Learning from the Past: Real-World Exploit Migration for Smart Contract PoC Generation

Kairan Sun*, Zhengzi Xu†§, Kaixuan Li*, Lyuye Zhang*, Yebo Feng*, Daoyuan Wu‡, and Yang Liu*

* Nanyang Technological University, Singapore
† Imperial College London, Imperial Global Singapore, Singapore
‡ Lingnan University, Hong Kong SAR, China

Abstract—Smart contract vulnerabilities continue to cause significant financial losses, despite the implementation of security measures such as manual audits and bug bounty platforms. A critical component often required by these security measures is the proof-of-concept (PoC) exploit, which validates vulnerability exploitability, assesses impact severity, and guides developers in fixes. Existing tools have explored automated PoC generation with techniques like symbolic execution, fuzzing, and program synthesis. However, these approaches frequently fail to generate PoCs for vulnerabilities exploited in real-world incidents, primarily due to their limitations in handling complex transaction dependencies, navigating vast on-chain state spaces, or requiring extensive manual specifications. Our migration-based approach extracts critical information from documented security incidents and applies it to generate PoCs for similar vulnerable code. This approach leverages proven exploit patterns rather than generating PoCs from scratch. This approach is motivated by two key observations: the prevalence of code reuse in smart contracts (up to 90% at the function level) and the increasing availability of documented PoCs for real-world incidents. Our approach operates in three phases: *(1)* abstracting essential components (i.e., environment properties, attack logic, and verification checks) from existing PoCs into templates, *(2)* given a new target contract, selecting suitable templates with adapted values through clone-detection and property-feasibility analysis, and *(3)* generating and validating PoCs in simulated environments. Our evaluation demonstrates effectiveness and efficiency across multiple scales. Our approach successfully generates valid PoCs for 62 out of 67 manually validated cases without false positives and completes analysis in 3.8 hours compared to 133.2 and 210.5 hours required by existing tools. Large-scale evaluation on 979,512 contracts identifies 256 vulnerable contracts across blockchain networks with 64 cross-chain cases, demonstrating real-world applicability.

Index Terms—Smart Contract Security, Proof-of-Concept Exploits, Real-World Incidents

I. INTRODUCTION

Blockchain provides fundamental infrastructure for cryptocurrencies like Bitcoin, managing over $3.9 trillion in assets [19]. Smart contracts are programs that execute on blockchain platforms and serve as the actual applications users interact with. Successful exploitation of contract vulnerabilities (e.g., unauthorized token transfer, reentrancy, and price manipulation) can result in immediate, irreversible financial losses, with more than $2.2 billion in 2024 alone [32]. Although traditional security measures like code audits provide a foundational layer of protection, they can be slow and resource-intensive. Security platforms such as Code4rena [2]

§Zhengzi Xu is the corresponding author (z.xu@imperial.ac.uk).

and Immunefi [7] encourage broader community participation in detecting vulnerabilities by offering monetary rewards for bug reports. However, regardless of whether vulnerabilities are identified during formal audits or reported through these open platforms, a proof-of-concept (PoC) exploit remains an essential requirement [1], [8]. PoCs play important roles not only in confirming the exploitability of vulnerabilities but also in assessing their severity and potential impact. This also aids developers in prioritizing fixes, thereby improving security practices and contributing to the broader understanding of vulnerability exploitation patterns within the community.

Despite the critical role of PoCs, creating them remains largely a manual process that demands considerable domain expertise and suffers from poor scalability. To reduce the manual effort required, researchers and practitioners have explored automated approaches, primarily through static analysis and symbolic execution [29], [4], [31], fuzzing techniques [38], [41], [33], and program synthesis [42], [48]. However, these automated approaches have shown limited success in generating PoCs for vulnerabilities exploited in real-world incidents [43], [46]. Static analysis and symbolic execution-based approaches often struggle with complex inter-contract dependencies and state transitions. Fuzzing approaches often face challenges in achieving adequate coverage of complex state spaces while incurring significant computational overhead. Program synthesis approaches, while promising for specific vulnerability types, require substantial manual effort in specifying attack goals and initial states, limiting their practical applicability. These limitations are particularly evident when dealing with vulnerabilities exploited in real-world incidents, as they often involve complex interactions and state conditions that are difficult to be discovered from scratch.

Given these limitations, migration-based solutions have emerged as a promising alternative in other automated software engineering domains. SEIGE [24] and VESTA [17] investigate migration-based test generation for Java libraries, while MAM [47] explores API mapping between Java and C#. Inspired by existing works, our work explores migration-based PoC generation which is motivated by two insights: ① Prevalence of code reuse in smart contracts. Recent studies [13], [40] reveal that smart contracts exhibit significant code reuse (up to 90% at the function level) through direct copying and protocol forking. This prevalent code reuse without sufficient security checks makes it possible for vulnerable functions to

propagate across multiple contracts and blockchain networks. Given that Ethereum alone manages over 79 million verified contracts (contracts with original source code and compiler settings available) [20], such vulnerability propagation could create a substantial attack surface. ② Growing availability of publicly documented PoCs for real-world incidents. Increasing transparency in the security community has led to a growing repository of PoCs, each targeting a successfully exploited contract in real-world incidents. These PoCs are valuable as they provide concrete evidence of successful exploitation patterns, yet their potential for systematic PoC generation remains largely unexplored by existing approaches.

Our work aims to address two critical security scenarios enabled by these insights. First, proactive contract auditing takes contract source code as input and outputs executable PoCs when vulnerabilities similar to previously exploited ones are identified, enabling security analysts to confirm exploitability before deployment. Second, post-exploit scanning takes known exploit PoCs and its target vulnerable function as input and outputs lists of similar vulnerable contracts with generated PoCs, enabling proactive vulnerability propagation analysis when major exploits occur throughout the ecosystem.

However, realizing these security scenarios requires overcoming the following challenges in automation. First, smart contract exploits for real-world incidents often require cross-contract interactions and specific blockchain state conditions to achieve successful exploitation. However, these exploits lack standardized format and contain hardcoded values that encode implicit exploit requirements, making it challenging to identify essential components and adapt them across different contract contexts. Second, even with effective abstraction and adaptation methods, practical deployment faces scalability challenges. When major exploits occur, attackers can hunt similar vulnerabilities across blockchain networks. Given the scale of verified contracts [20], post-exploit scanning requires rapidly processing millions of contracts before attackers exploit them, ensuring both effectiveness and efficiency at scale.

To address these challenges, we propose POCSHIFT, an automated migration-based PoC generation framework. *1)* PoC Abstraction: To address the first challenge, we identify three key components essential for migration-based generation (environment properties, attack logic, and validation checks). These components are derived from comprehensive analysis of existing PoCs, cross-referencing with recent studies on PoC generation [44], [38], [29]. We explore how to automatically identify these components and adapt them to new contexts by reverse engineering the PoC creation workflow in a systematic approach. *2)* Candidate Matching: To address the second challenge, we employ a two-stage filtering approach. We first perform hash-based filtering to quickly reduce the candidate pool, then conduct detailed feasibility analysis with only on filtered matches. This ensures computational efficiency by avoiding expensive ABI retrieval and analysis on the entire contract repository. *3)* Migration Test: Finally, with the selected template and adapted values, PoC will be generated and tested in an isolated simulation environment with validation

checks to validate the execution result.

Our evaluation demonstrates both the effectiveness and practical impact of our approach. While existing tools like ItyFuzz [38] and Mythril [4] achieve limited success with significant false positives, our migration-based approach achieves a 92% success rate with zero false positives. Furthermore, our tool completes PoC generation over 35 times faster than existing approaches. Large-scale evaluation on 979,512 contracts demonstrates real-world applicability, successfully generating and validating PoCs for 256 vulnerable contracts across multiple blockchain networks, including 64 cross-chain migrations. This improvement makes security assessment at scale possible with high precision, demonstrating the capability of our tool to generate PoCs for recurring vulnerabilities across diverse blockchain environments and protocols.

In summary, the main contributions of this work are:

- We propose POCSHIFT, a novel approach that automates PoC generation for smart contracts by migrating existing PoCs to contracts containing similar vulnerable functions.
- We evaluate POCSHIFT on 5,713 verified smart contracts and demonstrate its superior effectiveness and efficiency compared to state-of-the-art tools.
- We conduct large-scale experiments with POCSHIFT, which reveals 256 vulnerable contracts, including 64 cross-chain cases.

II. BACKGROUND AND MOTIVATION

A. Smart Contracts and States

Smart contracts are self-executing programs on blockchain platforms that automatically enforce encoded agreements. This work focuses on Solidity smart contracts on EVM-compatible chains, chosen for their widespread adoption and significant impact. A crucial aspect of smart contracts is that all their states are stored on-chain. These states are critical in determining contract behavior. Proper management of on-chain states is essential for contract security and functionality.

B. Proof of Concepts (PoCs) and Exploits

In this work, PoCs refer to *executable smart contracts that demonstrate vulnerability exploitation process*. While exploits typically aim to maximize profit, PoCs validate exploitability by demonstrating potential for profit or denial-of-service capabilities in real-world scenarios without pursuing maximum returns. We prioritize determining exploitability over financial impact assessment because confirming vulnerability existence and exploitability is the essential first step in security evaluation, regardless of potential profit magnitude.

C. Motivating Example

We analysis for two real-world incidents: Oynx Protocol [9] (May 2023), which suffered a loss of $2.1M in Ethereum; and Midas Capital [16] (June 2023), which was exploited on the BNB Smart Chain with a loss of $600K.

Both incidents exploited a vulnerability in the *redeemFresh* function, inherited from Compound V2 protocol [3]. Specifically, attackers could manipulate the exchange rate calculation

TABLE I: Summary of covered vulnerability categories in collected PoCs.

Vulnerability Category	Vulnerability Description
Price Manipulation (**PM**)	Enables attackers to alter asset prices within contracts, exploiting external dependencies.
Access Control (**AC**)	Enforces user access policies, often leading to unauthorized resource access or use.
Logic Flaw (**LF**)	Business logic errors that can result in incorrect states or token handling.
Reentrancy (**RE**)	Occurs when a contract is recursively called before its initial execution is complete.
Arithmetic (**AR**)	Includes numerical errors like overflow or underflow within contract operations.
Block Manipulation (**BM**)	Exploits blockchain elements such as timestamps to manipulate contract behavior.
Denial of Service (**DoS**)	Leads to service disruption when a contract is overloaded or misconfigured.
Other Vulnerabilities (**OT**)	Encompasses a range of uncommon or unique contract vulnerabilities.

Fig. 1: A decomposition example of Olympus DAO PoC [5].

(Equation (1)) by first depositing a minimal amount of tokens into an empty market, followed by significant donations of the same token to artificially inflate its market value. This manipulation altered the exchange rate and, compounded by rounding errors under low *totalSupply*, allowed the attackers to redeem more assets than their actual deposits.

$$\text{exchangeRate} = \frac{\text{totalCash} + \text{totalBorrows} - \text{totalReserves}}{\text{totalSupply}} \quad (1)$$

This recurring exploitation highlights a critical challenge in smart contract security: vulnerabilities can propagate across different protocols and blockchain networks through code reuse, leading to repeated exploitation and severe financial impact. While security audits are typically performed pre-deployment, many vulnerabilities exploited in real-world incidents emerge from complex runtime interactions between multiple on-chain contracts and state manipulations. These scenarios often exceed the capability of existing security tools. Moreover, the reuse of vulnerable code alone does not guarantee exploitability. Determining whether a vulnerability can be exploited requires extensive manual analysis of contract-specific implementations and runtime conditions.

Our work addresses these challenges by systematically extracting critical information from existing PoCs and adapting into new contexts. This approach eliminates repetitive manual analysis while accurately determining real-world exploitability of recurring vulnerabilities across blockchain ecosystems.

III. PoC Collection

To ensure the dataset suitable for migration-based PoC methodology, we conduct several filtering steps. We first assess each PoC for executable integrity, which leads to narrowing our dataset down to 320 PoCs. A PoC passes if it executes successfully and demonstrates the intended exploitation. The 83 filtered cases (403-320) fail due to compilation errors or invalid exploit outcomes (e.g., no attacker balance increase after price manipulation). We further refine this list based
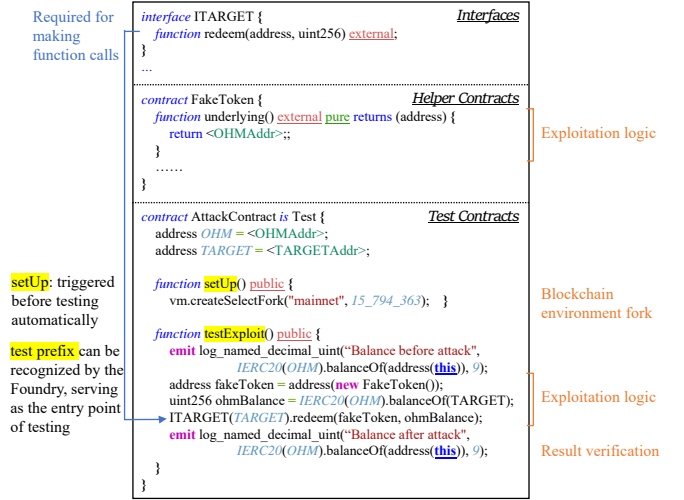
on the availability of vulnerable addresses and codes as well as the language of the contracts, prioritizing PoCs with comprehensive details and those written in Solidity due to our existing tooling preferences, resulting in 243 PoCs left. We then manually exclude PoCs that either lack complete exploit logic or are highly specific to their target construction. Two security experts independently complete this assessment within one day, as these patterns are easily distinguishable through code inspection. PoCs lacking exploit logic directly call attacker contracts without reconstructing the vulnerability exploitation logic (e.g., attackerContract.exploit()), making it impossible to extract exploit logic for migration analysis. Highly specific PoCs often contain hardcoded values such as long hash strings (e.g., signature replay with r, v, s parameters) that resist generalization across contracts. This results in a refined set of 201 PoCs for real-world attacks that happened between 2018 and July 2024, forming the basis for our subsequent analyses and experiments.

a) Vulnerability Type distribution: We categorize these attacks into 31 vulnerability types across 8 categories by extending existing taxonomies [30], [49]. The statistic summary is presented in Table I. Notably, access control, price manipulation, and logic flaw emerge as predominant categories. While existing tools perform poorly in price manipulation and logic flaw vulnerabilities, our PoC migration approach aims to help with the confirmation of these vulnerabilities, preventing recurring attacks across diverse smart contract environments.

b) Blockchain Network Distribution: The attacks span 7 EVM-compatible networks, with Binance Smart Chain (98 attacks) and Ethereum (77 attacks) accounting for the majority, while Polygon, Avalanche, Fantom, Optimism, and Base collectively represent 26 cases. This distribution reflects the market dominance of certain chains and highlights the need for cross-chain vulnerability detection approaches.

c) PoC Example: In Fig. 1, we present the PoC of Olympus DAO incident [5] as an example to illustrate key

components of PoCs (*for clarity of presentation, we have simplified the example by specific addresses and some functions*):

- *Interfaces*: Interfaces are required for enabling function calls to external contracts in Solidity due to its strong typing and need for explicit function signatures.
- *Helper Contracts*: Helper contracts are used to mimic legitimate behaviors or temporarily hold tokens, allowing attackers to manipulate the token flows. For instance, in the example, the *FakeToken* tricks the *TARGET* into transferring its *OHM* balance by mimicking an expired *bond* (insufficient validation of token metadata).
- *Test Contracts*: This component inherits from the Foundry framework [6], with built-in functions for easier simulations of EVM running environment. Only function with *test* prefix under Test Contracts will be served as the entry point of the simulation. Moreover, some PoCs include additional helper functions that are passively triggered but critical to the exploitation success.

IV. APPROACH

A. Overview

As illustrated in Fig. 2, we propose an automated framework for migration-based PoC generation. The framework operates in two primary modes: source code audit assistance, where it scans new contract code to identify vulnerabilities similar to past exploited cases and generates executable PoCs, and post-exploit scanning, where it verifies exploitability of contracts with similar vulnerabilities across blockchain networks following major attacks. Both scenarios utilize the same automated pipeline: in the first phase, we analyze existing PoCs to extract key components (i.e., environment properties, attacker contracts, validation checks), generating migration templates and function signatures to prepare our PoC database. Next, given a target address, we extract its source code and state variables (via Application Binary Interface) to search for suitable PoC templates from our database. Finally, the adapted PoC will be generated by passing the final similarity check. We then run the adapted PoCs and validate them in isolated environments with pre-defined validation checks.

B. PoC Abstraction

Smart contract PoCs for real-world incidents often rely on cross-contract interactions and specific blockchain state conditions. To enable automated migration, we extract three essential components: *1) Environment properties* to capture the complete state requirements for successful exploitation. While previous approaches focus on sender roles and gas limits, our analysis identifies broader requirements including state variable values and contract address relationships necessary for establishing exploitation preconditions. *2) Exploit logic* to capture both actively invoked and passively triggered functions critical to exploitation. Unlike previous approaches using simple EOA function calls, we capture both direct attacker calls and triggered functions (including fallbacks) that reflect complex real-world attack patterns. *3) Validation checks* to decide if an exploitation is successful, ensuring
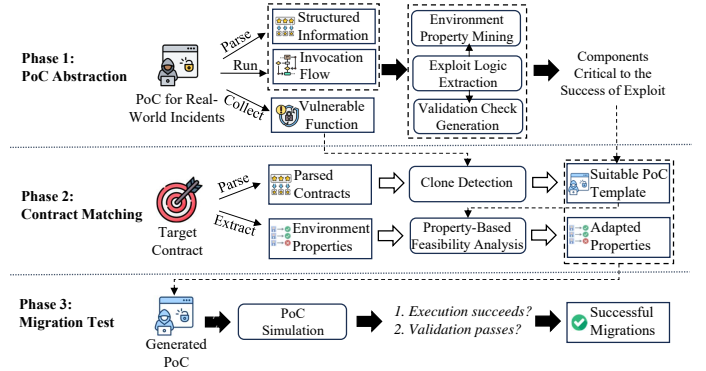

Fig. 2: Overview of our approach.

the practical effectiveness of migrated exploits. These components are derived from a comprehensive analysis of existing PoCs, cross-referencing with recent studies related to PoC generation [44], [38], [29]. This prioritizes soundness over completeness, ensuring every generated PoC corresponds to an exploitable vulnerability while potentially missing novel attack patterns outside our knowledge base.

Extracting key components from existing PoCs presents challenges due to the absence of standardized development practices and the implicit nature of exploitation requirements. Exploit logic can be structured in various ways: directly embedded in test functions, wrapped in helper functions, or distributed across multiple contracts, making systematic identification difficult. PoCs contain hardcoded addresses, token amounts, and contract parameters with limited documentation on their derivation and roles. Environment properties are rarely documented explicitly, as original PoCs typically restore required states by forking blockchain at specific block numbers. Existing PoCs also lack formal validation checks, instead relying on console log statements for test validation. To address these challenges, we leverage invocation traces as standardized execution representations that capture function call hierarchies, event emissions, and contract interactions, providing uniform structure across various PoC formats. The extraction process analyzes these traces to identify exploit sequences, trace involved addresses and events to reconstruct environment properties including contract relationships and state variables, and extract validation criteria from state changes between execution start and end points.

1) Exploit Logic Extraction: The first step of PoC abstraction focuses on extracting exploit logic. Our extraction process combines PoC parsing and invocation trace analysis.

We first parse the PoC into a structured representation with ANTLR4 [14], based on the exiting Solidity grammar [39] with minor modifications to support compiler-version compatibility and precise parsing (available on our webpage [12]). This representation maps interfaces, contracts, and their functions, along with variable-address bindings and function call dependencies. Each component maintains its source location information to facilitate precise tracking and reconstruction. However, PoCs present highly diverse structures: exploit logic

may appear inline in tests, encapsulated in helper functions, or distributed across contracts. This variability obscures the actual order of the exploit logic sequence.

To resolve this, we recover the order from invocation traces using Algorithm 1. The algorithm takes two inputs: $AddrFunc$ (address–function pairs parsed from PoC source code with ANTLR4) and $InvoTrace$ (the execution trace by running PoC where invocation order preserved). We first locate the parsed address–function pairs within the trace to recover their relative execution order (L1-4). Then, we traverse the annotated trace and classify functions by caller–callee relationships (L6-14): calls directly initiated by the attacker are marked as active exploit steps E, while those triggered by non-attacker addresses are treated as helper functions H that enable but do not constitute the attack (e.g., setup or approval).

Furthermore, to enhance migration success probability, we optimize the extracted exploit logic by detecting repeated function call sequences with different contract addresses but identical logic. For instance, an attack sequence {swap($pool_A$), drain($pool_A$), swap($pool_B$), drain($pool_B$)} contains redundant exploitation of independent pools. Such repetitions are typically for profit maximization, we keep the first instance and remove duplicates (reduce to {swap($pool_A$), drain($pool_A$)}). By removing address-related repetitions that merely enhance profitability, we reduce migration complexity by reducing the number of attack steps and addresses involved without compromising the fundamental attack logic. This simplification is justified as our goal is to validate the exploitability of a vulnerability rather than maximize profit.

2) Environment Property Mining: After deriving the exploit logic, POCSHIFT proceeds to mine the necessary environment properties. Unlike existing approaches that focus primarily on address roles and gas limits, our method leverages the rich context provided by the PoC and its invocation trace to identify additional properties such as address relationships and required state values. Although these properties are mostly implicit, we demonstrate how they can be systematically mined.

a) Address Relationship: To address the challenge of implicit address relationships, we first conducted address relationship mining using existing PoCs. This initial step is done by manually identifying each hardcoded address within the PoCs. These addresses are then labeled and clustered based on how they could be retrieved, given only the address containing the vulnerable code. This systematic approach helps us evaluate their interconnectedness and summarize their roles within the exploit frameworks as follows:

- **Common Addresses**: Widely used tokens (e.g., *WETH*, *WBNB*) and protocol addresses (e.g., *Uniswap V2 Router*). In PoCs, these addresses often serve as liquidity sources or interaction points for common DeFi operations, facilitating the setup of exploit conditions.
- **Derived Addresses**: Addresses retrievable from the read functions of the exploited address, representing contract-specific relationships. They frequently represent key contracts or tokens specific to the vulnerable protocol, crucial in manipulating the contract states.

Algorithm 1: Exploit Logic Extraction

Input: Address-function pairs $AddrFunc$,
       Invocation trace $InvoTrace$
Output: Exploit logic E, Helper functions H

1 **foreach** *node* n *in* $InvoTrace$ **do**
2   n.layer ← CalculateNestingDepth(n)
3   n.caller ← GetCallerAddress(n, $AddrFunc$)
4 **end**
5 $E \leftarrow \emptyset$, $H \leftarrow \emptyset$
6 **foreach** *node* n *in* $InvoTrace$ **do**
7   **if** $n.layer = \max(\{m.layer \mid m \in InvoTrace\})$
     then
8   **if** $n.caller = AttackerAddress$ **then**
9        $E \leftarrow E \cup \{n\}$
10   **else if** $n.caller \in ExternalAddresses$ **then**
11        $H \leftarrow H \cup \{n\}$
12   **end**
13   **end**
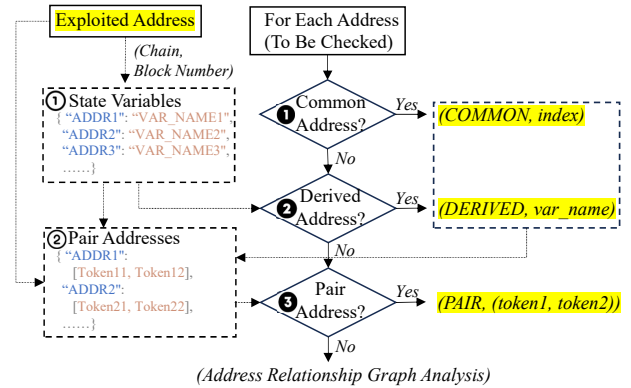14 **end**
15 **return** (E, H)

Fig. 3: Address relationship inference.

- **Pair Addresses**: Typically *Uniswap V2* pairs are formed across the first two types of addresses and the exploited address. In PoCs, these addresses are commonly used for price manipulation or flash loan attacks, playing a pivotal role in creating the necessary conditions for exploitation.

With the mined possible relationships, we are able to automate the address relationship inference process. As illustrated in Fig. 3, our inference process prioritizes the recovery of relationships with common addresses due to its efficiency and flexibility in real migration scenarios. We utilize a precompiled list of common protocol addresses, derived from our preliminary analysis. This comprehensive list was initially compiled from 1,178 unique chain-address pairs identified in our collected PoCs and further expanded using data from the Immunefi GitHub Repository [25]. In the end, we managed to compile a list of 45 commonly used addresses across 7 EVM-compatible chains. We encapsulated these addresses into a sub-module that can be automatically retrieved based
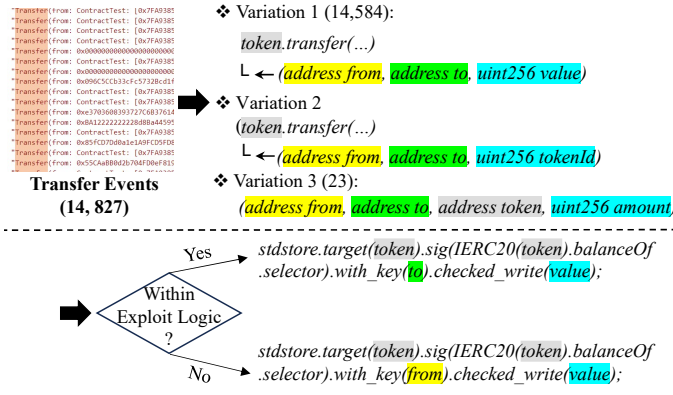
Fig. 4: Property mapping of *Transfer* event.

on the runtime chain-id. Within this submodule, addresses are carefully categorized and sorted according to their roles and usage frequencies for each network, ensuring efficient migration across various blockchain environments.

For addresses whose relationships to the exploited address remain unresolved after identifying common address, we proceed to examine the addresses against the state variables of the exploited address, accessible via its read functions. For any remaining unresolved addresses, we chcek potential pair relationships among all identified addresses to establish the necessary address mappings. If implicit address relationships still exist, we retrieve state variables from all recovered addresses, constructs a relationship graph based on these variables and known interactions. We then identify the shortest path from the exploited contract to unresolved addresses. This resolves cases where addresses connect to the exploited address indirectly through intermediate addresses. For example, if *Address A* lacks direct relation to the exploited address but *Address B* connects to both, we establish the path: Exploited Address $\rightarrow$ *Address B* $\rightarrow$ *Address A*, enabling address resolution.

b) State Variables: To mine implicitly required state variables and their values from the extracted exploit logic, we leverage the emitted events during exploitation. This strategy emerges from our observation that while PoC invocation flow traces typically contain over 20,000 function calls, they generate only a small, standardized set of event logs (208 distinct events across our PoC collection).

For instance, the *Transfer* event appears 14,827 times across our PoC corpus with only 3 variations as illustrated in Figure 4. Pattern 1 represents the standard ERC20 token event log, pattern 2 corresponds to ERC721 tokens, while pattern 3 is from a BSC token, occurring only 23 times in our dataset. Despite variations, these *Transfer* events can be translated into the same set of environment properties:

$$balance(from) \geq amount$$

and

$$approval(from, to) \geq amount$$

These two properties ensure that the *from* address has sufficient tokens and has granted enough approval for the *to* address to

successfully execute the transaction. However, they are not explicitly retrievable from the PoC code.

To leverage this insight, we first summarized an event-property mapping table, enabling the translation of standardized events into required properties. This was achieved by initially gathering event logs from the invocation traces of PoCs, from which we compiled a list of 208 events along with their frequencies. Subsequently, two smart contract security experts, each with at least two years of experience, analyzed each event and independently drafted the required properties. Any disagreements were resolved through discussions until a consensus was reached. This collaborative and systematic approach ensured a comprehensive and accurate mapping of events to their corresponding properties.

Moreover, to address the potential variability in event naming conventions across diverse smart contracts, we implemented a keyword-matching algorithm incorporating name stemming and lemmatization techniques. This enables us to group semantically similar events like *burn* and *burned* under the same condition settings, enhancing the robustness of our precondition reconstruction process across a wide range of smart contract implementations. We also discovered that the ordering of event mappings is crucial for accurately interpreting complex events. For example, an event named *transferDeposit* was identified during our examination, which contains two keywords (i.e., transfer and deposit). To deal with the ambiguities in multi-keyword events, we employed a ranking system for the matching process. With the ranking system, the event will be assigned to the higher-ranked transfer condition set rather than deposit based on the empirical analysis of their contextual relevance in exploit scenarios. The full list of event-condition mappings can be found in our webpage [12].

Apart from these implicit state properties, there are also explicit properties. Explicit properties are directly observable in PoC source code including token allowances, initial balance requirements, and direct state modifications. These properties are set through standardized testing patterns. For instance, *vm.deal()* is used for setting balances while *vm.prank()* and *approve()* calls are used together for setting token allowances. Since these leverage Foundry cheatcode [11], they can be automatically identified through pattern-based matching.

3) Validation Check Generation: Validation checks play important roles in minimizing false positives, which is a significant concern in smart contract vulnerabilities where successful execution of exploit logic does not necessarily equate to successful exploitation. For instance, given an exploit related to price manipulation, where the goal is to alter market conditions to the attacker's benefit, precise validation is required to ensure that the intended economic impact has occurred. Since over 95% PoCs log console values relevant to test validation after the exploit logic, we can effectively generate the required validation checks. For the remaining 5% of PoCs without console logs, validation is currently added manually by analyzing state changes and transaction outcomes. We have classified these validation checks into three distinct types as follows.

- **Profitability Oracle:** This oracle verifies whether the attacker successfully gains profit by the end of the exploit execution, comparing final balances against initial states; over 80% of generated oracles are this type. Specifically, $assert(token.balanceOf(attacker) > initBalance)$.
- **State Change Oracle:** This oracle verifies exploitation by detecting critical state changes before and after the exploit. For example, to validate ownership transfer, it uses $assert(contract.owner()! = originalOwner)$.
- **DoS Oracle:** To confirm the impact of denial-of-service vulnerabilities, this oracle verifies if a function call fails to execute post-exploit. Validation detects failures by catching function revert, out-of-gas errors, and timeouts.

C. Candidate Matching

To select appropriate PoC templates for a given target contract, our approach first uses clone-based detection to identify functions similar to known vulnerable functions exploited in real-world incidents. For identified functions, we then perform property-based feasibility analysis to determine whether an exploit path exists in the target contract. This two-stage filtering approach enables rapid candidate matching across large-scale contract repositories, with expensive property-based analysis performed only on hash-filtered candidates to reduce computational overhead, improving both efficiency and precision.

1) Clone Detection: We adopt Type-2 hash-based matching to achieve the scalability. This approach captures more vulnerable patterns than exact matching while maintaining efficiency compared to costly Type-3/4 approaches [37], [36]. Our implementation uses ANTLR4 parser to extract and normalize key contract components and the TLSH algorithm [10] to calculate hash values. Hash collisions are rare ($\sim 2.3 \times 10^{-7}$) and can be filtered out during subsequent feasibility analysis. While we build upon established clone detection techniques, our contribution lies in how we integrate this similarity detection with contextual feasibility analysis (Section IV-C2) to enable effective PoC migration.

2) Property-Based Feasibility Analysis: For feasibility analysis, our framework extracts state variables and functions from the target contract via its ABI which provides a standardized interface specification to identify public state variables (exposed as static calls) and functions, including their input and output parameter types.

Our analysis proceeds in two steps. First, we match functions and variables between the source PoC and target contract based on two criteria: input/output type compatibility (allowing for equivalent types like uint and uint256) and name similarity measured by cosine similarity. Only candidates satisfying both criteria are considered valid matches, ensuring semantic equivalence while maintaining efficiency across large contract codebases. Second, we validate the execution path between the identified vulnerable function and its entry point function (the public function that precedes the vulnerable call in the original PoC). By verifying the existence of a valid call path in the statical function graph of the target contract, we ensure the vulnerability can be triggered through an accessible sequence of function calls. Only templates passing this feasibility analysis proceed to migration testing.

This analysis is to filter out clearly incompatible cases. It uses intentionally relaxed criteria to avoid excluding potential vulnerabilities, balancing efficiency with effectiveness.

D. Migration Test

Given the selected template and adapted values, the final phase of our approach is to test the PoC. All PoCs are generated and tested in an isolated blockchain environment to simulate on-chain conditions without real-world impact.

The testing process begins with environment setup based on the adapted properties from the PoC abstraction phase (Section IV-B), reconstructing the required state conditions (e.g., state variable initialization and permission configuration). We then execute the adapted exploit sequence by mapping template addresses to target addresses and adjusting function parameters based on the state of the target contract. To validate exploitation success, we employ the generated validation checks from the original PoC, which verify whether the exploit achieved its intended impact. This ensures that only successfully migrated and validated PoCs are reported as confirmed vulnerabilities in the target contract.

V. EVALUATION

A. Research Questions

To evaluate the performance of our tool in PoC generation, we answer the following questions.

RQ1: How effective and efficient is POCSHIFT in generating functional PoCs for contracts similar vulnerable functions? What is the performance of other selected tools?

RQ2: How does each component contributes to POCSHIFT?

RQ3: Can POCSHIFT adapt existing PoCs to generate PoCs for similar vulnerabilities in real-world scenarios?

RQ4: To what extent does POCSHIFT demonstrate practical effectiveness on large-scale smart contracts?

B. Tool Selection

To the best of our knowledge, our work is the first attempt at migration-based PoC generation for smart contracts. To better evaluate the effectiveness of our approach, we select state-of-the-art tools for each PoC generation technique as follows:

- **ItyFuzz** [38]: A snapshot-based fuzzer with waypoint mechanisms to prioritize the exploration of interesting snapshot states and reduce re-execution overhead.
- **FORAY** [42]: An attack synthesis framework for deep logical vulnerabilities with a domain-specific language to analyze token flow graphs and generate attack sketches.
- **Mythril** [4]: A security analysis tool that uses symbolic execution to exploit vulnerabilities in bytecode.

While these tools represent various approaches to PoC generation, we exclude FORAY from our comparison due to its substantial manual requirements for writing attack goal specifications and function mappings [42]. These requirements make it impractical for automated analysis at scale.

C. Evaluation Dataset

To better assess POCSHIFT, we evaluate it on a dataset of 5,713 unique verified smart contracts deployed across seven EVM-compatible blockchains as mentioned in Section III. Unlike existing datasets that primarily contain simplified examples failing to reflect real-world complexity, this dataset represents diverse real-world deployments with retrievable on-chain state information. It is important to note that this dataset excludes contracts in our PoC collection, preventing data leakage and ensuring unbiased evaluation.

Regarding ground truth construction, since no comprehensive vulnerability annotations exist for real-world contracts, we follow the approach applied in existing works [45], [28], [21] for fair ground truth creation. We execute POCSHIFT and selected SOTA on this sampled dataset and manually validate their outputs to establish ground truth. The validation process is conducted by two smart contract security experts, each with more than two years of experience. They independently verify each reported vulnerability, and any disagreements are resolved through discussion until consensus is reached. By deriving ground truth from expert validation across all tool results rather than being biased toward the output of any single tool, we try to mitigate potential overfitting. Complete dataset details are available on our website [12].

D. PoC Generation Capability

1) Experiment Details: To answer RQ1, we compare our tool with selected SOTA on the evaluation dataset, measuring both effectiveness and efficiency. For effectiveness, we evaluate true positives (TP, runnable PoC to successfully exploit vulnerabilities), false positives (FP, runnable PoC but incorrect exploit logic). True negatives (TN) and false negatives (FN) would require comprehensive manual analysis of all 5,713 contracts to establish complete ground truth, which exceeds practical research scope. Our precision-focused evaluation corresponds to real-world deployment scenarios where analysts need trustworthy vulnerability identification. For efficiency, we compare the execution time required by each tool when processing the evaluation dataset. All generated PoCs are manually validated by two security experts with over two years of experience in smart contract security, with disagreements resolved through discussion until consensus is reached.

2) Result Analysis: We present the analysis for our tools and the selected tools from effectiveness and efficiency.

Effectiveness Analysis. Table II demonstrates the effectiveness of POCSHIFT compared to existing tools. Based on the ground truth established (Section V-C), our framework successfully generates valid PoCs for 62 out of 67 exploitable instances with zero false positives, while ItyFuzz and Mythril identified only 8 and 0 instances respectively. The 67 exploitable instances represent ground truth established through expert validation across outputs of all evaluated tools, following established methodology in security research when comprehensive annotations are unavailable [45], [28], [21]. While this represents a small fraction, it reflects the realistic

TABLE II: Summary of PoC generation results.

Category (#GT)	ItyFuzz	Mythril	POCSHIFT
AC (11)	✓ (TP: 2, FP: 8)	✓ (TP: 0, FP: 0)	✓ (TP: 11, FP: 0)
LF (18)	✓ (TP: 2, FP: 0)	✗	✓ (TP: 17, FP: 0)
PM (23)	✓ (TP: 4, FP: 3)	✗	✓ (TP: 19, FP: 0)
RE (1)	✓ (TP: 0, FP: 0)	✓ (TP: 0, FP: 0)	✓ (TP: 1, FP: 0)
AR (7)	✗	✓ (TP: 0, FP: 0)	✓ (TP: 7, FP: 0)
BM (7)	✓ (TP: 0, FP: 0)	✓ (TP: 0, FP: 0)	✓ (TP: 7, FP: 0)
Total (67)	**TP: 8, FP: 11**	**TP: 0, FP: 0**	**TP: 62, FP: 0**

* #GT: number of ground truth, TP: True positive, FP: False positive.

distribution of exploitable vulnerabilities in production environments rather than methodological limitations.

This performance gap comes from our method of modeling existing attack sequences from real-world exploits rather than relying on generic vulnerability patterns. Existing tools generally use generic vulnerability patterns and simplified environmental assumptions, which do not adequately capture the complex conditions of real-world exploits. While ItyFuzz can technically cover new vulnerability types, the 8 cases identified by ItyFuzz generate similar simple traces with fewer than 4 function calls each, significantly simpler than real-world exploits requiring multi-step exploit logic, complex state manipulation, and cross-contract interactions. Instead, our tool is designed to target successful exploitation by learning from actual attack patterns and maintaining essential environmental dependencies that enable real attacks. The observed false negatives (5 cases) are mainly due to environmental constraints inherent to real-world smart contract ecosystems: ① dependencies on deprecated or replaced external contracts that alter the exploitation context, and ② variations in storage layouts that affect state manipulation sequences. To address these cases would require in-depth semantic analysis and complex state dependency chains of the target contract. While such analysis is technically feasible, it would significantly increase computational overhead and analysis time, potentially limiting the practical applicability of our current approach that prioritizes efficiency and scalability. We leave this semantic-aware exploitation analysis as future work to maintain a balanced trade-off between coverage and real-world usability.

Our manual validation reveals that 11 of 19 cases reported by ItyFuzz are false positives. False positives occur when ItyFuzz generates a PoC for a reported vulnerability that cannot actually be exploited in real-world blockchain conditions. These false positives are mainly caused by its approximation of real-world conditions and oversight of access control modifiers. For instance, several ItyFuzz exploits failed during execution because of price slippage effects absent in its simplified market model. In contrast, our migration-based approach achieves zero false positives by preserving actual exploitation conditions from documented similar incidents rather than approximating them.

These results also underscore that our framework prioritizes precision over recall in security analysis. This ensures that reported cases are exploitable threats rather than theoretical weaknesses, reducing the burden of manual validation while increasing the reliability of automated security assessments.

TABLE III: Summary of time spent over the evaluation dataset.

Tool	Input	Time Spent (in hrs)
Mythril	Source Code	210.5 ▉
ItyFuzz	Chain & RPC	133.2 ▉
PoCSHIFT	Chain & RPC	3.8 ▏

Efficiency Analysis. As presented in Table III, POCSHIFT achieves a notable improvement in PoC generation time, completing the evaluation dataset in just 3.8 hours, compared to 133.2 and 210.5 hours required by ItyFuzz [38] and Mythril [4] respectively. Note that the time spent for Mythril includes additional compilation overhead due to source code input since its RPC support is limited to a restricted format.

This significant efficiency gain arises from our fundamentally different approach. Traditional approaches like ItyFuzz and Mythril rely on resource-intensive techniques such as exhaustive exploration and heavy symbolic reasoning. In contrast, our tool focuses on precisely generating PoCs for contracts with vulnerable functions similar to those exploited in real-world incidents. This specialized focus represents a strategic trade-off: while we may not cover all possible vulnerabilities, our approach enables rapid vulnerability assessment at a scale previously unattainable. Such capability is particularly valuable for security researchers and developers who need to quickly validate the security implications of code reuse, which is a common practice in smart contract development. In addition, the capability of our tool can be further strengthen by continuously compiling new vulnerability PoCs.

These results indicate that our precision-focused design reduces the effort of verifying false positives, while a 35 times efficiency improvement over existing tools suggests feasibility for integration into development cycles.

E. Ablation Study

1) Experiment Details: To answer RQ2, we perform an ablation study to evaluate the contributions of three critical components: environment property mining, validation check generation, and candidate matching. We exclude each component individually and evaluate their impact based on design objectives: environment property mining to increase migration success by abstracting essential components to successful migration, validation check generation to reduce false positives by ensuring exploitation verification, and candidate matching to improve efficiency by optimizing target selection procedure. In summary, the first two components target effectiveness while candidate matching targets efficiency.

2) Result Analysis: Table IV summarizes how removing environment property mining and validation check generation impacts key metrics such as True Positives (TP), False Positives (FP), Recall, and Precision, whereas Table V presents the effect of excluding candidate matching on execution time. Below, we provide a detailed discussion of each ablation.

Environment Property Mining. Environment property mining is a novel component introduced to ensure that PoCs are migrated with a precise understanding of cross-contract dependencies, relevant blockchain states, and external library

calls. The importance of environment property mining is evident from the results shown in Table IV. Excluding environment property mining causes the true positives to plummet from 62 to 7, resulting in a 88.70% drop in Recall. This steep decline underscores the complexity of migrating existing PoCs to new targets. In practice, even if the transaction sequence itself is valid, failing to replicate external properties prevents the exploit from succeeding. Thus, these results demonstrate that environment property mining is essential for reliable PoC migration in smart contract settings, and the properties we mined are critical to successful migrations.

Validation Check Generation. Validation check generation is designed to validate that the discovered exploit paths lead to meaningful exploitation outcomes (e.g., an unauthorized funds transfer or profit gain). Without validation check generation, our tool still can detect 62 true positives, indicating that it retains its capability to generate PoCs for new targets. However, the lack of robust oracles causes 81 false positives, dropping the precision rate to 43.36%.

The 81 false positives represent cases when PoCs are generated and executed successfully but fail to achieve actual exploitation goal. One representative example is that, in price manipulation exploits, merely executing a transaction sequence that swaps tokens successfully does not guarantee a net profit. With validation check generation, our tool can generate required checks to distinguish between a trivial sequence of interactions and a truly profitable exploit (e.g., verifying actual balances after swapping in the price manipulation example).

This finding highlights that many *exploitable* paths are not profitable or do not cause adverse effects once detailed financial or state-related checks are performed. Additionally, if vulnerability identification produces false positives in earlier stages, our precision-focused design filters them out during adaptation or migration testing, as they cannot generate valid exploitation outcomes under forked blockchain environments. Consequently, our ablation study confirms that validation check generation effectively reduces false positives by requiring tangible exploit outcomes, improving the overall precision of our PoC generation process.

Candidate Matching. While environment property mining and validation check generation focus on enhancing effectiveness, candidate matching serves as a filtering mechanism to improve efficiency. By checking the compatibility of conditions such as whether the environment properties of target contracts align with the original PoC, candidate matching helps to filter out unpromising test cases early on. Table V illustrates that candidate matching significantly reduces the time overhead of our approach, by cutting the number of test runs from 5,713 to 1,306, thereby shortening the total test execution time from 15.9 hours to 3.8 hours. The substantial 4.2 times reduction in test runs demonstrates the practical value of candidate matching for real-world deployment. To validate that this efficiency gain does not compromise effectiveness, we conduct a manual validation on 384 filtered-out cases (calculated using Cochran's formula with a 95% confidence level). Our investigation finds no missed exploits, confirming that our

TABLE IV: Contribution of environment property mining (EPM) and validation check generation (VCG).

	TP	FP	Recall	Precision
POCSHIFT	62	0	92.50%	100.00%
POCSHIFT *w/o* **EPM**	7	0	**10.45%**	100.00%
POCSHIFT *w/o* **VCG**	62	81	92.50%	**43.36%**

*TP: True positive, FP: False positive.

TABLE V: Time spent *with* and *without* candidate matching.

Configuration	#Test	Matching	Test	Total
POCSHIFT *w/* **CM**	1,306	1.3	2.5	**3.8**
POCSHIFT *w/o* **CM**	5,713	-	15.9	**15.9**

* #Test: Number of tests left after the matching process (if applicable), CM: Candidate matching. The rest columns present time spent in hours.

matching criteria remain sufficiently relaxed to preserve all viable exploit candidates while excluding only conclusively non-viable pairs based on environment properties.

F. Real-World Practicability

1) Experiment Details: To answer RQ3, we provide a case study with the collected PoC to exploit Onyx Protocol in 2023, whose exploitation logic is explained in our motivating examples (Section II-C). This example is to illustrate the complexity of real-world exploits, providing evidence on the limitations of generating PoCs from scratch, while demonstrating the effectiveness of our tool in extracting essential exploitation components for reuse across different vulnerable contracts.

2) Result Analysis: The original PoC for the Onyx Protocol incident involves over 1,200 lines of Solidity code across three contracts, coordinates with 15 external contracts using 25 distinct addresses, and features five-layer deep cross-contract calls with dynamic helper contract deployments. Such complexity makes it impractical for existing tools that struggle with precise state coordination and cross-contract interactions.

This complexity demonstrates why existing automated tools fail to generate such PoCs from scratch, while our approach can automatically abstract essential components for PoC migration. First, it parses the Solidity code to identify interfaces and map hardcoded addresses to their corresponding variables. By executing the PoC in a controlled environment, it generates an invocation trace of 5,359 lines that reveals the critical exploitation sequence. POCSHIFT also identifies the existence of repeated exploit steps across six different stablecoins to maximize profit. To simplify the template, our framework extracts the core pattern targeting just one stablecoin (WETH), reducing the required addresses from 25 to only 5. Subsequently, through event analysis, POCSHIFT identifies eight state requirements necessary for successful exploitation, including cross-contract states such as allowance approvals. As a result, POCSHIFT reduces the PoC to 94 lines of code while preserving the functionality of the PoC.

G. Large-scale Performance

1) Experiment Details: To answer RQ4, we evaluate POC-SHIFT on 979,512 verified contracts from 2021-2024 across 7 EVM-compatible blockchain networks (detailed statistics in our webpage [12]). We assess practical effectiveness by using collected PoCs to identify potential attack targets with similar vulnerable functions in large-scale verified contracts by migration-based PoC generation. For each PoC, we perform abstraction, candidate matching, and migration testing. Generated PoCs are manually verified, with 382 failed cases analyzed via stratified sampling (95% confidence, 5% margin of error); however, they are not necessarily false negatives.

2) Result Analysis: From the 979,512 contract dataset, we generated and validated PoCs for 256 contracts across 12 vulnerability types in 6 categories. The detailed breakdown with all execution trace could be found in our website due to space constraint [12]. Among these, 13 contracts are flagged as exploited or phishing on blockchain platforms, providing external validation of our identification accuracy. Additionally, the results include 64 cross-chain migrations, with only 23 contracts sharing protocols with the original vulnerable contracts. These results demonstrate the effectiveness of migrating PoCs across different blockchain networks and protocols.

For unsuccessful migration cases, we manually confirm that none of the 382 sampled contracts were exploitable, indicating no false negatives within this sample. This confirmation is conducted by contract security experts with consensus on disagreements. However, this validation faces inherent limitations due to the lack of large-scale, high-quality labeled datasets of real-world smart contract exploitability. While we can reliably validate positive cases, comprehensive false negative assessment remains challenging given the imbalanced distribution of vulnerable versus non-vulnerable contracts real-world deployments [35], where manual analysis of sufficient contracts for statistical confidence is impractical.

VI. DISCUSSION

A. Limitations and Future Work

While our automated PoC migration approach shows potential for enhancing smart contract security, we acknowledge its limitations and outline future work. First, our current implementation leverages well-established clone detection technique for candidate matching (Section IV-C). While this technique proves effective for the migration task, future research could explore more sophisticated semantic matching techniques to potentially increase recall without compromising the the overall efficiency and precision of the framework.

Second, our framework builds on knowledge extracted from existing PoCs. While this is supported by growing availability of organized repositories (e.g., DeFiHackLabs [18]), we recognize the gap between when incidents occur and when PoCs become available. Future work can explore automated transaction-based incident analysis to extract key exploitation components directly from blockchain data, reducing PoC dependency. However, this direction faces challenges in reconstructing contract semantics from transaction traces and identifying exploit operations, which remain open problems.

Third, our validation checks create a trade-off between precision and developer warnings. While potentially vulner-

able code that currently fails exploitation might eventually be exploited through different vectors, our approach prioritizes immediate demonstrable threats in blockchain environments. Future work could analyze validation-failed cases to categorize them by exploitability likelihood, helping developers prioritize manual review efforts on cases with higher potential for becoming exploitable under different conditions.

B. Threats to Validity

First, the manual exclusion of PoCs during PoC collection may introduce bias. To mitigate this, we implement predefined criteria with clear patterns (e.g., direct call functions from attacker contracts on chain) and invite two researchers to independently complete the task following predefined criteria with consensus on disagreements.

Second, our approach relies on the representativeness of our evaluation dataset. To mitigate selection bias, we collect contracts from diverse blockchain networks and conduct similarity matching to ensure different contract-level implementations. All confirmed vulnerable cases have distinct structures, implementation patterns, and deployment contexts, ensuring results are not inflated by analyzing variants of the same contract.

Third, regarding ground truth construction, to reduce bias and avoid overfitting toward POCSHIFT, ground truth is derived from expert validation across all tool outputs rather than any single tool. This ensures fairness in the constructed dataset, following established practices [45], [28], [21] when comprehensive vulnerability annotations are unavailable.

VII. RELATED WORK

A. Smart Contract PoC Generation

Exploit migration for smart contracts builds on exploit generation by adapting attacks to new contexts, though research here remains limited. Traditional methods like those proposed by Ignacio et al.[15] and Feng et al.[22] generate attacker contracts from victim contract ABIs and predefined templates, limiting their ability to detect vulnerabilities with complex logic and inter-contract interactions. In contrast, fuzzing methods such as ConFuzzius [41], ContractFuzzer [26], and Ity-Fuzz [38] focus on identifying vulnerabilities at the bytecode level or through ABI-driven inputs but typically lack cross-contract analysis capabilities. For example, ItyFuzz introduces snapshot-based fuzzing to minimize re-execution overhead, but struggles with complex vulnerabilities like price manipulation and lacks efficiency in cross-contract contexts.

However, existing tools often struggle to generate PoCs for real-world vulnerabilities. These vulnerabilities mostly involve complex transaction dependencies and state conditions that are challenging to automate from scratch, even with substantial computational resources. Our approach overcomes these limitations by systematically abstracting key components from existing PoCs and adapting to other contract environments.

B. Migration-based Test Generation

Research on migration-based test case generation for vulnerabilities in open-source software (OSS) libraries has pro-gressed in several directions. SIEGE [24] automates minimal test case creation for vulnerable library fragments, while TRANSFER [27] uses evolutionary search to reproduce vulnerability-triggering program states in client code. VESTA [17] integrates static analysis to migrate parameters into existing project tests, reducing manual effort through combined testing approaches. While conventional migration adapts source code across APIs [34], libraries [23], or languages [47] to preserve semantics, these approaches migrate testing artifacts to reproduce specific behaviors like vulnerability triggering conditions in new contexts.

However, smart contract PoC migration faces unique challenges. Smart contracts execute in persistent blockchain environments where exploitability depends on contract states, external addresses, and cross-contract interactions. Valid PoC construction requires orchestrating transaction sequences rather than adapting function parameters, limiting the applicability of existing migration techniques and necessitating specialized blockchain approaches.

VIII. CONCLUSION

In this paper, we introduce POCSHIFT, an automated PoC generation tool for smart contracts. Unlike existing methods that rely on exhaustive exploration or symbolic reasoning, POCSHIFT uses existing PoCs to generate PoCs for contracts with similar vulnerabilities. The evaluation demonstrates the effectiveness of POCSHIFT by successfully generating PoCs for 62 out of 67 cases, outperforming existing tools including ItyFuzz and Mythril. For efficiency, POCSHIFT completes analyzing 5,713 on-chain contracts in just 3.8 hours versus 133.2 and 210.5 hours by existing tools. Large-scale evaluation on 979,512 contracts identifies 256 vulnerable contracts across multiple blockchain networks, including 64 cross-chain cases. These results demonstrate the effectiveness of our approach in generating PoCs for recurring real-world vulnerabilities, contributing to smart contract security analysis.

IX. DATA AVAILABILITY

All experimental data, evaluation results, and tool implementations are publicly available on our webpage [12] to support reproducibility and future research.

REFERENCES

[1] "Auditing with poc tests in audit wizard," 2024, [Online; accessed 2024-12-28]. [Online]. Available: https://www.auditwizard.io/blog/auditing-with-poc-tests-in-audit-wizard

[2] "Code4rena — keeping high severity bugs out of production," 2024, [Online; accessed 2024-12-23]. [Online]. Available: https://code4rena.com/

[3] "Compound v2 documentation," 2024. [Online]. Available: https://docs.compound.finance/v2/

[4] "Consensys/mythril: Security analysis tool for evm bytecode. supports smart contracts built for ethereum, hedera, quorum, vechain, rootstock, tron and other evm-compatible blockchains." https://github.com/ConsenSys/mythril, 2024, (Accessed on 07/08/2024).

[5] "Defihacklabs/src/test/2022-10/olympusdao_exp.sol at 6811c608ff90d9bde1c3067da12b958dd63e890b · sunweb3sec/defihacklabs," https://github.com/SunWeb3Sec/DeFiHackLabs/blob/6811c608ff90d9bde1c3067da12b958dd63e890b/src/test/2022-10/OlympusDao_exp.sol, 2024, (Accessed on 09/11/2024).

[6] "foundry-rs/foundry: Foundry is a blazing fast, portable and modular toolkit for ethereum application development written in rust." https://github.com/foundry-rs/foundry, 2024, (Accessed on 03/17/2024).

[7] "Immunefi," 2024, [Online; accessed 2024-12-23]. [Online]. Available: https://immunefi.com/

[8] "Immunefi bug bounties — immunefi," 2024, [Online; accessed 2024-12-28]. [Online]. Available: https://immunefi.com/bug-bounty/immunefi/information/

[9] "Onyx address etherscan," https://etherscan.io/address/0x5FdBcD61bC9bd4B6D3FD1F49a5D253165Ea11750, 2024, (Accessed on 07/08/2024).

[10] "Tlsh - a locality sensitive hash," https://tlsh.org/, 2024, (Accessed on 03/19/2024).

[11] "foundry - ethereum development framework," 8 2025, [Online; accessed 2025-08-29]. [Online]. Available: https://getfoundry.sh/forge/tests/cheatcodes/

[12] "kairanskrr/pocshift," 2025, [Online; accessed 2025-10-02]. [Online]. Available: https://github.com/kairanskrr/PoCShift

[13] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE access*, vol. 7, pp. 86 121–86 144, 2019.

[14] ANTLR, "antlr/antlr4: Antlr (another tool for language recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files." https://github.com/antlr/antlr4, 2024, (Accessed on 03/17/2024).

[15] I. Ballesteros, C. Benac-Earle, L. E. B. de Barrio, L.-Å. Fredlund, Á. Herranz, and J. Mariño, "Automatic generation of attacker contracts in solidity," in *4th International Workshop on Formal Methods for Blockchains (FMBC 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.

[16] BscScan.com, "Cerc20delegator — address 0xf8527dc5611b589cbb365acacaac0d1dc70b25cb — bscscan," 2024. [Online]. Available: https://bscscan.com/address/0xf8527dc5611b589cbb365acacaac0d1dc70b25cb#code

[17] Z. Chen, X. Hu, X. Xia, Y. Gao, T. Xu, D. Lo, and X. Yang, "Exploiting library vulnerability via migration based automating test generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.

[18] DeFiHackLabs, "Defihacklabs: Reproduce defi hacked incidents using foundry." https://github.com/SunWeb3Sec/DeFiHackLabs/tree/main, 2023, (Accessed on 12/25/2023).

[19] T. Edge, "Crypto market cap races toward $4 trillion as congress passes three key bills - coincentral," 7 2025, [Online; accessed 2025-08-25]. [Online]. Available: https://coincentral.com/crypto-market-cap-races-toward-4-trillion-as-congress-passes-three-key-bills/?utm_source=chatgpt.com

[20] etherscan.io, "Ethereum charts and statistics — etherscan," 2025, [Online; accessed 2025-08-22]. [Online]. Available: https://etherscan.io/charts

[21] S. Feng, Y. Wu, W. Xue, S. Pan, D. Zou, Y. Liu, and H. Jin, "Fire: combining multi-stage filtering with taint analysis for scalable recurring vulnerability detection," in *Proceedings of the 33rd USENIX Conference on Security Symposium*, ser. SEC '24. USA: USENIX Association, 2024.

[22] Y. Feng, E. Torlak, and R. Bodík, "Summary-based symbolic evaluation for smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1141–1152.

[23] H. He, R. He, H. Gu, and M. Zhou, "A large-scale empirical study on java library migrations: prevalence, trends, and rationales," in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 478–490.

[24] E. Iannone, D. Di Nucci, A. Sabetta, and A. De Lucia, "Toward automated exploit generation for known vulnerabilities in open-source libraries," in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE, 2021, pp. 396–400.

[25] immunefi, "immunefi-team/hack-analysis-pocs," https://github.com/immunefi-team/hack-analysis-pocs/tree/main, 2023, (Accessed on 12/25/2023).

[26] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 259–269.

[27] H. J. Kang, T. G. Nguyen, B. Le, C. S. Păsăreanu, and D. Lo, "Test mimicry to assess the exploitability of library vulnerabilities," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 276–288.

[28] W. Kang, B. Son, and K. Heo, "Tracer: Signature-based static analysis for detecting recurring vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1695–1708. [Online]. Available: https://doi.org/10.1145/3548606.3560664

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

[30] K. Li, Y. Xue, S. Chen, H. Liu, K. Sun, M. Hu, H. Wang, Y. Liu, and Y. Chen, "Static application security testing (sast) tools for smart contracts: How far are we?" *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660772

[31] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[32] B. News, "Web3 security threats surge in 2024 with significant financial losses — binance news on binance square," 12 2024, [Online; accessed 2024-12-23]. [Online]. Available: https://www.binance.com/en/square/post/12-22-2024-web3-security-threats-surge-in-2024-with-significant-financial-losses-17887756196249

[33] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[34] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring api embedding for api usages and applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 438–449.

[35] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1325–1341.

[36] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[37] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[38] C. Shou, S. Tan, and K. Sen, "Ityfuzz: Snapshot-based fuzzer for smart contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 322–333.

[39] solidity parser, "Github - solidity-parser/antlr: Solidity grammar for antlr4," 2025, [Online; accessed 2025-08-29]. [Online]. Available: https://github.com/solidity-parser/antlr

[40] K. Sun, Z. Xu, C. Liu, K. Li, and Y. Liu, "Demystifying the composition and code reuse in solidity smart contracts," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 796–807.

[41] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021, pp. 103–119.

[42] H. Wen, H. Liu, J. Song, Y. Chen, W. Guo, and Y. Feng, "Foray: Towards effective attack synthesis against deep logical vulnerabilities in defi protocols," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 1001–1015.

[43] S. Wu, Z. Li, L. Yan, W. Chen, M. Jiang, C. Wang, X. Luo, and H. Zhou, "Are we there yet? unraveling the state-of-the-art smart contract fuzzers," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[44] ——, "Are we there yet? unraveling the state-of-the-art smart contract fuzzers," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639152

[45] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "{MVP}: Detecting vulnerabilities using {Patch-Enhanced} vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.

[46] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 615–627.

[47] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 195–204.

[48] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 919–936.

[49] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) attacks," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2444–2461.