VeriExploit: Automatic Bug Reproduction in Smart Contracts via LLMs and Formal Methods

Chenfeng Wei
The University of Manchester
Manchester, United Kingdom
chenfeng.wei@manchester.ac.uk

Shiyu Cai
The University of Manchester
Manchester, United Kingdom
shiyu.cai@manchester.ac.uk

Yiannis Charalambous
The University of Manchester
Manchester, United Kingdom
yiannis.charalambous-4@postgrad.manchester.ac.uk

Tong Wu
The University of Manchester
Manchester, United Kingdom
tong.wu-11@postgrad.manchester.ac.uk

Sangharatna Godboley
National Institute of Technology Warangal
Warangal, India
sanghu@nitw.ac.in

Lucas Cordeiro
The University of Manchester
Manchester, United Kingdom
lucas.cordeiro@manchester.ac.uk

***Abstract*—Bug reproduction is becoming an important task in the security analysis of Solidity smart contracts. By simulating attacks, developers and auditors can better understand how a vulnerability is triggered in practice. To reproduce a bug, one often needs to define an attacker contract and a specific sequence of interactions that exploit the vulnerability. However, in smart contracts, there are rarely automated tools that can generate such contracts and sequences and validate their correctness. Existing security tools, such as formal verifiers, are effective at detecting bugs, but they are not designed for bug reproduction. They often omit execution traces or produce incomplete ones. Moreover, their reports rarely reflect the behaviour patterns of attacker contracts. This gap motivates our work. We propose `VeriExploit`, a framework that combines formal methods and large language models to automatically generate, validate, and refine reproduction contracts and execution steps. Given a vulnerable contract and its counterexample, `VeriExploit` produces a contract that re-triggers the same bug and outputs a concrete trace showing how the exploit works. Experiments show that `VeriExploit` is effective at automating bug reproduction, achieving a success rate of 85.60% on our benchmark dataset.**

***Index Terms*—Smart Contract, Formal Verification, Large Language Model**

I. Introduction

Bug reproduction is fundamental to understanding and confirming software vulnerabilities: it demonstrates how a bug is triggered and whether a fix actually works. In smart contracts this need is amplified. Contracts do not run in isolation; they interact with other contracts and users through sequences of transactions that evolve state over time. Many vulnerabilities manifest only after multi-step, cross-contract interactions (e.g., reentrancy when an external call precedes a state update), which makes faithful reproduction harder than in traditional software.

Several recent studies have explored how to automate this process for smart contracts. Ballesteros et al. [1] propose a method that uses property-based testing to generate attacker contracts targeting reentrancy bugs. Their approach identifies a common structure for such attackers and uses it as a template, so-called semi-random. `Advscanner` [2] takes a different route, combining static analysis and large language models. It extracts attack flows for reentrancy from `Slither` reports and guides the LLM to complete attacker contract templates. It also uses feedback from compilation and runtime execution to refine the generated results. `Skyeye` [3], on the other hands, utilised such generated adversarial contracts, and showed that pairing adversarial and vulnerable contracts helps detect attacks in advance, using large language models to identify vulnerability types from generated contracts.

While promising, these approaches still have limitations. (1) They are commonly designed for a single bug type (e.g., reentrancy) and are not easily extended to other vulnerability classes. (2) They often depend on manual or simulation-based testing to determine whether the generated contract actually reproduces the bug; such testing can miss cases and provides no formal guarantee that the bug is triggered. (3) The prior work focuses mainly on attacker-contract generation while overlooking the importance of formally validating the execution sequences.

These gaps motivate our work: `VeriExploit`, a framework that combines formal verification with large language models to automatically generate and validate both reproduction contracts and the concrete execution steps needed to trigger the bug across diverse vulnerability types (e.g., logic errors, arithmetic bugs, reentrancy). Instead of templates, it extracts context from verifier reports to guide generation; it then checks exploitability and returns a step-by-step trace, and—if validation fails—diagnoses the cause and retries with targeted feedback.

This work advances the state of the art in smart-contract bug reproduction by tackling three open problems: limited bug coverage, lack of formal validation of exploitability, and incomplete counterexample traces.

In general, our contributions are:

Vulnerability Coverage. We present `VeriExploit`, a novel framework that targets reproducing multiple vulnerability types reported by verifiers. Our framework covers six vulnerability types, including arithmetic errors, assertion violations, and reentrancy, among others. This contribution goes beyond the state of the art in bug reproduction (i.e., `Advscanner`), which only supports reproducing reentrancy bugs. We also conduct a comparative evaluation with it, which shows that, on reentrancy, `VeriExploit` performs on par with the prior work when evaluated on its dataset.

Formal Validation. We propose a formal model to validate the bug-reproduction process via automated reasoning. Previously, validating whether there exists a valid exploit trace between contracts often required manual inspection or dynamic execution that exhaustively traverses transactions to find a witness [2]. To address this, we provide bounded cross-contract verification (BCCV), which extends verifiers from bug detection to exploit validation. In brief, BCCV binds addresses and models cross-contract transactions to decide whether an adversarial contract can re-trigger the bug within the known contract set. We provide a proof sketch showing *soundness* (every reported trace under BCCV is valid) and *completeness* (if a witness exists within the known contract set, BCCV can find it).

Trace Enhancement. By leveraging LLMs and BCCV, `VeriExploit` enriches verifiers' counterexample traces. As shown in the motivational example in Section III, a common issue with reported traces is that they tend to be incomplete, especially when the attack flow involves external calls. This drawback prevents developers from reconstructing the full picture needed to simulate an exploit. `VeriExploit` synthesises a valid adversarial contract and uses BCCV to produce an extended trace; in Section VII-E, our reproduction of CVE-2021-34270 [4] shows how the missing steps are recovered.

Implementation & Evaluation. We build `VeriExploit` on two off-the-shelf verifiers and evaluate it on open-source benchmarks and real-world contracts. Across bug types, it attains high correctness with few reflection rounds and scales to complex transaction logic.

II. Background

A. Smart Contracts

Smart contracts are programs that run on the Ethereum Virtual Machine (EVM), a stack-based, deterministic engine replicated by all blockchain nodes; each deployed contract has a unique address [5]. Solidity is the most common language for EVM contracts: a contract groups state variables and functions (akin to a class) but has no single "main" entry—any account or contract can invoke it by sending a transaction to its address [6]. An *external call* occurs when one contract invokes another address [7]. In Solidity, this may appear as `target.foo(arg1)`, where `target` is an address; the compiler ABI-encodes the function selector and arguments into low-level call data. Low-level calls such as `addr.call(data)` similarly send raw bytes and, if no function matches, transfer control to `fallback()` or `receive()`. Thus, both high- and low-level external calls carry two essentials: (1) the target address and (2) the encoded call data (selector + arguments) [8].

B. Formal Verification

Formal verification [9] is a mathematical approach to analyse whether a program satisfies a formally defined property under all relevant execution conditions, which may be constrained based on the verification scope. Specifically, a safety property is a specification of a system's expected behaviour. It defines what the system should do and is used as a formal requirement to be verified, while a constraint is a restriction imposed on the system's behaviour during verification. It defines what the system must follow to ensure sound verification. When a property is found violated, most verification tools, known as verifiers, generate a counterexample that illustrates how the failure occurs. A counterexample is an execution stack trace that details the sequence of steps leading to a failure, commonly with additional information that includes violated safety property, location, vulnerability type, et al. This information is particularly helpful for developers to initiate the debugging [10].

C. Large Language Models

Large Language Models (LLMs) are transformer-based deep learning models whose self-attention mechanism conditions on previous tokens during sequence processing [11], [12]; their parallelizable training [13] on vast unlabelled corpora [14] makes them strong sequence generators across tasks, including code synthesis and mutation [15]–[17]. In our setting, LLMs serve as a program generator (mutation engine) to modify or synthesise code [17]. Prompt engineering—designing instructions and task data to steer the model without changing hyperparameters—constrains the response space towards desired outputs [18], and is standard in code generation where instructions specify the transformation goal while the data provides the concrete inputs to be transformed [14], [18], [19].

III. Motivation

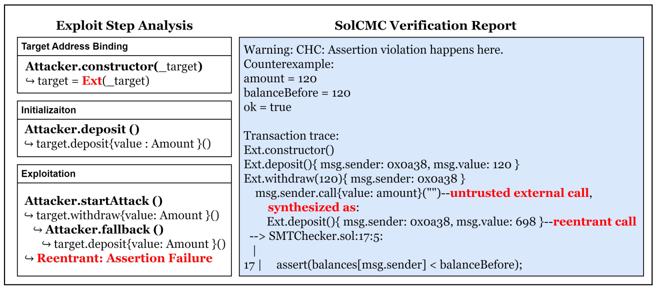
We start with an example to show why combining formal verification with large language models (LLMs) can help automatically reproduce bugs in smart contracts. We examine a smart contract named `Ext` in Listing 1. This contract has a common vulnerability called *reentrancy*. Reentrancy happens when a smart contract calls an external contract before updating its own state. This allows a malicious contract to repeatedly enter the vulnerable function before the first call completes. For illustration, the code ends with an assertion that the balance after withdrawal is less than the balance before; we use a violation of this assertion as a trigger to extract counterexamples that capture reentrant behaviour

Fig. 1 compares two approaches to bug reproduction: an explicit exploit analysis (left) and the counterexample provided by `SolCMC` [20] (also referred to as SMTChecker [21]), a state-of-the-art verifier (right). To simulate the exploit, we build an attacker contract named `Attacker`. The exploit involves three main steps:

Target Address Binding. The attacker identifies and connects to the vulnerable contract `Ext`.

```
1  pragma solidity ^0.8.29;
2  contract Ext {
3    mapping(address => uint256) public balances;
4    function deposit() external payable {
5      require(msg.value > 0);
6      balances[msg.sender] += msg.value;
7    }
8    function withdraw(uint256 amount) external {
9      require(amount > 0 && balances[msg.sender] >=
         amount);
10     uint256 balanceBefore = balances[msg.sender];
11     // Risk: External call jumping out and re-
         entry
12     (bool ok, ) = msg.sender.call{value: amount}(
         "");
13     require(ok);
14     balances[msg.sender] -= amount;
15
16     // Should never be violated unless reentrancy
17     assert(balances[msg.sender] < balanceBefore);
18   }
19 }
```

Listing 1. Reentrancy vulnerability example



Exploit Step Analysis

Target Address Binding
```
Attacker.constructor(_target)
↪ target = Ext(_target)
```

Initializaiton
```
Attacker.deposit ()
↪ target.deposit{value : Amount }()
```

Exploitation
```
Attacker.startAttack ()
↪ target.withdraw{value: Amount }()
  ↪ Attacker.fallback ()
    ↪ target.deposit{value: Amount }()
↪ Reentrant: Assertion Failure
```

SolCMC Verification Report

```
Warning: CHC: Assertion violation happens here.
Counterexample:
amount = 120
balanceBefore = 120
ok = true

Transaction trace:
Ext.constructor()
Ext.deposit(){ msg.sender: 0x0a38, msg.value: 120 }
Ext.withdraw(120){ msg.sender: 0x0a38 }
  msg.sender.call{value: amount}("")--untrusted external call,
    synthesized as:
    Ext.deposit(){ msg.sender: 0x0a38, msg.value: 698 }--reentrant call
 --> SMTChecker.sol:17:5:
  |
17 |    assert(balances[msg.sender] < balanceBefore);
```

Fig. 1. Exploit Analysis and SolCMC's Counterexample

Initialization. The attacker deposits ether into `Ext` to establish a balance. This step bypasses the condition check at line 9 in the `withdraw` function.

Exploitation. The attacker initiates the exploit by calling function `withdraw`. Within this function, an external call (`msg.sender.call{value: amount}("")`) transfers ether back to the attacker. Because this call has no function signature, it triggers the attacker's fallback function, which re-enters `deposit`, increasing the attacker's balance again. After this reentrant call completes, the execution returns to `withdraw`, causing the assertion to fail. In our discussion, the assertion failure serves to surface a witness; whether the execution is truly reentrant is determined by the call sequence (a callback before the state update), not by the assertion alone.

Next, comparing this exploit analysis with the verifier's counterexample, we have three observations. (1) The verifier provides useful debugging information: it identifies the violated assertion, vulnerability type, bug location, and transaction sequence causing the bug. (2) Some details are abstracted by the verifier, marked as "untrusted external call, synthesised as". Concretely, the tool reports an assertion violation and synthesises a callback that looks reentrant, but it does not by itself certify that the assertion uniquely indicates reentrancy. In other words, we cannot identify the attack flow—"call to external `fallback` and re-entry"—from the provided trace. (3) The verifier's trace only shows function calls within the vulnerable function and does not reveal the potential attacker's behaviour pattern (e.g., address binding, initialisation).

These findings motivate our bug reproduction to combine formal verification with large language models:

LLM-driven reproduction contract generation. A *reproduction contract* acts as an adversarial counterpart to the vulnerable contract—such as an `Attacker` contract. It is designed to reliably re-trigger the same identified bug. Formal verifiers provide sound but incomplete information. We leverage this rich, partial information to construct effective reproduction contracts.

Verification trace enrichment. A *reproduction trace* is a concrete sequence of transactions that reliably exploits the same bug. This trace enriches and clarifies the abstract transaction sequence provided by the verifier. Intuitively, it spells out who calls whom and in what order, turning the abstract report into a concrete, replayable flow.

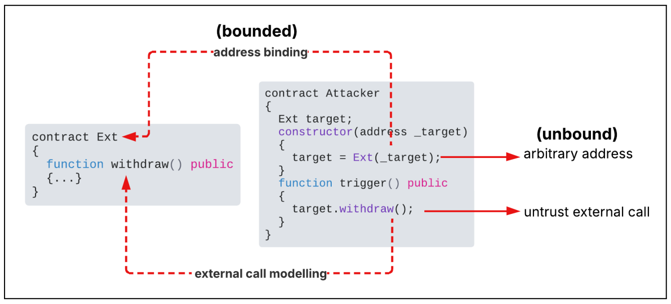
IV. Problem Formalization

We formalise our objective. Let C denote a smart contract; C_v is a known vulnerable contract and C_r the reproduction contract to be generated. A verifier's counterexample is $CE = (P, VT, L, T)$, where P is the violated safety property, VT the vulnerability type (e.g., overflow, reentrancy), $L = (c, f, \ell)$ the bug location (contract, function, line), and T the transaction trace. Given C_v and CE, we aim to automatically produce C_r and a reproduction counterexample $CE_r = (P, VT, L, T_r)$, where T_r is a *concrete* inter-contract call sequence that reproduces the same bug P at L; thus CE_r preserves (P, VT, L) while replacing the abstract trace T with T_r.

V. Bounded Cross-Contract Verification (BCCV)

We now formalise BCCV, a verification mode that enables vulnerability reproduction by producing concrete and valid traces across known contracts.

The goal of BCCV is to validate whether a reproduction contract C_r can successfully re-trigger a known bug in a vulnerable contract C_v. If successful, it outputs a trace T_r showing how the bug is exercised. This trace serves as a validated reproduction trace. If no violation is found, the attempt is considered unsuccessful.



```
contract Ext
{
  function withdraw() public
  {...}
}
```

```
contract Attacker
{
  Ext target;
  constructor(address _target)
  {
    target = Ext(_target);
  }
  function trigger() public
  {
    target.withdraw();
  }
}
```

Fig. 2. Bounded vs. Unbounded Verification

A. Concept Overview

In a typical exploit, interactions happen within a fixed system—such as an attacker contract calling a known vulnerable contract, or a reentrancy pattern where the callee calls back into the caller before the caller's state is safely updated.

The goal of Bounded Cross-Contract Verification (BCCV) is to formalise such systems and provide a verification procedure to validate exploitability. Specifically, BCCV introduces constraints that bind symbolic addresses to known contracts, allowing the verifier to analyse real interactions. As illustrated on the left in Fig. 2, binding the address `target` enables the external call to precisely match a known contract (e.g., `Ext`) and produce an actionable trace. This differs from traditional verification without such constraints: the verifier makes no assumptions about the code behind a symbolic address, so an external call (e.g., `target.withdraw()`) may be modelled as jumping to any unknown contract. While this over-approximation preserves soundness, it sacrifices precision and often yields vague reports such as "untrusted external call," as shown on the right in Fig. 2.

BCCV aims to provide an automatic and formally grounded method to validate exploitability via symbolic constraints. In theory, it can be instantiated on any verifier that meets the following mandatory prerequisites:

Unbounded reasoning: the host verifier offers either an unbounded-checking backend (e.g., constrained Horn clauses (CHC) [20]) or an unbounded-reasoning algorithm (e.g., k-induction [22]), enabling depth-unbounded exploration of attack steps within the fixed in-scope contract set after BCCV restricts call targets.

External-call binding: when given address/interface constraints, the verifier links the constrained addresses to concrete codes and dispatches external calls to these implementations during analysis—much like linking a library at analysis time (e.g. `SolCMC`'s trusted external calls [21]); we assume this engineering facility to instantiate BCCV's address bindings and entry-point fixings.

Witness extraction: the verifier can emit concrete counterexamples (states/calls) so that BCCV can return an executable cross-contract trace.

B. Formal Model

We first define key components of the model, and then state how BCCV modifies standard verification semantics.

a) Contract Set.: Let $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ be the bounded set of known contracts. Each C_i has a unique address $a(C_i)$.

b) Global State.: The state of the system is $S = (s_1, s_2, \ldots, s_n)$, where s_i represents the local state of contract C_i.

c) Call Event.: An external call in the system is denoted as $\text{Call}(C_i, \alpha, f, \bar{v})$, meaning contract C_i invokes function f with arguments $\bar{v}$ at address α.

d) Address-unconstrained semantics.: In traditional verification, α may point to any contract—known or unknown. The semantics permit

$$(S, \text{Call}(C_i, \alpha, f, \bar{v})) \longrightarrow_{\textsf{ua}} S'$$

where S' is computed using unknown or symbolic code. This abstraction prevents the verifier from inferring concrete interaction behaviour.

e) BCCV Rule.: BCCV restricts external calls to known addresses:

$$\frac{\alpha \in \mathcal{A}}{(S, \text{Call}(C_i, \alpha, f, \bar{v})) \longrightarrow_{bccv} S'}$$
$$\text{where } \mathcal{A} = \{a(C) \mid C \in \mathcal{C}\}$$

This ensures that external calls are dispatched to the concrete code of a contract in $\mathcal{C}$ (rather than to an abstract unknown callee) via the backend's external-call binding when address/interface constraints are provided; BCCV constrains only the admissible set $\mathcal{A}$, while the verifier performs the actual address-to-code association. A typical instance in bug reproduction is $\mathcal{A} = \{a(C_v), a(C_r)\}$, where C_v is the vulnerable contract and C_r the attacker/reproduction contract.

f) Trace Definition.: Let Σ_{bccv} denote all valid traces under BCCV. A trace $\pi \in \Sigma_{bccv}$ consists of a finite sequence of transitions, including inter-contract calls.

g) Violation Matching.: BCCV returns a trace π only if it re-triggers the original violation:

$$CE_r = (P, VT, L, T_r)$$

such that $CE_r.P = P,\ CE_r.VT = VT,\ CE_r.L = L.$

C. Entry Point Constraint

To ensure the bug is triggered through C_r, we require all transactions to originate from the attacker contract. Formally:

$$\forall\, \text{tx} \in \mathcal{T}, \quad \text{tx.origin} = a(C_r)$$

To ensure that the reproduction contract C_r is the source of all interactions, we fix the verification entry point. This means all transactions must originate from C_r. Formally, for each transaction tx $\in \mathcal{T}$, we require: $\forall \text{tx} \in \mathcal{T}, \quad \text{tx.origin} = a(C_r)$

This prevents the verifier from finding internal-only traces that do not reflect true exploitation. For example, in Fig. 2, this constraint ensures the call to `Ext.withdraw()` must begin from `Attacker.trigger()`.

D. Correctness and Termination

We now define and sketch proofs for two key properties of BCCV in the context of bug reproduction: soundness and completeness. Here, validation soundness means: if BCCV reports a reproduction trace, then the trace is valid and reproduces the same (P, VT, L) once the runtime state satisfies the trace's constraints; validation completeness means: if a witness exists using only the known contracts, BCCV can discover it under the assumptions stated below. These properties ensure that (1) any reproduction trace found by BCCV is real, and (2) if a valid reproduction exists over the known contracts, BCCV will discover it.

a) Definitions.: Let $CE = (P, VT, L, T)$ be the original counterexample from a verifier. Let $CE_r = (P_r, VT_r, L_r, T_r)$ be a candidate reproduction counterexample found by BCCV. We say CE_r is valid only if:

$$P_r = P, \quad VT_r = VT, \quad L_r = L$$

That is, it re-triggers the same property violation of the same type at the same location.

Let Σ_∞ be the set of traces under address-unconstrained semantics (no address/contract restrictions), and Σ_{bccv} be the set of traces under BCCV. Assumption (reasoning over known contracts): given a fixed set of known contracts $\mathcal{C}$ (with addresses $\mathcal{A}$), the backend explores arbitrarily long call/loop chains over $\mathcal{C}$ (e.g., via CHC or k-induction) and returns a witness if one exists over $\mathcal{C}$, unless it times out.

b) Validation Soundness.: If BCCV returns a trace π that re-triggers the bug at (P, VT, L), then this trace also exists in address-unconstrained semantics:

$$\forall \pi \in \Sigma_{bccv}, \text{ if } CE_r \text{ is valid, then } \pi \in \Sigma_\infty.$$

Proof Sketch. All steps in π use known contracts in $\mathcal{C}$ and execute real code. Thus, π remains valid in the conventional model and reproduces (P, VT, L) whenever the runtime state satisfies the trace's constraints.

c) Validation Completeness.: If there exists any trace $\pi_\infty \in \Sigma_\infty$ that uses only known contracts and re-triggers the bug (P, VT, L), then BCCV will find such a trace:

$$\text{If } \pi_\infty \in \Sigma_\infty, \ \forall \text{Call}(\cdot, \alpha, \cdot) \in \pi_\infty : \ \alpha \in \mathcal{A}, \\ \text{then } \pi_\infty \in \Sigma_{bccv}.$$

Proof Sketch. Since π_∞ uses only allowed addresses and contracts, it satisfies BCCV's constraints. Thus, BCCV will be able to execute the same steps and reproduce the same bug under the above assumption .

d) Termination.: BCCV does not alter the backend's termination behaviour; termination or timeouts follow from the solver configuration. In practice, BCCV terminates when the backend returns: (i) **SAT** with a reproduction trace π where $(P_r, VT_r, L_r) = (P, VT, L)$; (ii) **UNSAT**, indicating no witness over $\mathcal{C}$; or (iii) **TIMEOUT/UNKNOWN** due to resource limits (validation completeness applies to (i)–(ii)).

E. Summary

BCCV is a modelling constraint on top of existing verifiers. It is applied to formally validate exploitability and produce executable cross-contract traces; it is not a new backend nor a direct increment to any algorithm. In bug reproduction, BCCV plays two roles: (1) as a **validator**, confirming whether C_r can trigger the same bug (P, VT, L) in C_v; and (2) as a **trace generator**, outputting a valid trace T_r showing the exploit.

VI. VeriExploit

VeriExploit integrates BCCV-extended formal verification with LLM-based generation to reproduce bugs in Solidity smart contracts. As shown in Fig. 3, the process consists of three stages: generation (P_{ge}), validation (P_{va}), and self-reflection (P_{re}). In the generation stage, a verifier analyses a vulnerable contract and produces a counterexample. This counterexample is then combined with contextual vulnerability knowledge to prompt a large language model to generate a reproduction contract. The validation stage checks whether the generated contract compiles and successfully re-triggers the original bug using BCCV. If it fails, the self-reflection stage diagnoses the failure, collects the feedback and construct the reflection prompt for the next iteration—repeating this process until a valid reproduction is found or a time limit is reached.

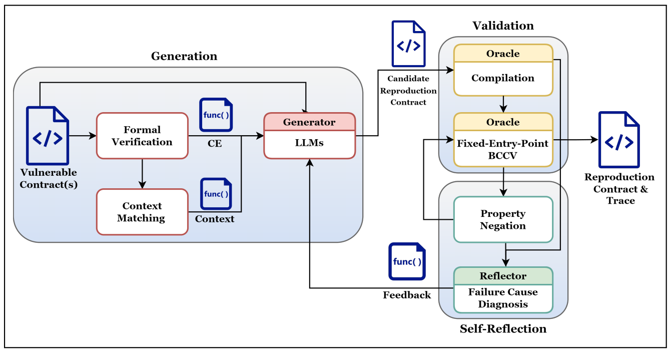
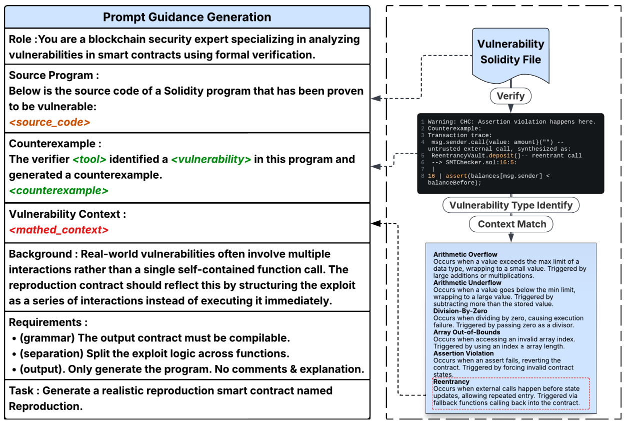


Fig. 3. Overall workflow of VeriExploit.

A. Generation Prompting



Prompt Guidance Generation

Role :You are a blockchain security expert specializing in analyzing vulnerabilities in smart contracts using formal verification.

Source Program :
Below is the source code of a Solidity program that has been proven to be vulnerable:
<source_code>

Counterexample :
The verifier *<tool>* identified a *<vulnerability>* in this program and generated a counterexample.
<counterexample>

Vulnerability Context :
<mathed_context>

Background : Real-world vulnerabilities often involve multiple interactions rather than a single self-contained function call. The reproduction contract should reflect this by structuring the exploit as a series of interactions instead of executing it immediately.

Requirements :
- **(grammar) The output contract must be compilable.**
- **(separation) Split the exploit logic across functions.**
- **(output). Only generate the program. No comments & explanation.**

Task : Generate a realistic reproduction smart contract named Reproduction.

Vulnerability Solidity File

Verify

```
Warning: CHC: Assertion violation happens here.
Counterexample:
Transaction trace:
 msg.sender.call{value: amount}("") --
untrusted external call, synthesized as:
 ReentrancyVault.deposit()-- reentrant call
 --> SMTChecker.sol:16:5:
 |
16 | assert(balances[msg.sender] <
balanceBefore);
```

Vulnerability Type Identify

Context Match

Arithmetic Overflow
Occurs when a value exceeds the max limit of a data type, wrapping to a small value. Triggered by large additions or multiplications.
Arithmetic Underflow
Occurs when a value goes below the min limit, wrapping to a large value. Triggered by subtracting more than the stored value.
Division-By-Zero
Occurs when dividing by zero, causing execution failure. Triggered by passing zero as a divisor.
Array Out-of-Bounds
Occurs when accessing an invalid array index. Triggered by using an index ≥ array length.
Assertion Violation
Occurs when an assert fails, reverting the contract. Triggered by forcing invalid contract states.
Reentrancy
Occurs when external calls happen before state updates, allowing repeated entry. Triggered via fallback functions calling back into the contract.

Fig. 4. Generation Prompt Construction

We denote the generation prompt as a tuple:

$$\langle C_v, CE, CTX, R, T \rangle$$

where: C_v is the vulnerable contract, $CE = (P, VT, L, T)$ is the counterexample produced by the verifier, CTX is the matched vulnerability context, R is the generation requirement, and T is the instruction task.

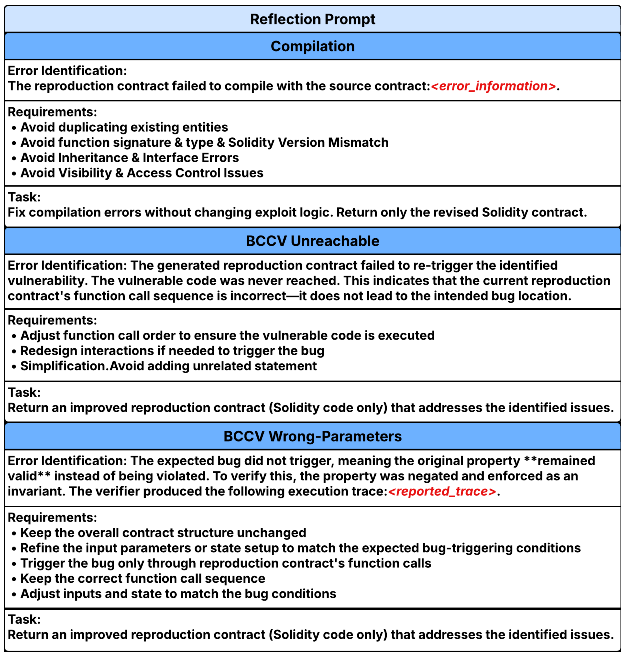
The context CTX consists of predefined templates stored in a local database. The idea is to provide contextual knowledge about how each vulnerability type typically manifests, so that the LLM can generate more realistic and structured exploits.

CTX is obtained by applying a deterministic matching function $\texttt{Match}(VT)$, which selects the appropriate context based on the vulnerability type VT:

$$CTX = \texttt{Match}(VT)$$

As shown in Fig. 4, the prompt includes both raw analysis data (code) and contextual guidance (CE, CTX) to inform LLM generation. We use the term *contextual information* to refer to any structured prompt content that provides background knowledge (e.g., VT), generation guidance (e.g., CTX), or post-failure feedback. The requirement R specifies the contract must compile, support multiple function-based interactions, and be compatible with the verifier. The prompt also enforces *interaction-separation*: the LLM must decompose the exploit into modular functions rather than placing all logic into a single function. The idea is that a realistic reproduction often involves multiple calls and steps, which are better modelled as separate interactions instead of a single monolithic execution.

The task T instructs the model to generate a reproduction contract named `Reproduction` that follows these constraints, without explanations or comments.



Reflection Prompt

Compilation

Error Identification:
The reproduction contract failed to compile with the source contract:*<error_information>*.

Requirements:
- Avoid duplicating existing entities
- Avoid function signature & type & Solidity Version Mismatch
- Avoid Inheritance & Interface Errors
- Avoid Visibility & Access Control Issues

Task:
Fix compilation errors without changing exploit logic. Return only the revised Solidity contract.

BCCV Unreachable

Error Identification: The generated reproduction contract failed to re-trigger the identified vulnerability. The vulnerable code was never reached. This indicates that the current reproduction contract's function call sequence is incorrect—it does not lead to the intended bug location.

Requirements:
- Adjust function call order to ensure the vulnerable code is executed
- Redesign interactions if needed to trigger the bug
- Simplification.Avoid adding unrelated statement

Task:
Return an improved reproduction contract (Solidity code only) that addresses the identified issues.

BCCV Wrong-Parameters

Error Identification: The expected bug did not trigger, meaning the original property **remained valid** instead of being violated. To verify this, the property was negated and enforced as an invariant. The verifier produced the following execution trace:*<reported_trace>*.

Requirements:
- Keep the overall contract structure unchanged
- Refine the input parameters or state setup to match the expected bug-triggering conditions
- Trigger the bug only through reproduction contract's function calls
- Keep the correct function call sequence
- Adjust inputs and state to match the bug conditions

Task:
Return an improved reproduction contract (Solidity code only) that addresses the identified issues.

Fig. 5. Self-Reflection Prompt

B. Self-Reflection Prompting

To guide contract refinement after a failed reproduction attempt, we construct a structured prompt for the LLM. As shown in Fig. 5, each prompt consists of three parts: error identification, revision requirements, and task specification. We denote this as $\langle EI, R, T \rangle$. The error type is first determined by checking whether the generated contract compiles. If it fails, the prompt addresses syntax, interface, or visibility issues. If it compiles, we run BCCV. When BCCV finds no violation, we apply property negation (Section VI-B) to assess whether the vulnerable location is reachable. If no trace is produced, we conclude the exploit structure is invalid. If a trace exists, we infer the parameters of the (last) calls to that function are incorrect. The reflection prompt uses this information to provide targeted feedback for correcting the reproduction contract, while preserving the original generation constraints.

a) Property Negation: If the BCCV verification passes, no property violation occurs, resulting in the absence of a counterexample or reproduction trace. We introduce **property negation** to address this limitation by explicitly forcing a verification failure at the previously identified vulnerability location. The core idea is that since the original property has been proven safe and can be established as invariant within the bounded system, its logical negation should trigger a violation if the vulnerable code is reachable.

b) Negation Semantics.: Let $P(L)$ be the original safety property encoded as an assertion at the bug location L. We construct a negated property:

$$Q(L) = \neg P(L)$$

and re-run BCCV with this modified assertion. If a counterexample trace π is found, the location L is reachable from C_r, suggesting the issue lies not in the contract logic but due to *wrong parameters* (at least in the final call). If no counterexample is produced, then L is *unreachable*, implying fundamental structural issues in C_r's interaction pattern.

c) Simplified Variant.: When direct property inversion is difficult to implement in the verifier, we use a conservative fallback: inserting an assertion `assert(false)` at location L, i.e.,

$$Q(L) = \texttt{false}$$

This guarantees a violation if the vulnerable code is reached. While this form lacks variable-level diagnostics, it is sufficient to verify reachability and aid reproduction refinement.

d) Termination.: `VeriExploit` terminates under either of two conditions: (1) BCCV finds a valid counterexample $CE_r = (P, VT, L, T_r)$, confirming successful reproduction; or (2) both the original and negated BCCV runs report no counterexample, indicating reproduction failure.

C. Termination

`VeriExploit` stops when one of the following occurs:

Successful Reproduction. BCCV finds a valid counterexample $CE_r = (P, VT, L, T_r)$, confirming that the reproduction contract successfully re-triggers the bug.

Reflection Limit Reached. The iterative refinement process (based on feedback and re-generation) exceeds a fixed retry threshold. In this case, `VeriExploit` terminates and reports failure.

D. Implementation

We prototype `VeriExploit` on two verifiers as backends: `SolCMC` [20] and `ESBMC` [23]. These two verifiers are selected because: (1) both can target diverse vulnerability classes, including arithmetic errors and assertion violations, among others; (2) both provide unbounded reasoning capabilities—SolCMC [20] encodes contracts as CHCs amenable to IC3-style reasoning [24], and ESBMC uses k-induction [22]; together, these ensure completeness for our reproduction task since any valid reproduction trace, regardless of its transaction length or loop depth, will eventually be discovered; (3) both can generate counterexamples when detecting bugs, though the resulting traces can be abstract or incomplete for reproduction.

We mainly extend the backends with two features: (i) BCCV, which binds symbolic call addresses to a finite set of known contracts in order to recover precise inter-contract traces, and (ii) property negation, which allows us to force reachability checks at target vulnerable locations. We take ESBMC as an example to explain our implementation. ESBMC already models Solidity/EVM external call mechanics (e.g., function-selector decoding and low-level *call* $\rightarrow$ *fallback/receive* resolution), so our BCCV layer only needs to rebind the symbolic call address α to a concrete set of contract addresses $\alpha(C)$, after which ESBMC dispatches to the correct function. For property negation, ESBMC represents every safety property as a C-style *assert*; we invert the Boolean expression inside the generated *assert* at location L to encode the negated property.

VII. Evaluation

We evaluate VeriExploit on open-source datasets to verify that it meets our objectives. We address three research questions:

RQ1 How effective and scalable is VeriExploit in reproduction—measured by success rate, structural validity, runtime, and the number of reflection iterations?
RQ2 How does each guidance and feedback component affect reproduction generation and reflection?
RQ3 How does VeriExploit perform compared to the prior state-of-the-art tool (Advscanner)?

To answer RQ1, we compare VeriExploit against a baseline that has no generation guidance or reflection feedback. For RQ2, we perform an ablation study, removing each component that provides contextual data in turn and measuring its impact. For RQ3, we conduct an external-dataset comparison on Advscanner's open-sourced reentrancy benchmark under aligned settings and report success rates. Finally, we include a case study of a real-world vulnerability (CVE-2021-34270 [4]) to show the exact form of a generated reproduction contract and its trace. This demonstrates that, despite being a prototype, VeriExploit works on practical smart contracts.

A. Benchmark Setup

Dataset Criteria Our benchmark has two parts: (i) subsets drawn from established vulnerability suites used in prior research, collectively referred to as an open-source benchmark corpus (**OSBC**); and (ii) an on-chain dataset (**REAL**) collected from real-world blockchains. These datasets are selected according to three principles: (1) they are publicly available; (2) each is explicitly pre-labelled with one or more vulnerability types to enable structured evaluation; and (3) the labelled vulnerability is detectable by the verifier (i.e., the verifier can report the target property violation in the contract).

Specifically, the **OSBC** includes contracts from Certora (SVB) [25], [26], SmartBug (**SBG**) [27], [28], smartcontract-benchmark (**SCB**) [29], [30], SolSee (**SEE**) [31], [32], JiuZhou (**JZU**) [33], [34], SolTG (**STG**) [35]–[37], and ESBMC-Solidity (**ESL**) [38], [39].

Additionally, we collect 30 on-chain contracts from real-world blockchains, denoted as **REAL**. This subset represents contracts that have historically caused major security incidents or posed significant security risks. The code of each contract can be publicly accessed via its on-chain address. The vulnerability for each contract has been pre-identified by authorised groups such as CVE [40], SWC [41], and OpenZeppelin Security [42], among others.

TABLE I
Structural metrics of the datasets. SLOC = source lines of code (excluding comments and blank lines); #Functions = number of function definitions per file; #Contracts = number of contract types per file; Avg = average; Med = median; Max = maximum.

	SLOC			**#Functions**			**#Contracts**		
	Avg	Med	Max	Avg	Med	Max	Avg	Med	Max
OSBC	127.5	56.0	427.0	23.0	9.0	79.0	2.1	1.0	7.0
REAL	186.4	134.5	674.0	23.9	19.5	99.0	4.6	4.0	14.0
Tot.	145.2	130.0	674.0	23.3	18.5	99.0	2.9	2.0	14.0

Complexity Table I shows the structural characteristics of the benchmark. **OSBC** mostly stands for easier cases but is specifically curated to include bug types that are compatible with the verifiers; while **REAL** tends to contain larger and more function-dense contracts than **OSBC**.

TABLE II
Distribution of Vulnerabilities Across Datasets

	SVB	**SBG**	**SEE**	**SCB**	**JZU**	**STG**	**ESL**	**REAL**	**Tot.**
AO	-	0(6)	0(1)	16(25)	0(1)	0(7)	0(2)	18	34(60)
AU	-	0(7)	-	17(25)	0(1)	0(2)	0(2)	1	18(38)
DV	-	-	-	-	1(1)	4(4)	2(2)	-	7(7)
OB	-	-	-	-	-	1(4)	5(5)	1	7(10)
AV	-	-	4(19)	1(1)	-	5(46)	1(1)	-	11(66)
RE	13(17)	0(31)	-	0(50)	-	-	-	10	23(108)
Tot.	13(17)	0(44)	4(20)	34(100)	1(3)	10(63)	8(12)	30	100(289)

Vulnerabilities The Overall dataset comprises 100 contracts, categorised into arithmetic overflow (**AO**), arithmetic underflow (**AU**), division by zero (**DV**), index access out of bound (**OB**), assertion failures (**AV**), and reentrancy issues (**RE**). Table II provides a detailed breakdown of the vulnerability distribution. The numbers in parentheses represent the original counts from the benchmark, while the outer values indicate the number of cases remaining after filtering based on the criteria mentioned above. A "**-**" denotes the absence of a particular vulnerability type in a given benchmark dataset.

Preprocessing All contracts are minimally modernised to satisfy the runtime requirements of both verifier backends (ESBMC and SolCMC). We update the pragma to Solidity $\geq$0.8.0 where necessary and replace any incompatible or unsupported syntax while preserving the core program logic. Note that, in theory, Solidity $\geq$0.8.0 reverts on arithmetic overflow/underflow by default, and such violations are therefore not reported by verifiers under default settings. To address this—and to reflect the pre-upgrade semantics—we explicitly enable overflow/underflow detection in both backends verifier during evaluation, so that the reproduced issues correspond to the original (pre-0.8.0) behaviour.

Additionally, since SolCMC does not natively provide a reentrancy verification target [21], we adopt two synthetic instrumentation-based strategies according to data sources. For the **OSBC** suites (e.g., **SVB**), we utilise their vendor-provided, *pre-instrumented* reentrancy checks—typically direct assertions over balance-related invariants. For the **REAL** cases, we instrument the reentrant-hosting functions identified by Slither (i.e., functions with external calls implicated in the vulnerability) with a lightweight lock-based assertion: on entry, we check that no prior activation is in progress, mark the region as active, execute the body, and clear the mark on exit. This ensures that: (*Soundness*) any genuine reentrant call necessarily overlaps an active execution and thus triggers the assertion, while sequential calls never do; (*Completeness*) for all guarded entry points, any re-entrant path will encounter the mark and violate the assertion. In the real world, this instrument represents a risky reentrant behaviour, which may not necessarily lead to balance loss. We then restrict verification to assertion-violation properties and re-verify each case, confirming that the counterexample indeed arises from reentrant behaviour.

Environment VeriExploit is evaluated with two off-the-shelf verifiers as backends: ESBMC (v7.9) and SolCMC (v0.8.30). For the LLM, we use GPT-4o with its default configuration; the temperature is 1.0. For each case, we conduct five trials, with each run using a timeout of 120 seconds and a reflection limit of 5. The experiments were conducted on an Ubuntu 22.04 system equipped with an Intel Core i9-13900H CPU @ 2.60GHz (14 cores, 20 threads) and 32 GB of RAM. The material and artifact are publicly available at Zenodo[1].

B. RQ1: Effectiveness

We measure effectiveness by two criteria: (1) **success rate**, the percentage of cases where a valid reproduction contract is generated (both overall and after each reflection iteration), and (2) **structural validity**, whether the exploit logic is split across functions rather than packed into a single call.

We also include a simple "Baseline" variant. The idea is to measure performance when no verification-based guidance information is provided during the generation and reflection state. Specifically, in (P_{ge}), its prompt contains only the role, the vulnerable contract, and the task ($R + C_v + T$) from Figure 4. It omits both the counterexample (CE) and the matching context (CTX). Moreover, no (P_{re}) is applied.

TABLE III
COMPARISON OF SUCCESS RATE

	Baseline	**SOLCMC**	**ESBMC**
AO	7.65%	74.12%	88.24%
AU	23.33%	55.56%	74.44%
DZ	28.57%	71.43%	100.00%
OB	14.29%	85.71%	85.71%
AV	1.82%	74.55%	98.18%
RE	7.83%	44.35%	80.00%
Tot.	11.80%	64.60%	85.60%

1) Success rates.: Table III shows success rates for each bug type. SolCMC and ESBMC reach 64.60% and 83.23%, respectively. The LLM-Only baseline only reaches 11.80%. This comparison confirms that verification-based feedback is important in guiding the LLM. The verification-assisted methods succeed across all bug categories. Particularly, in reentrancy—the hardest case—ESBMC hits 80.00%, followed by SolCMC 44.35%, and baseline 11.80%. This shows our approach works on different kinds of vulnerabilities. We attribute the gap between ESBMC and SolCMC to backend capabilities: under BCCV constraints, SolCMC more often returns "could not prove" or times out, which aligns with SolCMC's own note that enabling external-call binding can "make the analysis much more computationally costly" [21].

TABLE IV
IMPACT OF SELF-REFLECTION ITERATIONS

	1	**2**	**3**	**4**	**5**
SOLCMC	58.80%	59.60%	61.20%	63.80%	64.60%
ESBMC	65.00%	71.60%	80.20%	81.60%	85.60%

Next, we evaluate how self-reflection iterations affect success rates. The result is illustrated in Table IV. As reflection iterations increase, ESBMC improves sharply in the first three rounds (65.00%$\rightarrow$71.60%$\rightarrow$80.20%), then levels off (81.60%$\rightarrow$85.60%). SolCMC shows smaller, steady gains across rounds (58.80%$\rightarrow$64.60% by iteration 5). This convergence indicates that most benefits arrive early; small retry caps (around three rounds) capture the bulk of improvements, with diminishing returns thereafter for both backends.

2) Structural validity.: Next, we check structural validity. We call an exploit *self-contained* if all its logic is in one function, and *interaction-separated* if it spreads across multiple functions. We prefer interaction-separated, since it helps to reveal multiple attack paths and better matches real-world patterns. Classifying full interaction separation can be subjective, but identifying self-contained exploits is straightforward. We therefore use "self-contained" as our oracle. We manually inspect each generated contract. SolCMC and ESBMC each produce *zero* self-contained contracts across 323 and 428 valid

[1]https://zenodo.org/records/15560759

cases, respectively. As a comparison, among the 59 valid cases from the Baseline, 15 failed to achieve functional separation.

3) Scalability: To examine the scalability, we split cases by the median SLOC: contracts at or below the median are *Simp*, and those above are *Comp*. We compare success rate (SC), average per-case runtime in seconds (RT), and average reflection iterations (IT) in Table V. All metrics include failed cases. ESBMC backend maintains a relatively high success rate on *Comp* (81.20%, down from 90.00%), while SolCMC decreases from 77.20% to 52.00%. We attribute this to ESBMC performing better on interaction-heavy cases such as reentrancy (see Table III). For both backends, *Comp* incurs higher runtime (SolCMC: 25.41$\rightarrow$73.88 s; ESBMC: 23.74$\rightarrow$45.78 s). Iterations rise on ESBMC (0.91$\rightarrow$1.33) but slightly drop on SolCMC (1.12$\rightarrow$1.06); for SolCMC, the sharp RT increase suggests more runs hit the time budget or return inconclusive results, limiting opportunities for further reflection.

TABLE V
SCALABILITY BY SIZE STRATA. **SIMP.** = SLOC $\leq$ DATASET MEDIAN; **COMP.** = SLOC $>$ DATASET MEDIAN. SC = SUCCESS RATE (%); RT = AVERAGE RUNTIME (S); IT = AVERAGE REFLECTION ITERATIONS.

	SC (%)		**RT** (s)		**IT**	
	Simp	Comp	Simp	Comp	Simp	Comp
SolCMC	77.20	52.00	25.41	73.88	1.12	1.06
ESBMC	90.00	81.20	23.74	45.78	0.91	1.33

Our answer to RQ1 is that VeriExploit is broadly effective across diverse vulnerability classes; it converges within a small number of self-reflection rounds, making bounded retry caps practical; it preserves structurally realistic exploit patterns; and it remains capable on larger contracts, delivering reasonable accuracy at the cost of moderate overhead.

C. RQ2: Ablation Study

We study how different components of VeriExploit affect the success rate of reproduction contracts. Table VI shows the results of removing contextual components that provide guidance or feedback. For clarity, we explain each setting and why some are omitted:

- $-CE$**,** CTX: Removes both the counterexample (*CE*) and context (*CTX*). The LLM sees only the raw source code and generation task prompt.
- $-RE$: Disable the self-reflection stage. No feedback is provided when validation fails.
- $-RE$**,** CTX: Removes context only, but under the same condition as $-RE$. This isolates the role of CTX.
- $-NE$: Disable BCCV-negation reflection. This blocks the interaction trace used by self-reflection, while still keeping other components.

Note that we do not isolate *CE* alone because it is needed to generate the CTX. Similarly, compilation must succeed before BCCV can run, so we do not study the removal of Compile in isolation.

TABLE VI
IMPACT OF COMPONENT REMOVAL ON SUCCESS RATE(IN %)

	-CE, CTX		**– RE**		**– RE, CTX**		**– NE**	
	Sol	ESB	Sol	ESB	Sol	ESB	Sol	ESB
AO	12.35	41.18	36.47	42.35	32.35	40.59	30.59	51.18
AU	35.56	32.22	56.67	46.67	52.22	54.44	50.00	62.22
DV	85.71	85.71	60.00	60.00	80.00	74.29	88.57	85.71
OB	77.14	68.57	40.00	40.00	22.86	40.00	77.14	51.43
AV	61.82	83.64	72.73	81.82	47.27	67.27	78.18	83.64
RE	0.87	29.57	46.96	46.96	0.87	40.00	36.52	45.22
Tot.	29.00	46.60	48.40	49.60	33.00	48.20	48.00	57.80

Generation vs. Reflection. We compare removing all contextual guidance during generation (*–CE, CTX*) versus removing reflection feedback (*–RE*). Table VI shows that removing verification-guided generation (*–CE, CTX*) lowers success rates from the full pipeline (Table III) to 29.00% (SolCMC) and 46.60% (ESBMC). Removing the self-reflection component (*–RE*) yields 48.40% for SolCMC and 49.60% for ESBMC. This indicates that self-reflection is important for improving results and handling complex vulnerabilities like Assertion Violations and Reentrancy, while verification-guided generation is likewise critical for overall success.

Verification vs. Context within Generation (Under –RE). We inspect the effect of removing verification guidance (*CE*) versus contextual knowledge (*CTX*) during generation, under the condition without self-reflection (*–RE*). Comparing *–RE* to *–RE, CTX*, removing contextual guidance further reduces success from 48.40% to 33.00% (SolCMC) and from 49.60% to 48.20% (ESBMC). The drop is pronounced for Reentrancy on SolCMC (to 0.87%), and ESBMC also declines, showing that contextual information provides important background, especially for interaction-heavy bugs.

BCCV-Negation. We also inspect the BCCV symbolic trace guidance (*–NE*). Removing BCCV yields 48.00% (SolCMC) and 57.80% (ESBMC) in total. The impact is notable for Reentrancy: for SolCMC, from 44.35% (full pipeline; Table III) to 36.52% under *–NE*; for ESBMC, from 80.00% to 45.22%. This shows BCCV is important for providing detailed traces that support effective reflection on complex bugs.

Our answer to RQ2 is that self-reflection is the key driver of effectiveness; verification-guided generation also contributes, and contextual knowledge helps when reflection is limited; BCCV-negation's trace is especially important for interaction-heavy bugs; taken together, counterexamples, context, reflection, and BCCV-negation are complementary.

D. RQ3: Comparison with Advscanner

To our knowledge, Advscanner is the only prior system close to Solidity bug reproduction; it focuses on reentrancy and generates adversarial contracts via static analysis and

LLMs (see §VIII). Advscanner publishes its dataset[2] but no executable artifact. Therefore, we evaluate VeriExploit on the public Advscanner corpus. Since SolCMC does not natively report reentrancy, we use ESBMC as the backend (enabling --reentry-check).

To enable a fair external-dataset comparison, we align the settings with Advscanner's best-performing setup, including the GPT-4.0 API, temperature 1.0, at most 6 reflection rounds. Advscanner does not state a time budget; guided by its maximum used runtimes, we set a per-round cap of 180 s. We use the success rate as the criterion, averaged across five trials. All logs are added to the artifact [1].

TABLE VII
COMPARISON OF SUCCESS RATE ON ADVSCANNER DATASET

AdvScanner (reported)	VeriExploit (ESBMC)
88.48%	90.91%

Table VII shows the comparison result on 66[3] cases from Advscanner's dataset. Based on Advscanner's artifact spreadsheet, we recompute its success rate as 88.48%. For comparison, on the same 66 cases, VeriExploit achieves a success rate of 90.91% across five trials. This comparison indicates that VeriExploit performs on par with the state of the art.

Our answer to RQ3 is that VeriExploit performs on par with prior work (Advscanner) on its reentrancy corpus under matched settings.

E. Case Study

We present a case study to show (1) how VeriExploit handles a real-world vulnerability, (2) what the resulting reproduction contract and trace look like and (3) how to explain them. Our example is the integer-overflow bug in the Doftcoin project [43], listed as CVE-2021-34270 [4]. In Doftcoin's *mintToken* function, the owner can mint arbitrary amounts. If they mint more than the maximum *uint256* value, an overflow resets a user's balance to zero, enabling unauthorized fund drains.

We first updated the original Doftcoin contract (e.g. removed deprecated keywords) so both SolCMC and ESBMC can complete unbounded verification—this is a necessary precondition for running VeriExploit. Listing 2 shows a sample reproduction contract; Listing 3 and 4 show a simplified version of the extracted counterexamples. The full contracts and traces are available at [1].

Analysing the reproduction contract reveals a typical attack flow. First, the attacker calls setupExploit, minting maxUint to the victim's balance and causing an overflow that resets it to zero. Then, the attacker calls triggerExploit, minting one token to the attacker's account to complete the

```
contract Reproduction {
  Doftcoin public vulnerableContract;
  address public attacker;
  constructor(address _vulnerableContract) {
   vulnerableContract = Doftcoin(
       _vulnerableContract);
  }
  function setupExploit(address _target) public {
    uint256 maxUint = type(uint256).max;
    vulnerableContract.mintToken(_target, maxUint
        );
  }
  function triggerExploit() public {
    vulnerableContract.mintToken(attacker, 1);
  }
  function checkBalance(address _target) public
      view returns (uint256) {
    return vulnerableContract.balanceOf(_target);
  }
}
```

Listing 2. Reproduction Contract

exploit. Optionally, the attacker can call checkBalance to verify the result.

Here, we observe two points. (1) The reproduction contract includes extra elements that are not needed to trigger the bug but aid testing. For example, the checkBalance function provides a way to confirm success, even though it does not affect the overflow itself. (2) The separated interaction structure allows multiple valid attack paths. For example, one can call triggerExploit before setupExploit, or call setupExploit twice, both of which can cause overflow.

```
Warning: CHC: Overflow (resulting value larger
    than 2**256 - 1) happens here.
Counterexample:
vulnerableContract = 0, attacker = 0x0
_target = 0x0
balanceOf_target = 282

Transaction trace:
Reproduction.constructor(0x0)
State: vulnerableContract = 0, attacker = 0x0
Reproduction.setupExploit(0x0)
    Doftcoin.mintToken(0x0,
        115792089237316195423570985008687790
    7853269984665640564039457584007913129639935)
        -- trusted external call
   --> contract.sol:166:9:
    |
162 |         balanceOf_target += _mintedAmount;
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Listing 3. SolCMC's Counterexample

Next, we combine the Doftcoin and reproduction contracts and run BCCV on SolCMC and ESBMC to generate counterexamples, shown in 3 and 4, respectively. Despite having different format, the information provided from these two trace are similar: Vunerablity types is "Overflow", property (implicitly) is $balanceOf\_target + \_mintedAmount < 2^{256}$, and is located in contract Doftcoin, function mintToken and line 166. Although their formats differ, both traces report the same vulnerability type "Overflow," use the property

[2]No executable artifact was available at the time of writing.

[3]12 cases were not included as they did not satisfy our dataset criteria.

```
 1  file contract.sol line 183 function setupExploit
 2  ----------------------------------------------
 3    maxUint = 0xFFFFFFFFFFFFFFFFFFFFF
 4    FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
 5  file contract.sol line 161 function mintToken
 6  -----------------------------------------------
 7    balanceOf_target = 2
 8  file contract.sol line 165 function mintToken
 9  ----------------------------------------------
10  Violated property:
11    file contract.sol line 166 function mintToken
12  Stack trace:
13    sol:@C@Doftcoin@F@mintToken#291 at file
          contract.sol line 184 function
          setupExploit
14    sol:@C@Reproduction@F@setupExploit#330 at
15    arithmetic overflow on add
16    !overflow("+", balanceOf_target, _mintedAmount)
```

Listing 4. ESBMC's Counterexample

`balanceOf_target + _mintedAmount < 2^256`.

Both tools simulate a similar attack flow, with one difference. That is, `SolCMC`'s trace records an initial value of `balanceOf_target` as 282 (line 5 of Listing 3), while `ESBMC`'s trace records `balanceOf_target` as 2 (line 3 of Listing 4). This initialisation is performed implicitly by calling `triggerExploit` repeatedly. Then, both invoke `setupExploit` (lines 10 and 13), causing the overflow.

In conclusion, we note four facts. (1) Each trace binds external calls to specific functions and tracks state across contracts. (2) The reported call order can be non-intuitive, such as triggering the exploit before setup. (3) Only one counterexample is reported, even when multiple valid sequences exist. (4) Some details may be implicit or omitted because of the verifier's internal abstractions.

F. Threat to Validity

Our evaluation is subject to certain constraints due to verification tool capabilities and dataset availability. First, `SolCMC` will produce inconclusive results, reporting that a property violation "might happen here" due to computational resource limits or problem complexity. To solve this, we use pre-labelled vulnerabilities to confirm the presence of actual violations, ensuring the correctness.

Second, the two verifiers used in this evaluation are still in development, limiting support for certain language features and sometimes failing to detect vulnerabilities. This constrains our benchmark selection to reliably processable contracts, reducing the suitable candidates.

Additionally, the distribution of test contracts across vulnerability categories is uneven, with categories like division-by-zero cases being less frequent than others. This is due to the scarcity of publicly available verified contracts that meet our selection criteria rather than a design choice. While a perfectly balanced dataset is ideal, we prioritize realism and availability over artificial adjustments.

VIII. Related Work

LLMs have been applied to smart-contract security for analysing and generating adversarial contracts. Two representative lines are `Advscanner` [2] and `Skyeye` [3]. `Advscanner` targets reentrancy only and uses static analysis to extract attack flows that guide LLM-based attacker contracts. Unlike `VeriExploit`, `Advscanner` (i) limits its scope to reentrancy; (ii) relies on templates with fixed initial logics to constrain generation—its ablation shows templates boost success but reduce diversity of the produced exploits. and (iii) validates exploits via local EVM deployment/simulation (py-evm), whereas we validate by formal verification with BCCV, which preserves soundness and completeness. `Skyeye` monitors adversarial smart contracts around deployment time to detect threats before attacks; this detection task is orthogonal to our goal of formal, reproducible bug triggering.

IX. Conclusion

We presented `VeriExploit`, a framework that combines formal verification with LLMs to automatically reproduce smart-contract vulnerabilities by synthesising reproduction contracts and validated traces across multiple bug types. At its core, Bounded Cross-Contract Verification (BCCV) binds addresses to a known contract set and, given an attacker harness, models cross-contract transactions to return proof-carrying counterexamples. `VeriExploit` repairs failures via iterative self-reflection; costs are dominated by a few reflection rounds, and scalability is currently limited by the verifier on very large code, so we target medium-sized contracts.

In the future, we plan to replace whole-contract regeneration during reflection with partial generation—re-synthesising only suspected faulty fragments guided by property-negation traces—to reduce cost and improve scalability to larger real-world contracts. We currently abstract away live on-chain state and focus on code-level vulnerabilities: actively encoding real-time chain state into reproduction is beyond our scope due to operational and ethical risks; however, BCCV guarantees that once a deployed contract's state matches the reproduced trace state, the violation will manifest, making state-aware reproduction a promising direction.

X. Acknowledgment

We thank the anonymous reviewers for their constructive feedback. This work was supported by XC5 Hong Kong Limited (CR No. 2647858). Any opinions, findings, conclusions, or recommendations expressed in this paper are solely those of the authors and do not necessarily reflect the views or policies of the sponsor.

References

[1] 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, pp. 3–1.

[2] Y. Wu, X. Xie, C. Peng, D. Liu, H. Wu, M. Fan, T. Liu, and H. Wang, "Advscanner: Generating adversarial smart contracts to exploit reentrancy vulnerabilities using llm and static analysis," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1019–1031.

[3] N. Gosala, K. Petek, P. L. Drews-Jr, W. Burgard, and A. Valada, "Sky-eye: Self-supervised bird's-eye-view semantic mapping using monocular frontal view images," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 14 901–14 910.

[4] CVE Program, "Cve-2021-34270," https://www.cve.org/CVERecord?id=CVE-2021-34270, accessed: 2025-03-01.

[5] L. Alt and C. Reitwiessner, "Smt-based verification of solidity smart contracts," in *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV 8*. Springer, 2018, pp. 376–388.

[6] R. Modi, *Solidity Programming Essentials: A beginner's guide to build smart contracts for Ethereum and blockchain*. Packt Publishing Ltd, 2018.

[7] P. Praitheeshan, L. Pan, X. Zheng, A. Jolfaei, and R. Doss, "Solguard: Preventing external call issues in smart contract-based multi-agent robotic systems," *Information Sciences*, vol. 579, pp. 150–166, 2021.

[8] Q. Lyu, C. Ma, Y. Shen, S. Jiao, Y. Sun, and L. Hu, "Analyzing Ethereum smart contract vulnerabilities at scale based on inter-contract dependency," *CMES-Computer Modeling in Engineering & Sciences*, vol. 135, no. 2, 2023.

[9] O. Hasan and S. Tahar, "Formal verification methods," in *Encyclopedia of Information Science and Technology, Third Edition*. IGI Global Scientific Publishing, 2015, pp. 7162–7170.

[10] T. Wu, P. Schrammel, and L. C. Cordeiro, "Wit4java: A violation-witness validator for java verifiers (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 484–489.

[11] K. Han, A. Xiao, E. Wu, J. Guo, C. Xu, and Y. Wang, "Transformer in transformer," *Advances in neural information processing systems*, vol. 34, pp. 15 908–15 919, 2021.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," Aug. 2023, arXiv:1706.03762 [cs]. [Online]. Available: http://arxiv.org/abs/1706.03762

[13] R. Cotterell, A. Svete, C. Meister, T. Liu, and L. Du, "Formal Aspects of Language Modeling," Apr. 2024, arXiv:2311.04329 [cs]. [Online]. Available: http://arxiv.org/abs/2311.04329

[14] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[15] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Melbourne, Australia: IEEE, May 2023, pp. 1482–1494. [Online]. Available: https://ieeexplore.ieee.org/document/10172803/

[16] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," Sep. 2021, arXiv:2109.00859 [cs]. [Online]. Available: http://arxiv.org/abs/2109.00859

[17] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating Language Models for Efficient Code Generation," Aug. 2024, arXiv:2408.06450. [Online]. Available: http://arxiv.org/abs/2408.06450

[18] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine-tuning: An empirical assessment of llms for code," 2025. [Online]. Available: https://arxiv.org/abs/2310.10508

[19] V. A. Braberman, F. Bonomo-Braberman, Y. Charalambous, J. G. Colonna, L. C. Cordeiro, and R. de Freitas, "Tasks people prompt: A taxonomy of llm downstream tasks in software verification and falsification approaches," 2024. [Online]. Available: https://arxiv.org/abs/2404.09384

[20] L. Alt, M. Blicha, A. E. Hyvärinen, and N. Sharygina, "Solcmc: Solidity compiler's model checker," in *International Conference on Computer Aided Verification*. Springer, 2022, pp. 325–338.

[21] SMTChecker, "SMTChecker," https://docs.soliditylang.org/en/latest/smtchecker.html, accessed: 2025-03-01.

[22] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of c programs via k-induction," *International journal on software tools for technology transfer*, vol. 19, no. 1, pp. 97–114, 2017.

[23] T. Wu, X. Li, E. Manino, R. S. Menezes, M. R. Gadelha, S. Xiong, N. Tihanyi, P. Petoumenos, and L. C. Cordeiro, "ESBMC v7.7: Efficient concurrent software verification with scheduling, incremental SMT and partial order reduction (competition contribution)," in *Proceedings of the 31st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2025.

[24] S. Prabhu, G. Fedyukovich, K. Madhukar, and D. D'Souza, "Specification synthesis with constrained horn clauses," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1203–1217.

[25] M. Bartoletti, F. Fioravanti, G. Matricardi, R. Pettinau, and F. Sainas, "Towards benchmarking of solidity verification tools," *arXiv preprint arXiv:2402.10750*, 2024.

[26] Scontracts-verification-benchmark, "Contracts verification benchmark," https://github.com/fsainas/contracts-verification-benchmark, accessed: 2025-03-01.

[27] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 530–541.

[28] SmartBugs, "Smartbugs-curated," https://github.com/smartbugs/smartbugs-curated, accessed: 2025-03-01.

[29] Z. Wei, X. Zhang, J. Sun, Z. Zhang, and L. Zhu, "A comparative evaluation of automated analysis tools for solidity smart contracts," 2024. [Online]. Available: https://arxiv.org/abs/2310.20212

[30] bit-smartcontract-analysis, "smartcontract-benchmark," https://github.com/bit-smartcontract-analysis/smartcontract-benchmark, accessed: 2025-03-01.

[31] S.-W. Lin, P. Tolmach, Y. Liu, and Y. Li, "Solsee: A source-level symbolic execution engine for Solidity," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1687–1691.

[32] SolSEE, "Solsee: Source-level symbolic execution for solidity," https://github.com/ntu-SRSLab/SolSEE, accessed: 2025-03-01.

[33] H. Chu, P. Zhang, H. Dong, Y. Xiao, and S. Ji, "Deepfusion: Smart contract vulnerability detection via deep learning and data fusion," *IEEE Transactions on Reliability*, pp. 1–15, 2024.

[34] JiuZhou, "Jiuzhou: Smart contract vulnerability detection via deep learning and data fusion," https://github.com/xf97/JiuZhou, accessed: 2025-03-01.

[35] K. Britikov, I. Zlatkin, G. Fedyukovich, L. Alt, and N. Sharygina, "Soltg: A chc-based solidity test case generator," in *International Conference on Computer Aided Verification*. Springer, 2024, pp. 466–479.

[36] Z. Peng, X. Yin, R. Qian, P. Lin, Y. Liu, C. Ying, and Y. Luo, "Soleval: Benchmarking large language models for repository-level solidity code generation," 2025. [Online]. Available: https://arxiv.org/abs/2502.18793

[37] SolTG, "Soltg: Solidity test case generator," https://github.com/BritikovKI/solidity_testgen, accessed: 2025-03-01.

[38] K. Song, N. Matulevicius, E. B. de Lima Filho, and L. C. Cordeiro, "ESBMC-solidity: an SMT-based model checker for Solidity smart contracts," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 65–69.

[39] ESBMC, "Esbmc: Efficient SMT-based bounded model checker," https://github.com/esbmc/esbmc/, accessed: 2025-03-01.

[40] CVE, "Cve," https://www.cve.org/, accessed: 2025-09-03.

[41] SWC Registry, "Swc registry: Smart contract weakness classification and test cases," https://swcregistry.io/, accessed: 2025-09-07.

[42] OpenZeppelin, "Openzeppelin security audits," https://www.openzeppelin.com/security-audits, accessed: 2025-09-03.

[43] Etherscan, "Smart contract doftcoin on Etherscan," https://etherscan.io/address/0x2eb9cc28c34c6d427aac9f259ee5c4b33f1c4448#code, accessed: 2025-09-03.