (2) The smart contracts deployed by bridge Dapps across their supported chains, including both EVM and non-EVM chains, exhibit consistent program logic. Therefore, the absence of support for non-EVM chains does not compromise the efficacy of SmartAxe in detecting vulnerabilities.

6 RELATED WORK

Smart Contract Vulnerability Detection. In recent years, many security tools have been proposed to detect vulnerabilities in smart contracts. They can be divided into static analysis tools and dynamic analysis tools. For example, static tools include Oyente [Luu et al. 2016], MadMax [Grech et al. 2018], Securify [Tsankov et al. 2018], Zeus [Kalra et al. 2018], Clairvoyance [Xue et al. 2020], SailFish [Bose et al. 2022], eTainter [Ghaleb et al. 2022], SmartState [Liao et al. 2023], etc. Other tools such as ContractFuzzer [Jiang et al. 2018], Harvey [Wüstholz and Christakis 2020], Echidna [Grieco et al. 2020], sFuzz [Nguyen et al. 2020], SMARTIAN [Choi et al. 2021], and ItyFuzz [Shou et al.

2023] belong to dynamic analysis or testing. As for access control in smart contracts, early research like Mythril [Consensys 2017] only focuses on unprotected Ether withdrawal and specific instructions. While recent works such as Ethainer [Brent et al. 2020], SPCon [Liu et al. 2022], and AChecker [Ghaleb et al. 2023] consider the access control policy or model, they are limited to analyzing contracts of a single address. Therefore, These frameworks cannot identify access control constraints of cross-chain bridges that involve multiple addresses. As all these tools do not consider cross-chain analysis, they are not effective in detecting CCVs.

Cross-chain Attack Detection. Our work is also closely related to various cross-chain attack detection mechanisms. Cross-chain systems have become necessary infrastructures for decentralized applications [Lee et al. 2023]. Previous work mainly focuses on the security of designing a cross-chain system, such as zkBridge [Xie et al. 2022] and CrossLedger [Vishwakarma et al. 2023]. In addition, Duan et.al [Duan et al. 2023] and Lee et.al [Lee et al. 2023] conduct the survey on cross-chain attacks and propose advice on designing cross-chain systems. These studies discuss the severity of cross-chain attacks and vulnerabilities. To the best of our knowledge, the most relevant work on identifying CCV is Xscope [Zhang et al. 2022]. It collects the transactions and status on different blockchains for existing attack detection with a set of security properties and patterns. Nonetheless, as smart contracts are hard to fix after deployment, such a framework cannot find vulnerabilities in contract code before deployment and further eliminate economic losses. Our proposed framework, SmartAxe, is the first of its kind to detect cross-chain vulnerabilities in smart contracts.

7 CONCLUSION

In this paper, we propose a new static analysis framework, SmartAxe, for locating cross-chain vulnerabilities in cross-chain bridge contracts. We evaluate SmartAxe over a manually-labeled dataset of 16 real attack cross-chain bridge applications (i.e., 203 smart contracts) and a large-scale dataset of 129 real-world cross-chain bridge applications (i.e., 1703 smart contracts). Evaluation results show that SmartAxe can identify cross-chain vulnerability effectively, with a high precision of 84.95% and a high recall of 89.77%. In addition, we find that 232 new CCVs exist in 126 on-chain smart contracts, affecting a total asset of 1,885,250 USD.

ACKNOWLEDGMENTS

The research was supported by the National Natural Science Foundation of China (62032025), and the Technology Program of Guangzhou, China (No. 202103050004).

REFERENCES

Yousra Aafer, Jianjun Huang, Yi Sun, Xiangyu Zhang, Ninghui Li, and Chen Tian. 2018. AceDroid: Normalizing Diverse Android Access Control Checks for Inconsistency Detection.. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.*

Rob Behnke. 2022. The nomad bridge hack: a deeper dive. https://www.halborn.com/blog/post/the-nomad-bridge-hack-a-deeper-dive. [Accessed 20-Sep-2023].

Binance. 2022. BNB Chain. https://www.bnbchain.org/. [Accessed 20-Sep-2023].

Bitcoin. 2009. Bitcoin. https://bitcoin.org/. [Accessed 20-Sep-2023].

Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds.. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022.* 161–178. https://doi.org/10.1109/SP46214.2022.9833721

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*. ACM, London UK, 454–469. https://doi.org/10.1145/3385412.3385990

CelerNetwork. 2022. A DNS cache poisoning attack on cBridge. https://twitter.com/CelerNetwork/status/1560123830844411904. [Accessed 20-Sep-2023].

Chainspot. 2023. Chainspot. https://chainspot.io/. [Accessed 20-Sep-2023].

Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Melbourne, Australia, 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

Consensys. 2017. Mythril. https://github.com/Consensys/mythril. [Accessed 20-Sep-2023].

Li Duan, Yangyang Sun, Wei Ni, Weiping Ding, Jiqiang Liu, and Wei Wang. 2023. Attacks Against Cross-Chain Systems and Defense Approaches: A Contemporary Survey. *IEEE/CAA Journal of Automatica Sinica* 10, 8 (Aug. 2023), 1647–1667. https://doi.org/10.1109/JAS.2023.123642

Zeinab El-Rewini, Zhuo Zhang, and Yousra Aafer. 2022. Poirot: Probabilistically Recommending Protections for the Android Framework.. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. 937–950. https://doi.org/10.1145/3548606.3560710

Ethereum. 2015. Ethereum. https://www.ethereum.org/. [Accessed 20-Sep-2023].

Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts.. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. 8–15. https://doi.org/10.1109/WETSEB.2019.00008

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*. ACM, Virtual South Korea, 728–739. https://doi.org/10.1145/3533767.3534378

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)*. IEEE, Melbourne, Australia, 945–956. https://doi.org/10.1109/ICSE48619.2023.00087

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-Gas Conditions in Ethereum Smart Contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.

Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts.. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. 557–560. https://doi.org/10.1145/3395363.3404366

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*. ACM, Montpellier France, 259–269. https://doi.org/10.1145/3238147.3238177

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

Sung-Shine Lee, Alexandr Murashkin, Martin Derka, and Jan Gorzny. 2023. SoK: Not Quite Water Under the Bridge: Review of Cross-Chain Bridge Hacks. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, Dubai, United Arab Emirates, 1–14. https://doi.org/10.1109/ICBC56567.2023.10174993

Zeqin Liao, Sicheng Hao, Yuhong Nan, and Zibin Zheng. 2023. SmartState: Detecting State-Reverting Vulnerabilities in Smart Contracts via Fine-Grained State-Dependency Analysis.. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*. 980–991. https://doi.org/10.1145/3597926.3598111

Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: A Bytecode-Based Static Analysis Approach for Detecting Cross-Contract Vulnerability.. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. 752–764. https://doi.org/10.1145/3533767.3534222

Ye Liu, Yi Li, Shang-Wei Lin, and Cyrille Artho. 2022. Finding Permission Bugs in Smart Contracts with Role Mining.. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. 716–727. https://doi.org/10.1145/3533767.3534372

Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart Contract Vulnerability Detection: From Pure Neural Network to Interpretable Graph Feature and Expert Pattern Fusion. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*. International Joint Conferences on Artificial Intelligence Organization, Montreal, Canada, 2751–2759. https://doi.org/10.24963/ijcai.2021/379

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*. ACM, Vienna Austria, 254–269. https://doi.org/10.1145/2976749.2978309

Poly Network. 2020. PolyNetwork. https://www.poly.network/. [Accessed 20-Sep-2023].

Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 778–788. https://doi.org/10.1145/3377811.3380334

Polygon. 2023. Polygon Bridge. https://chainspot.io/bridge/polygon-bridge. [Accessed 20-Sep-2023].

Rubic. 2022. Rubic admin wallet addresses was compromised. https://twitter.com/CryptoRubic/status/1587704548688367619. [Accessed 20-Sep-2023].

Sam Cooling. 2021. ChainSwap hackers steal $8m and crash token prices. https://finance.yahoo.com/news/chainswap-hackers-steal-8m-crash-121056965.html. [Accessed 20-Sep-2023].

Sebastian Sinclair. 2021. Blockchain Protocol Thorchain Suffers $8M Hack. https://www.coindesk.com/markets/2021/07/23/blockchain-protocol-thorchain-suffers-8m-hack/. [Accessed 20-Sep-2023].

Yuru Shao, Qi Alfred Chen, Zhuoqing Morley Mao, Jason Ott, and Zhiyun Qian. 2016. Kratos: Discovering Inconsistent Security Policy Enforcement in the Android Framework.. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.*

Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract.. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023.* 322–333. https://doi.org/10.1145/3597926.3598059

Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts.. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 67–82. https://doi.org/10.1145/3243734.3243780

Lokendra Vishwakarma, Amritesh Kumar, and Debasis Das. 2023. CrossLedger: A Pioneer Cross-Chain Asset Transfer Protocol. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid).* IEEE, Bangalore, India, 568–578. https://doi.org/10.1109/CCGrid57682.2023.00059

Wikipedia. 2023. Non fungible token. https://en.wikipedia.org/wiki/Non-fungible_token. [Accessed 20-Sep-2023].

Wikipedia contributors. 2022. Poly Network exploit. https://en.wikipedia.org/wiki/Poly_Network_exploit. [Accessed 20-Sep-2023].

Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts.. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020.* 1398–1409. https://doi.org/10.1145/3368089.3417064

Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. 2022. zkBridge: Trustless Cross-Chain Bridges Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security.* ACM, Los Angeles CA USA, 3003–3017. https://doi.org/10.1145/3548606.3560652

Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.* ACM, Virtual Event Australia, 1029–1040. https://doi.org/10.1145/3324884.3416553

Jiashuo Zhang, Jianbo Gao, Yue Li, Ziming Chen, Zhi Guan, and Zhong Chen. 2022. Xscope: Hunting for Cross-Chain Bridge Attacks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* ACM, Rochester MI USA, 1–4. https://doi.org/10.1145/3551349.3559520

Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. *CoRR* abs/2311.10372. https://doi.org/10.48550/ARXIV.2311.10372

Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. 2020. An overview on smart contracts: Challenges, advances and platforms. *Future Gener. Comput. Syst.* 105 (2020), 475–491. https://doi.org/10.1016/J.FUTURE.2019.12.019

Received 2023-09-29; accepted 2024-01-23