Scutum: Temporal Verification for Cross-Rollup Bridges via Goal-Driven Reduction

YANJU CHEN, University of California, Santa Barbara
JUSON XIA, Orbiter Finance
BO WEN, Orbiter Finance
KYLE CHARBONNET, Ethereum Foundation
HONGBO WEN, University of California, Santa Barbara
HANZHI LIU, University of California, Santa Barbara
YU FENG, University of California, Santa Barbara

Scalability remains a key challenge for blockchain adoption. Rollups—especially zero-knowledge (ZK) and optimistic rollups—address this by processing transactions off-chain while maintaining Ethereum's security, thus reducing gas fees and improving speeds. Cross-rollup bridges like Orbiter Finance enable seamless asset transfers across various Layer 2 (L2) rollups and between L2 and Layer 1 (L1) chains. However, the increasing reliance on these bridges raises significant security concerns, as evidenced by major hacks like those of Poly Network and Nomad Bridge, resulting in losses of hundreds of millions of dollars. Traditional security analysis methods such as static analysis and fuzzing are inadequate for cross-rollup bridges due to their complex designs involving multiple entities, smart contracts, and zero-knowledge circuits. These systems require reasoning about temporal sequences of events across different entities, which exceeds the capabilities of conventional analyzers.

In this paper, we introduce a scalable verifier to systematically assess the security of cross-rollup bridges. Our approach features a comprehensive multi-model framework that captures both individual behaviors and complex interactions using temporal properties. To enhance scalability, we approximate temporal safety verification through reachability analysis of a graph representation of the contracts, leveraging advanced program analysis techniques. Additionally, we incorporate a conflict-driven refinement loop to eliminate false positives and improve precision. Our evaluation on mainstream cross-rollup bridges, including Orbiter Finance, uncovered multiple zero-day vulnerabilities, demonstrating the practical utility of our method. The tool also exhibited favorable runtime performance, enabling efficient analysis suitable for real-time or near-real-time applications.

1 Introduction

As blockchain technology evolves, scalability remains a persistent challenge for mainstream adoption. Rollups—particularly zero-knowledge (ZK) rollups and optimistic rollups—have emerged as leading solutions to this problem, enabling greater throughput by processing transactions off-chain while maintaining the security of the Ethereum mainnet. These rollups rely on cryptographic proofs or game-theoretic mechanisms to batch transactions, drastically reducing gas fees and improving transaction speeds. To fully harness the benefits of these solutions, users need a way to transfer assets seamlessly across various Layer 2 (L2) rollups and between L2 and Layer 1 (L1) chains. Cross-rollup bridges, like Orbiter Finance [23], provide a vital infrastructure to facilitate these transfers. For instance, Orbiter Finance supports multiple rollups including zkSync and Arbitrum, allowing asset transfers with transaction times as low as 10–20 seconds and minimal fees [23]. However, as the adoption of these bridges grows, so too do concerns around their security.

Authors' Contact Information: Yanju Chen, yanju@ucsb.edu, University of California, Santa Barbara; Juson Xia, zerokpunk@orbiter.finance, Orbiter Finance; Bo Wen, wenbo@orbiter.finance, Orbiter Finance; Kyle Charbonnet, kylecharbonnet@gmail.com, Ethereum Foundation; Hongbo Wen, hongbowen@ucsb.edu, University of California, Santa Barbara; Hanzhi Liu, hanzhi@cs.ucsb.edu, University of California, Santa Barbara; Yu Feng, yufeng@cs.ucsb.edu, University of California, Santa Barbara.

The critical nature of rollup bridges has made them a prime target for security exploits. In recent years, high-profile hacks and vulnerabilities have exposed the fragility of these systems. For example, in August 2021, the Poly Network hack resulted in the theft of over $600 million due to a vulnerability in the bridge's smart contract logic. Similarly, Nomad Bridge was compromised in 2022 due to an initialization bug, leading to losses exceeding $190 million. These examples illustrate how security breaches in cross-chain or cross-rollup bridges can have catastrophic financial consequences. The inherent complexity of cross-rollup bridges, which combine cryptographic proofs, smart contracts, and off-chain actors, exacerbates the difficulty of ensuring their security. Therefore, it is paramount to develop robust, systematic methods to analyze and guarantee their security.

Although several approaches—such as static analysis and fuzzing—have been proposed to secure blockchain bridges, these techniques are ill-suited for cross-rollup bridges. General blockchain bridges, typically implemented purely in smart contracts, can be effectively analyzed using these methods. However, cross-rollup bridges involve a more intricate design, often incorporating multiple entities and combining smart contract code with zero-knowledge circuits. This hybrid design introduces a range of complexities that traditional analyzers cannot effectively address. For example, smart contracts may process an unbounded number of transactions, which need to be soundly handled by the bridge verifier. Additionally, many functional properties in cross-rollup bridges require reasoning about temporal sequences of events from different entities, which exceeds the capabilities of off-the-shelf safety verifiers. Static analyzers tend to over-approximate, leading to numerous false positives, while dynamic fuzzing struggles to cover the vast behavioral space of these systems, often failing to detect subtle vulnerabilities arising from interactions between different entities.

In this paper, we introduce a scalable verifier to systematically assess the security of cross-rollup bridges. Our approach is distinct in several key aspects. First, we present the first comprehensive multi-model framework that captures both the individual behaviors of standard cross-rollup bridge entities and their complex interactions using temporal properties. Second, recognizing the scalability challenges of classical temporal verifiers, we over-approximate the problem of temporal safety verification with reachability analysis by analyzing a graph representation of the original contracts. This enables us to directly leverage state-of-the-art program analysis methods. Finally, our approach incorporates a conflict-driven refinement loop, allowing us to eliminate false alarms and achieve high precision in our security analysis.

To demonstrate the effectiveness of our approach, we evaluated it on several mainstream cross-rollup bridges, including Orbiter Finance and others. Our analysis revealed multiple zero-day vulnerabilities, highlighting the practical utility of our method. Furthermore, our tool achieved favorable performance in terms of runtime, enabling real-time or near-real-time analysis of bridge systems.

In summary, this paper makes the following contributions:

- We present the first multi-model symbolic reasoning framework specifically tailored for cross-rollup bridges.
- We introduce novel abstract semantics to address the computational challenges of analyzing large-scale systems like cross-rollup bridges.
- We implement a conflict-driven refinement loop to ensure high precision and reduce false positives during security analysis.
- Our evaluation uncovered critical vulnerabilities in widely-used cross-rollup bridges, highlighting both the necessity and effectiveness of our approach.

2 Background

In this section, we briefly elaborate some background on blockchains and the rollups built on top of them. To connect rollups from different blockchains, we then introduce the cross-rollup bridge and its basic mechanism.

2.1 Blockchain and Rollups

Blockchain and Ethereum. Blockchain functions as a decentralized record-keeping platform that chronicles and disseminates transaction data among multiple users. It is an expand-only chain of interconnected blocks, managed by a consensus mechanism, where each block contains a collection of transactions. Among various blockchain systems, Ethereum [29] is the first blockchain capable of storing, managing, and running Turing-complete scripts, termed *smart contracts*. Ethereum operates on a comprehensive state system updated via transaction execution. The transactions are initiated by and received by users through their accounts. Ethereum has two principal types of accounts: those owned by users and those governed by smart contracts, each associated with a distinct *address*. Besides making transactions, users can also develop customized smart contracts that are programmed to execute transactions autonomously.

Scaling up blockchains with rollup solutions. Rollups are designed to increase the throughput of blockchains. A typical rollup processes transactions off-chain and *roll (or bundle) them up* into a single transaction, which is later posted to the blockchain. This reduces the load on the main network while preserving the security and decentralization provided by the underlying blockchain.

There are two prevailing types of rollups. An *optimistic rollup* assumes transactions are valid by default and allows challenge to incorrect data through submission of a fraud proof, and a *zero-knowledge (ZK) rollup* generates and posts ZK proofs to verify the correctness of off-chain transactions.

Rollups are built in a hierarchical way in that they naturally connect with the chain they build upon, but not with each other. As the total value locked (TVL) of each rollup grows more than 1500% over the past year, it's of growing demand that a fast and reliable yet affordable way can serve for assets and data transfer between rollups, and that's what a cross-rollup bridge is for.

2.2 Cross-Rollup Bridges

A cross-rollup bridge is a mechanism that enables the transfer of assets or data between different rollups (i.e., layer 2 scaling solutions built on blockchains). Rollups are designed to offload transaction processing from the main blockchain (layer 1) by batching transactions and executing them off-chain, while still relying on layer 1 for security. Since rollups operate independently, they need a bridge to facilitate communication and asset movement between them.

A typical workflow in cross-rollup bridges. Figure 1 shows a high-level overview of a typical cross-rollup bridge [1] that transfers assets between users from rollups of different chains. As different chains/rollups post and finalize transactions in an asynchronous way, the core task of a cross-rollup bridge is to ensure the integrity of the asset transfer activity, which is covered by its on-chain and off-chain components and services. A basic bridge workflow can be described by the following four steps:

[1]The bridge example infrastructure is distilled from Orbiter cross-rollup protocol (https://www.orbiter.finance/). Without loss of generality, other bridging solutions can find similar core structures.

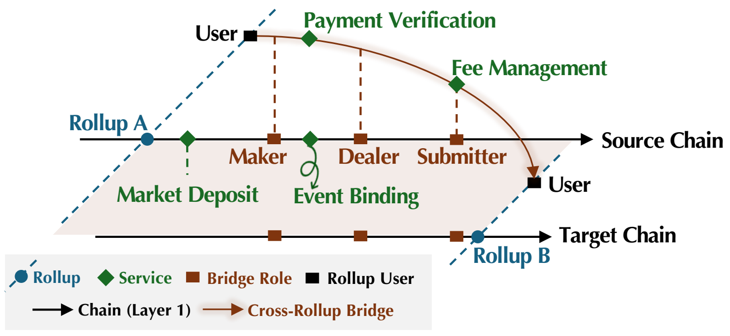


Fig. 1. A higher-level overview of a typical cross-rollup bridge between multiple blockchains.

(1) **Source Initialization** When a user launches an asset transfer request, an on-chain maker from the source chain responds by issuing an asset transfer transaction using the decentralized front-ends provided by a dealer. A chain event is then emitted as evidence of request initialization.
(2) **Target Respondence** Once the off-chain event binding service captures the source event, a transaction manager invokes the on-chain maker from the target chain for a paired transfer transaction that deposits the assets to the designated user. As a result, an event is expected on the target chain as an evidence of respondence to the source chain request.
(3) **Payment Verification** With both evidence events from the source and target chains, the off-chain prover then composes proofs for both transactions and sends them for on-chain verification. Such a procedure is called payment verification, where the proofs are usually generated by cryptographic protocols such as zero-knowledge proofs.
(4) **Finalization** If the payment verification is successful, states of both chains are finalized by their submitters, with the guidance of the fee management service that distributes revenue and benefits for dealers and makers; otherwise, transactions and bridge states will be rolled back.

The workflow secures the integrity of cross-rollup asset transfer by 1) association and finalization of the paired transactions on both chains with event binding mechanism, and 2) joint validation of both transactions by cryptographic protocols.

Vulnerabilities in cross-rollup bridges. Given the complexity of the async interactions between multiple on-/off-chain components and services involved in a typical workflow of a cross-rollup bridge, keeping the bridge infrastructure bug-free is critical but non-trivial. For example, our empirical study of the open-source repos related to the key components and services of Orbiter Finance reveals that, of all the bug-fixing commits, more than 50% require reference to design specification; in other words, these bugs involve deep logical issues that cannot be detected by existing tools tailored for common vulnerabilities.

Figure 2 illustrates an example transfer logic in Orbiter's cross-rollup protocol, where different roles (e.g., sender/receiver, makers and transaction manager) have to involve in a sequence of operations in order (marked as ❶, ❷, ❸ and ❹) to complete the process, which exposes potential attack surfaces amid these steps. For example, an attacker can insert a fake maker for carrying out step ❸. Such a malicious maker can then compromise the entire process by transferring assets on

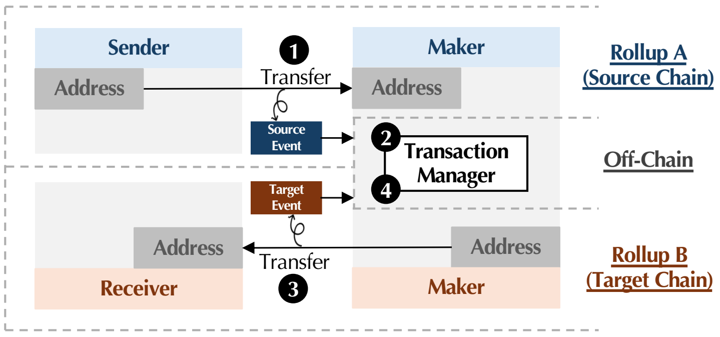


Fig. 2. An example transfer logic in Orbiter's cross-rollup protocol, where different roles are involved in a complex sequence of operations, exposing potential attack surfaces in various steps.

the target chain to unsafe addresses, causing an unrecoverable invalid state on the bridge with financial loss [2].

To detect for such a bug, one needs to reason about behavior of different on-/off-chain components from different chains and rollups against logical specification, which existing tools and approaches would find non-trivial to address.

3 Overview

In this section, we begin by specifying our problem scopes and demonstrating a logical bug that we dubbed as ChallengeEscape that we adapted from a real-world bridge. We then iterate the technical challenges and limitations while addressing it with existing solutions, which motivate the design of Scutum. We illustrate at the end from a high-level perspective how it takes for Scutum to detect the ChallengeEscape bug.

3.1 Preliminaries and Problem Scope

We describe a cross-rollup bridge $\mathbb{B} = (\mathbb{C}, \mathbb{O})$ by its on-chain components (usually smart contracts) $\mathbb{C}$ and off-chain components $\mathbb{O}$. On-chain components are executed in the virtual environment of the blockchain (e.g., Ethereum Virtual Machine, EVM) which ensures the consistency of the code execution and machine states in a decentralized manner.

A bridge may contain bugs as it usually involves interactions between various components to achieve certain business logic, which is better quantifiable by temporal logic according to an existing study [24]. Given a temporal specification ϕ and an initial state S_0 for the blockchain, we detect for violation of ϕ over the bridge $\mathbb{B}$. As ϕ is a temporal property, the violation to ϕ usually corresponds to a sequence of transactions that breaks such property.

3.2 Motivating Example

We illustrate a motivating example of how components in a bridge interact with each other, and how this allows an attack to construct a sequence of transactions (i.e., an attack) that triggers the bug and brings damage to the system.

The arbitration procedure in a bridge. For a bridge-related transaction that has been posted to the chain, a user can challenge its authenticity by submitting a proof that validates otherwise

[2]A brief description of this bug and its fix can be found in the following official commit: https://github.com/Orbiter-Finance/OB_ReturnCabin/commit/36c735d.

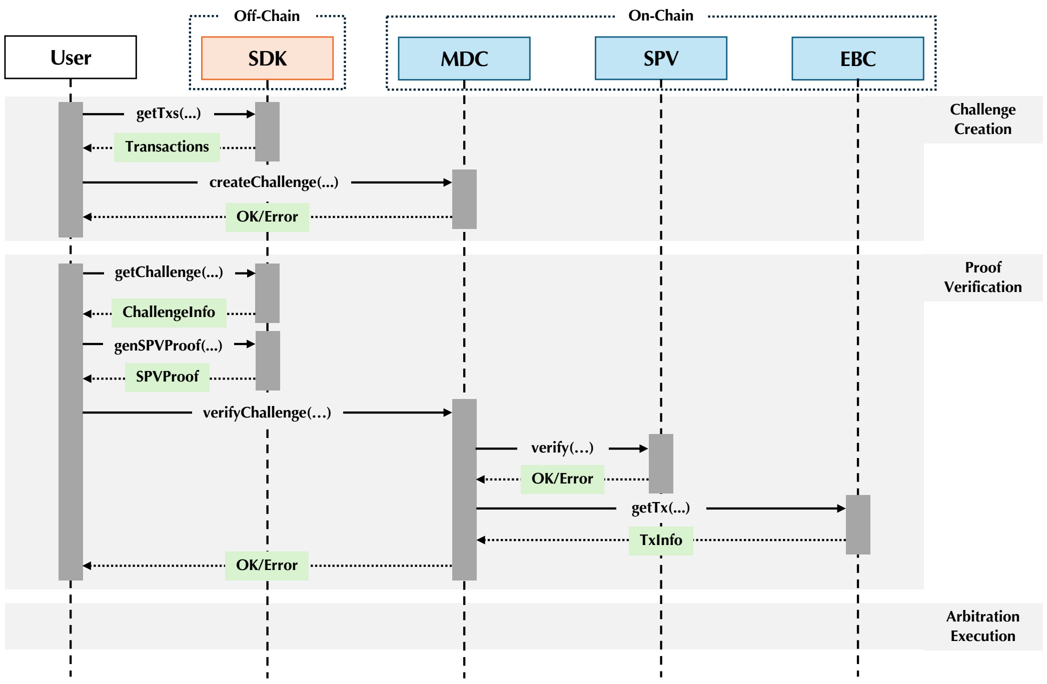
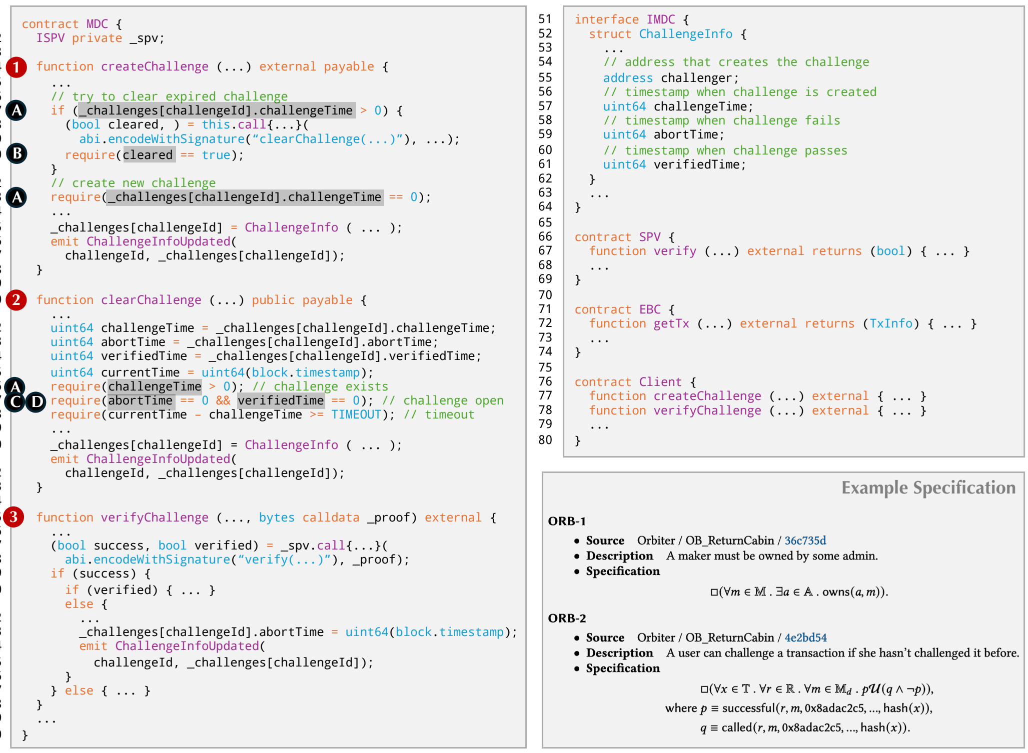


Fig. 3. An example (partial) sequence diagram for a bridge's arbitration procedure. An user can interact with the bridge via either an on-chain component (e.g., EOAs, agent contracts) or off-chain component (e.g., client, SDK, etc.); SDKs are deployed as off-chain components, and Maker Deposit Contract (MDC), Simplified Payment Verification (SPV) and Event Binding Contract (EBC) are all deployed as on-chain components.

and have the transaction undone by the bridge. Such a procedure, also known as the arbitration, is the core part of reaching consensus between different parties within a bridge.

Figure 3 presents a (partial) sequence diagram illustrating how a typical arbitration procedure works, which contains at least three phases involving interactions between different parties:

- In the *challenge creation* phase, when a dispute occurs for a transaction, to start the arbitration process, a user first retrieves the information of the target transaction from the off-chain bridge SDK. A special handle that describes the arbitration case, called *challenge*, is then created when the user directly interacts with the maker (deployed as the MDC on-chain component) and tracked within.
- In the *proof verification* phase, the bridge verifies the user's claim in the challenge. In a restricted time frame after the challenge has been created, the user generates a *Simplified Payment Verification* proof (created and secured by cryptographic algorithms, e.g., zero-knowledge proofs) that encodes the key information of the target transaction, and submits it to the maker (i.e., the MDC contract). The maker then queries the on-chain verification service (i.e., the SPV contract) about the correctness of the submitted proof, and retrieves related information from the event binding service (i.e., the EBC contract) for a cross-checking of the blockchain states. A final decision is then made and returned to the user based on facts collected.
- In the *arbitration execution* phase, the arbitration result is then applied to all affected parties, with payments and fees allocated or re-allocated based on the arbitration result.



```
contract MDC {
  ISPV private _spv;

  function createChallenge (...) external payable {
    ...
    // try to clear expired challenge
    if (_challenges[challengeId].challengeTime > 0) {
      (bool cleared, ) = this.call{...}(
        abi.encodeWithSignature("clearChallenge(...)"), ...);
      require(cleared == true);
    }
    // create new challenge
    require(_challenges[challengeId].challengeTime == 0);
    ...
    _challenges[challengeId] = ChallengeInfo ( ... );
    emit ChallengeInfoUpdated(
      challengeId, _challenges[challengeId]);
  }

  function clearChallenge (...) public payable {
    ...
    uint64 challengeTime = _challenges[challengeId].challengeTime;
    uint64 abortTime = _challenges[challengeId].abortTime;
    uint64 verifiedTime = _challenges[challengeId].verifiedTime;
    uint64 currentTime = uint64(block.timestamp);
    require(challengeTime > 0); // challenge exists
    require(abortTime == 0 && verifiedTime == 0); // challenge open
    require(currentTime - challengeTime >= TIMEOUT); // timeout
    ...
    _challenges[challengeId] = ChallengeInfo ( ... );
    emit ChallengeInfoUpdated(
      challengeId, _challenges[challengeId]);
  }

  function verifyChallenge (..., bytes calldata _proof) external {
    ...
    (bool success, bool verified) = _spv.call{...}(
      abi.encodeWithSignature("verify(...)"), _proof);
    if (success) {
      if (verified) { ... }
      else {
        ...
        _challenges[challengeId].abortTime = uint64(block.timestamp);
        emit ChallengeInfoUpdated(
          challengeId, _challenges[challengeId]);
      }
    } else { ... }
  }
  ...
}
```

```
interface IMDC {
  struct ChallengeInfo {
    ...
    // address that creates the challenge
    address challenger;
    // timestamp when challenge is created
    uint64 challengeTime;
    // timestamp when challenge fails
    uint64 abortTime;
    // timestamp when challenge passes
    uint64 verifiedTime;
  }
  ...
}

contract SPV {
  function verify (...) external returns (bool) { ... }
  ...
}

contract EBC {
  function getTx (...) external returns (TxInfo) { ... }
  ...
}

contract Client {
  function createChallenge (...) external { ... }
  function verifyChallenge (...) external { ... }
  ...
}
```

Example Specification

ORB-1

- **Source** Orbiter / OB_ReturnCabin / 36c735d
- **Description** A maker must be owned by some admin.
- **Specification**

$$\square(\forall m \in \mathbb{M} \,.\, \exists a \in \mathbb{A} \,.\, \mathrm{owns}(a, m)).$$

ORB-2

- **Source** Orbiter / OB_ReturnCabin / 4e2bd54
- **Description** A user can challenge a transaction if she hasn't challenged it before.
- **Specification**

$$\begin{gathered}\square(\forall x \in \mathbb{T} \,.\, \forall r \in \mathbb{R} \,.\, \forall m \in \mathbb{M}_d \,.\, p\,\mathcal{U}(q \wedge \neg p)),\\ \text{where } p \equiv \mathrm{successful}(r, m, \mathrm{0x8adac2c5}, ..., \mathrm{hash}(x)),\\ q \equiv \mathrm{called}(r, m, \mathrm{0x8adac2c5}, ..., \mathrm{hash}(x)).\end{gathered}$$

Fig. 4. Code snippets (simplified) showing key paths for arbitration process with some of the related temporal specifications and their natural language description. The code snippets contain a vulnerability that allows random address to completely block the arbitration process of a given transaction by deliberately failing the verification of transaction's challenge.

As the arbitration procedure involves various interactions between multiple parties, which opens up a wide attack interface for malicious participants, it's crucial but non-trivial for developers to get rid of all bugs, especially for logical ones.

Meanwhile, the sequential nature of the business logic reflected in these procedures makes temporal specification a best fit for quantifying the behavior of a bridge. We can derive some of the temporal properties to make sure of the correctness of the arbitration process, for example:

- *"A maker must be owned by some admin at all times"*, which can be formulated by:

$$\square(\forall m \in \mathbb{M} \,.\, \exists a \in \mathbb{A} \,.\, \mathrm{owns}(a, m))$$

  with $\mathbb{M}$ denoting all makers, $\mathbb{A}$ for all admins, and the predicate $\mathrm{owns}(a, m)$ describes the ownership of a over m. The temporal operator $\square$ (always) is used here.
- *"A user can always challenge a transaction if she hasn't challenged it before."*, which can be formulated by:

$$\begin{gathered}\square(\forall x \in \mathbb{T} \,.\, \forall r \in \mathbb{R} \,.\, \forall m \in \mathbb{M}_d \,.\, p\,\mathcal{U}(q \wedge \neg p)),\\ \text{where } p \equiv \mathrm{successful}(r, m.\mathrm{createChallenge}, x),\\ q \equiv \mathrm{called}(r, m.\mathrm{createChallenge}, ..., x)\end{gathered} \tag{1}$$

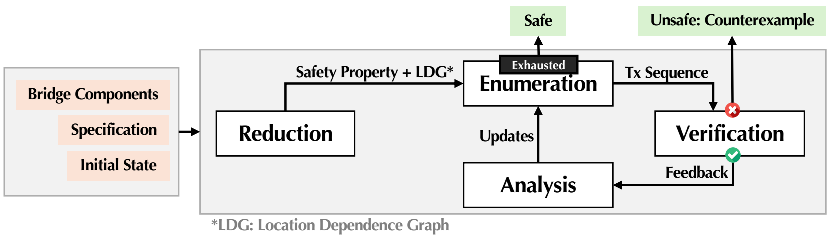


Fig. 5. Framework overview of Scutum.

with $\mathbb{T}$ denoting all finalized transactions, $\mathbb{R}$ denoting all users, and $\mathbb{M}$ denoting all makers. The predicate successful$(\cdot)$ returns whether a function can be successfully invoked and returned without reverting, and the predicate called$(\cdot)$ indicates whether a function has been called before the current time step. A temporal operator $\mathcal{U}$ (until) is used to describe an event must hold until another happens; in this example, p must hold until $q \wedge \neg p$ happens.

The* ChallengeEscape *vulnerability. The ChallengeEscape vulnerability is adapted from a real-world bug reported in a bridge [3]. It allows malicious attackers to completely block the challenge creation against a given transaction for any other users in the challenge creation phase; that is, it violates the specification formulated in Equation 1.

We show how this happens via some of the key code snippets in Figure 4. An attacker can first invoke the `createChallenge` function (line 4) on the MDC contract that initializes the challenge information into the storage mapping `_challenges` (line 15). Following the arbitration procedure set in Figure 3, the attacker then invokes the `verifyChallenge` function (line 35) with a faked proof that eventually fails the verification check. As shown by line 43, when a proof fails the verification, the `abortTime` field of the challenge is set to the current timestamp rather than `0`. This thus causes the follow-up challenge creation from a different user blocked, because the attempt to clear the challenge (line 8-9) will always fail due to the fact that `clearChallenge` function marks a non-zero value of `abortTime` as a signal for the challenge remaining open, and thus `createChallenge` refuses to create a new challenge for anyone since the call to `clearChallenge` always fails (line 10).

Technical challenges. To verify the correctness of the arbitration process and detect for the ChallengeEscape bug with the given temporal specification in Equation 1, a general-purpose algorithm usually takes two steps to reason about this:

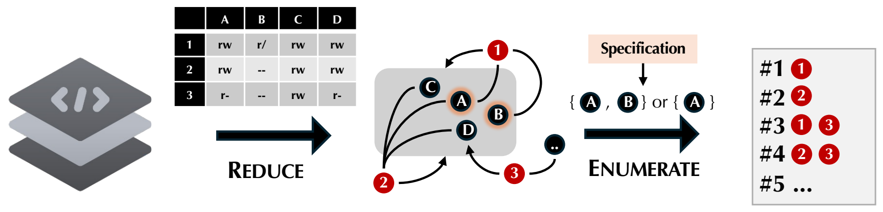
- Reduction: It first converts the liveness property in the temporal specification to an equivalent safety property which should always hold for arbitrary transactions. Such approach is widely used in temporal verification problems for various domains [12, 13, 24]. Following them, we can reduce Equation 1 into the following:

$$\Box(\forall x \in \mathbb{T} \,.\, \forall r \in \mathbb{R} \,.\, \forall m \in \mathbb{M}_d \,.\, (p \wedge \neg q) \oplus (\neg p \wedge q)),$$

  which describes a safety property that should always be satisfied.
- Enumeration & Verification: The reduced safety property is then checked against different transaction sequences emitted by the target system.

While there are infinite amount of transaction sequences, verification on real-world systems is usually infeasible without proper abstractions. This is especially challenging for systems like

[3]A brief description of the original bug and its fix can be found in the following official commit: https://github.com/Orbiter-Finance/OB_ReturnCabin/commit/4e2bd54.



	A	B	C	D
1	rw	r/	rw	rw
2	rw	--	rw	rw
3	r-	--	rw	r-

(a) bridge source code (b) location dependence graph (c) enumerated sequences

Fig. 6. Reduction from büchi automaton to transaction dependence graph for enumeration. (a): the büchi automaton generated from the bridge system; (b) a transaction dependence graph inferred by analysis through the dependence relations from the büchi automaton; (c) the transactions generated for verification by enumerating edge covers for the critical locations.

cross-rollup bridge that involve non-trivial interactions between multiple parties. On the one hand, it inevitably brings scalability issues for the general-purpose temporal verification approach due to complex interactions; on the other hand, cryptographic operations in a blockchain's virtual machine also create excessive computational overheads that makes it difficult to reason about logical bugs.

3.3 Our Approach: Reasoning with SCUTUM

Instead of enumeration over the infinite search space of transaction sequences, which in most cases won't scale, our approach, SCUTUM, performs a smart enumeration that combines 1) a *goal-driven* reduction that locates a *sufficient* subset of the search space, and 2) a conflict-driven loop that learns to prune and explore the search space more efficiently.

As shown in Figure 5, SCUTUM starts with a goal-driven reduction process that not only reduces the temporal specification into a safety property, but also produces a *transaction dependence graph (TDG)* that describes relations between critical functions (transactions) that could affect the truth value of the reduced specification. Then the enumeration of transaction sequences can be formulated as an optimal reachability problem over TDG. Enumerated transaction sequences that are proven safe against the reduced specification will be further analyzed to provide updated preference to the enumeration.

As an example, we show in Figure 6 the key steps taken by SCUTUM for detection of the ChallengeEscape vulnerability.

- SCUTUM first obtains a büchi automaton Q that models the behavior of the bridge $\mathbb{B}$ (as shown in Figure 6(a)), and follows the approach discussed in existing approaches [12, 13, 24] to convert the temporal specification ϕ to a safety property ϕ'.
- Based on the expressions (especially predicates) within ϕ', SCUTUM analyzes the automaton Q and identifies *critical locations* whose valuation could change the truth value of ϕ'. In the ChallengeEscape example, four locations Ⓐ, Ⓑ, Ⓒ and Ⓓ are identified (as marked out in Figure 4) since they fall into the expressions consumed by the `require` operator, which if failed would revert the transaction, causing the predicates successful and called to fail. The functions that contain references to those critical locations are thus marked out as *critical functions* by ❶, ❷ and ❸.
- SCUTUM then infers a transaction dependence graph (as shown in Figure 6(b)) that describes the write-before-read relations between the critical functions over the critical locations.

- Based on the specification ϕ', Scutum then enumerates candidate transaction sequences that could trigger the value changes of one or more critical locations. In this case, it's Ⓐ and Ⓑ as the goal is to find a transaction sequence that eventually fails the execution of createChallenge, which reads Ⓐ and Ⓑ. The transaction sequences proposed are shown in Figure 6(c).
- Scutum performs verification over enumerated transaction sequence in order to find a model that satisfies $\neg\phi'$. In this example, transaction sequence #1 and #2 are not satisfied given the above query. Scutum performs a conflict analysis over the results and provides feedback to the enumeration loop for improved preference over the transaction sequence search space. Finally, Scutum finds a counter-example when provided with transaction sequence #3, which is (❶, ❸), thus proving that the original bridge is unsafe against the specification ϕ.

4 Model and Query of System Behavior

In this section, we introduce Scutum's built-in language $\mathcal{L}_s$ for modeling the system's behavior and constructing verification queries. We then describe a vanilla solution for performing safety verification on top of the programs written in the $\mathcal{L}_s$ language, exposing several limitations and challenges that our proposed verification algorithm addresses. As the verification algorithm is the core contribution of this paper, we defer a detailed discussion to Section 5.

There are two parts of the $\mathcal{L}_s$ language, namely the behavioral modeling language $\mathcal{S}_b$ and the query language $\mathcal{S}_q$.

4.1 Modeling of System Behavior

As a cross-rollup bridge is a system composed by components on and off chain, verification over queries that quantify the behavior of such a system requires a domain-specific language $\mathcal{S}_b$ capable of modeling those components. We start by a definition of the program state that $\mathcal{S}_b$ operates on.

Definition 4.1 (Program State). A *program state* (or also referred to as *execution context*) $\Gamma = \langle \varepsilon, \mu, \zeta, \omega \rangle$ is a four-tuple of stack ε, memory μ, storage ζ and event pool ω.

The program state describes a set of abstractions for both on and off chain components, where both stack ε, memory μ and storage ζ are typical internal data structures for program execution, and the event pool ω directly inherits from the blockchain state where *events* emitted during blockchain and system initialization and execution are stored in. Since an event is a special case of program execution trace, for off-chain components, a subset of these traces related to the final goal is tracked as events and store in the event pool.

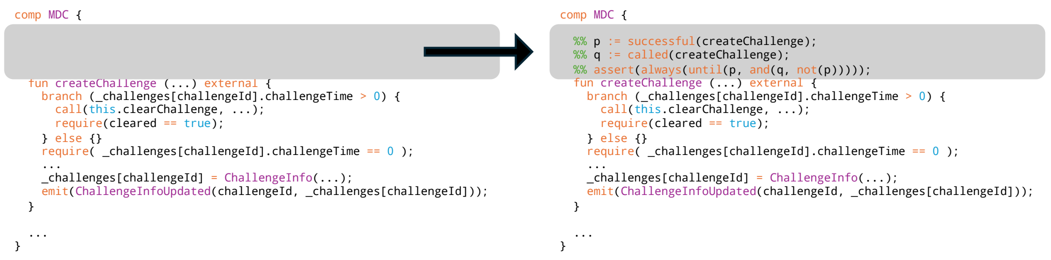
Syntax of the $\mathcal{S}_b$ language. We then show the grammar of a set of key language constructs for $\mathcal{S}_b$ in Figure 7, and elaborate its key designs as follows:

- **High-Level Program and Memory Structure** A Scutum component is defined by the comp construct, to which an identifier and a set of statements are provided. From within the component, functions are then defined using the fun construct. Each component has its own local scope with standalone memory and storage shared by all its functions. Each function has its own local stack. All components read and write to a shared event pool.
- **Expressions and Access Paths** $\mathcal{S}_b$ supports a set of expressions covering operations from different components within a bridge system. In particular, to refer to a specific location on a program state, or a specific function on a component, $\mathcal{S}_b$ introduces a set of access paths that are compatible across on and off chain components.
- **On and Off Chain Calls** For example, invocation of function from other components (i.e., external function) is modeled by xcall, and call is used for those functions within the same

m	$::=$			$\mathsf{comp}(i, s*)$	**Component**
s	$::=$				**Statements:**
	$\mid$	b	$\equiv$	$\{s*\}$	block
	$\mid$	g	$\equiv$	$\|e\|b$	guarded block
	$\mid$	f	$\equiv$	$\mathsf{def}(i, z, z, b)$	function definition
	$\mid$	a	$\equiv$	$\mathsf{assign}(\varrho, e)$	assignment
	$\mid$	r	$\equiv$	$\mathsf{branch}(b, g*)$	branch
	$\mid$	l	$\equiv$	$\mathsf{loop}(b, e, b, b)$	loop
	$\mid$	e			expression
	$\mid$			break	break
	$\mid$			continue	continue
	$\mid$			leave	leave
	$\mid$	q	$\equiv$	$\mathsf{query}(\psi)$	query

e	$::=$				**Expressions:**
	$\mid$	c_n	$\equiv$	$\mathsf{call}(i, e*)$	internal call
	$\mid$	c_x	$\equiv$	$\mathsf{xcall}(\varrho, e*)$	external call
	$\mid$	c_d	$\equiv$	$\mathsf{dcall}(\varrho, e*)$	delegate call
	$\mid$	$u\langle t\rangle$	$\equiv$	$[e*]$	list, typed
	$\mid$	$w\langle t, t\rangle$	$\equiv$	(e, e)	tuple, typed
	$\mid$	v			literal
	$\mid$	t			type
	$\mid$	ϱ			access path
ϱ	$::=$				**Access Paths:**
	$\mid$	i			identifier
	$\mid$	ϱ_d	$\equiv$	$\varrho.i$	access by field
	$\mid$	ϱ_k	$\equiv$	$\varrho[e]$	access by key

Fig. 7. SCUTUM's behavioral modeling language $\mathcal{S}_b$. z denotes typed identifier list, which is a shorthand for $u\langle w\langle t, i\rangle\rangle$.



```
comp MDC {

  fun createChallenge (...) external {
    branch (_challenges[challengeId].challengeTime > 0) {
      call(this.clearChallenge, ...);
      require(cleared == true);
    } else {}
    require( _challenges[challengeId].challengeTime == 0 );
    ...
    _challenges[challengeId] = ChallengeInfo(...);
    emit(ChallengeInfoUpdated(challengeId, _challenges[challengeId]));
  }

  ...
}
```

(a) example program in SCUTUM

```
comp MDC {

  %% p := successful(createChallenge);
  %% q := called(createChallenge);
  %% assert(always(until(p, and(q, not(p)))));
  fun createChallenge (...) external {
    branch (_challenges[challengeId].challengeTime > 0) {
      call(this.clearChallenge, ...);
      require(cleared == true);
    } else {}
    require( _challenges[challengeId].challengeTime == 0 );
    ...
    _challenges[challengeId] = ChallengeInfo(...);
    emit(ChallengeInfoUpdated(challengeId, _challenges[challengeId]));
  }

  ...
}
```

(b) example program in SCUTUM (with query and annotation)

Fig. 8. An example program written in SCUTUM's languages, where (a) describes the behavior of a component using $\mathcal{S}_b$, and (b) is then enhanced with verification queries using $\mathcal{S}_q$ for composing annotations.

component. A special case is dcall, which invokes an external function without switching of execution context; this corresponds to the *delegate call* mechanism in blockchain programming languages, as well as library call for traditional off-chain components.

Example 4.2 (An Example Program in $\mathcal{S}_b$). Figure 8(a) shows the `createChallenge` function from the motivating example (shwon in Figure 4) written in $\mathcal{S}_b$. The function first reads in a storage variable `_challenges[challengeId].challengeTime` using $\mathcal{S}_b$'s universal access path, whose value is used to trigger an optional branch. The external call in line 8 of the original code in Solidity is converted to a xcall in $\mathcal{S}_b$, whose returned results are then stored in program stack ε. Eventually, the snippet calls the emit construct to write to the event pool ω before the function returns.

Symbolic evaluation rules for $\mathcal{S}_b$. SCUTUM symbolically interprets programs written in $\mathcal{S}_b$ that transits from one state to another. Figure 9 shows a representative subset of the symbolic evaluation rules for $\mathcal{S}_b$, where a rule written in the following form:

$$\langle x, \Gamma, \delta, \pi\rangle \rightsquigarrow \langle y, \Gamma', \delta', \pi'\rangle$$

denotes a successful transition of program state. That is, execution of the form x which results in the return form y. The four-tuple $\langle p, \Gamma, \delta, \pi\rangle$ describes a symbolic state during execution, where:

- p is a program counter that points to the immediate next language construct or the evaluated result. We place $\varnothing$ for empty result.

$$\frac{\langle b, \Gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi' \rangle}{\langle \mathsf{comp}(\_, \_*, b), \Gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi' \rangle} \ (\textsc{Comp}) \qquad \frac{\begin{array}{c}\langle s_0, \Gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \Gamma_0, \delta_0, \pi_0 \rangle \quad \dots \\ \langle s_n, \Gamma_{n-1}, \delta_{n-1}, \pi_{n-1} \rangle \rightsquigarrow \langle \varnothing, \Gamma_n, \delta_n, \pi_n \rangle\end{array}}{\langle \mathsf{block}(s_0, \dots, s_n), \Gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \Gamma_n, \delta_n, \pi_n \rangle} \ (\textsc{Blk})$$

$$\frac{\begin{array}{c}\langle e, \Gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \Gamma_0, \delta_0, \pi_0 \rangle \qquad \langle \varrho, \Gamma_0, \delta_0, \pi_0 \rangle \rightsquigarrow \langle \varrho, \Gamma', \delta_0, \pi' \rangle \\ \delta' = \delta_0 \uplus \{\varrho : \{\|\pi'\|v\}\}\end{array}}{\langle \mathsf{assign}(\varrho, e), \Gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi' \rangle} \ (\textsc{Asn}) \qquad \frac{v = \{\|\pi'\|q \in \delta[i] \mid \pi' \rightarrow \pi\}}{\langle i, \Gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \Gamma, \delta, \pi \rangle} \ (\textsc{Id})$$

$$\frac{\begin{array}{c} g_0 \equiv \mathsf{guarded}(e_0, b_0) \qquad \dots \qquad g_n \equiv \mathsf{guarded}(e_n, b_n) \\ \pi_0 = \pi \qquad \langle e_0, \Gamma, \delta, \pi_0 \rangle \rightsquigarrow \langle v_0, \Gamma_0, \delta_0, \pi_0 \rangle \qquad \langle b_0, \Gamma_0, \delta_0, \pi \wedge v_0 \rangle \rightsquigarrow \langle \varnothing, \Gamma'_0, \delta'_0, \pi_0 \wedge v_0 \rangle \\ \pi_1 = \pi_0 \wedge \neg v_0 \qquad \langle e_1, \Gamma, \delta, \pi_1 \rangle \rightsquigarrow \langle v_1, \Gamma_1, \delta_1, \pi_1 \rangle \qquad \langle b_1, \Gamma_1, \delta_1, \pi_1 \wedge v_1 \rangle \rightsquigarrow \langle \varnothing, \Gamma'_1, \delta'_1, \pi_1 \wedge v_1 \rangle \\ \dots \\ \pi_n = \pi_{n-1} \wedge \neg v_{n-1} \qquad \langle e_n, \Gamma, \delta, \pi_n \rangle \rightsquigarrow \langle v_n, \Gamma_n, \delta_n, \pi_n \rangle \qquad \langle b_n, \Gamma_n, \delta_n, \pi_n \wedge v_n \rangle \rightsquigarrow \langle \varnothing, \Gamma'_n, \delta'_n, \pi_n \wedge v_n \rangle \\ \pi_b = \pi_n \wedge \neg v_n \qquad \langle b, \Gamma, \delta, \pi_b \rangle \rightsquigarrow \langle \varnothing, \Gamma'_b, \delta'_b, \pi_b \rangle \qquad \Gamma' = \Gamma \cup \Gamma'_0 \cup \dots \cup \Gamma'_n \cup \Gamma'_b \qquad \delta' = \delta \uplus \delta'_0 \uplus \dots \uplus \delta'_n \uplus \delta'_b \end{array}}{\langle \mathsf{branch}(b, g_0, \dots, g_n), \Gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi \rangle} \ (\textsc{Bch})$$

$$\frac{\begin{array}{c} p \equiv \mathsf{snippet}(i, i_0, \dots, i_n, b) \\ \langle e_0, \Gamma, \delta, \pi \rangle \rightsquigarrow \langle v_0, \Gamma_0, \delta_0, \pi_0 \rangle \qquad \dots \qquad \langle e_n, \Gamma_{n-1}, \delta_{n-1}, \pi_{n-1} \rangle \rightsquigarrow \langle v_n, \Gamma_n, \delta_n, \pi_n \rangle \\ \delta_i = \delta \uplus \{i_0 : \{\|\pi_0\|v_0\}, \dots, i_n : \{\|\pi_n\|v_n\}\} \qquad \langle p, \Gamma_n, \delta_i, \pi_n \rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi' \rangle \end{array}}{\langle \mathsf{expand}(i, e_0, \dots, e_n), \Gamma, \delta, \pi \rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi' \rangle} \ (\textsc{Exp}) \qquad \frac{\begin{array}{c} \langle \varrho, \Gamma, \delta, \pi \rangle \rightsquigarrow \langle x, \Gamma_0, \delta_0, \pi_0 \rangle \\ \langle e, \Gamma_0, \delta_0, \pi_0 \rangle \rightsquigarrow \langle y, \Gamma_1, \delta', \pi' \rangle \\ v = \{\|\pi''\|q \in x[y] \mid \pi'' \rightarrow \pi'\} \\ \Gamma' = \Gamma_1 \cup \{y < |x|\} \end{array}}{\langle \varrho[e], \Gamma, \delta, \pi \rangle \rightsquigarrow \langle v, \Gamma', \delta', \pi' \rangle} \ (\textsc{Lac})$$

Fig. 9. A representative subset of the symbolic evaluation rules for the behavioral modeling langauge $\mathcal{S}_b$.

- Γ is the program state that provides access to the stack, memory, storage and event pool during execution. In Scutum's, a program has access to δ via different access paths (e.g., variable, snippet or identifier, index of memory or storage location, etc.) as described by the AccessPaths section in Figure 7.
- δ is the *assertion store* that tracks verification conditions generated during the execution. Such conditions can be explicitly produced and pushed to δ via verification constructs (which we elaborate in Section 4.2), or implicitly added via some operations that implies some facts. For example, access to a list $l[c]$ pushes an implicit condition $c < |l|$ to δ indicating the index c must be smaller than the size of the list l.
- π keeps track of the current path condition, which is a boolean value that must evaluate to true in order to reach the current program state. That being said, if a path condition evaluates to false after execution of the form x, then the current program state is *unreachable* and should not be considered anymore; this can also be written as $\langle x, \Gamma, \delta, \pi \rangle \rightsquigarrow \bot$.

As shown by Figure 9, Scutum's symbolic evaluation starts by identifying the entrance specified by the user with the (Comp) rule, which directs the program counter to each of the statements within the attaching block via the (Bloc) rule. The (Expr) rule unrolls the content of a comp in the current execution context. Rule (Brch) formulates the state transition when branches are met: each if condition is appended to the current path condition when entering its corresponding block, and negated when entering the next branch.

Here, we introduce the notation $\|\pi\|v$ to denote a value v that is obtained under a certain path condition π (aka, a *guarded* value). The (Brch) rule ends with merging of the newly obtained guarded values in program stores that correspond to different branches using the merging operator $\uplus$:

$$\begin{aligned} \Gamma_0 \uplus \Gamma_1 = {} & \{\varrho : \Gamma_0[\varrho] \mid \varrho \in \Delta(\Gamma_0, \Gamma_1)\} \cup \{\varrho : \Gamma_1[\varrho] \mid \varrho \in \Delta(\Gamma_1, \Gamma_0)\} \cup \\ & \{\varrho : \Gamma_0[\varrho] \cup \Gamma_1[\varrho] \mid \varrho \in \mathsf{dom}(\Gamma_0) \wedge \varrho \in \mathsf{dom}(\Gamma_1)\}, \\ \text{where } \Delta(\Gamma_0, \Gamma_1) = {} & \mathsf{dom}(\Gamma_0) \backslash \mathsf{dom}(\Gamma_1). \end{aligned}$$

Similar to the (Bch) rule, in the (Asn) rule, any value being assigned to a location also carries the current path condition as its guard and should be merged with the program store δ using the *uplus* operator.

$$
\begin{array}{rcll}
\psi & ::= & & \textbf{Queries:} \\
& \mid & \psi \wedge \psi & \text{conjunction} \\
& \mid & \neg\psi & \text{negation} \\
& \mid & \psi \,\mathcal{U}\, \psi & \text{until} \\
& \mid & \circ\psi & \text{next} \\
& \mid & \square\psi & \text{always} \\
& \mid & \forall e\,.\,\psi & \text{universal quantifier} \\
& \mid & p & \text{predicate} \\
& \mid & e & \text{expression}
\end{array}
\qquad
\begin{array}{rcll}
p & ::= & & \textbf{Predicates:} \\
& \mid & \mathsf{reverted}(\varrho, e) & \text{called before and reverted} \\
e & ::= & & \textbf{Expressions} \text{ (from } \mathcal{S}_b\text{)} \\
\varrho & ::= & & \textbf{Access Paths} \text{ (from } \mathcal{S}_b\text{)}
\end{array}
$$

Fig. 10. Scutum's query language $\mathcal{S}_q$. We formalize a minimal version for better presentation; note that the full set of operators can be expressed and expanded using this version. For example, $\psi \vee \psi ::= \neg(\psi \wedge \psi)$, etc.

$$
\frac{\langle e, \Gamma, \delta, \pi\rangle \rightsquigarrow \langle c, \Gamma', \delta', \pi'\rangle \quad c \equiv \mathsf{false} \vee \neg\tau_{\mathsf{bool}}(c)}{\langle \mathsf{assume}(e), \Gamma, \delta, \pi\rangle \rightsquigarrow \bot}\ (\textsc{Asm1})
\qquad
\frac{\langle e, \Gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \Gamma', \delta', \pi'\rangle}{\langle \mathsf{assume}(e), \Gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi' \wedge v\rangle}\ (\textsc{Asm2})
\qquad
\frac{\langle e, \Gamma, \delta, \pi\rangle \rightsquigarrow \langle c, \Gamma', \delta', \pi'\rangle \quad c \equiv \mathsf{false} \vee \neg\tau_{\mathsf{bool}}(c)}{\langle \mathsf{assert}(e), \Gamma, \delta, \pi\rangle \rightsquigarrow \bot}\ (\textsc{Ast1})
$$

$$
\frac{\langle e, \Gamma, \delta, \pi\rangle \rightsquigarrow \langle v, \Gamma', \delta', \pi'\rangle}{\langle \mathsf{assert}(e), \Gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \Gamma' \cup \{\pi' \rightarrow v\}, \delta', \pi'\rangle}\ (\textsc{Ast2})
\qquad
\frac{\langle s, \Gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi'\rangle}{\langle \mathsf{annotate}(s), \Gamma, \delta, \pi\rangle \rightsquigarrow \langle \varnothing, \Gamma', \delta', \pi'\rangle}\ (\textsc{Ant})
$$

Fig. 11. Symbolic evaluation rules for the query language $\mathcal{S}_q$.

4.2 Writing Queries for Safety Checks

Scutum provides a set of language extensions $\mathcal{S}_q$ for verification. As shown in Figure 10, $\mathcal{S}_q$ inherits the expressions and access paths from the $\mathcal{S}_b$ language, and incorporates two new categories of forms for constructing verification queries and specification:

- **Queries** We show a minimum subset of the quantifiers (e.g., $\forall$), logical operators (e.g., $\neg$, $\wedge$), and temporal operators (e.g., $\mathcal{U}$, $\circ$, $\square$). Note that the full set of operators can be expressed and expanded using the existing ones; for example, $\psi \vee \psi ::= \neg(\psi \wedge \psi)$.
- **Predicates** In addition to regular predicates that can be defined by arithmetic operations, Scutum introduces an additional set of special predicates related to different aspects of program execution in systems like cross-rollup bridges. For example, the reverted predicate models a fact over the historical state of whether a function has been called and successfully returned. These predicates provide extra power and make it easier to refer to some special states and facts that could not be easily expressed in normal verification interface.

Figure 11 shows the symbolic evaluation rules for the verification extension $\mathcal{S}_q$. Verification related rules, namely (Asm1), (Asm2), (Ast1) and (Ast2), directly push verification conditions into assertion store δ or path condition π. Such verification conditions need to be implied when a location from the program state Γ is accessed. For example, the (Id) rule and (Lac) rule can only access the corresponding values that are guarded by its current path condition.

Example 4.3 (An Example Program in $\mathcal{S}_q$). Figure 8(b) shows the `createChallenge` function from the motivating example (shown in Figure 4) written with extra verification annotation $\mathcal{S}_q$.

4.3 A Vanilla Solution for Safety Verification

Temporal property verification seeks to ensure that a system behaves correctly over time by verifying that its sequences of states satisfy specific conditions. Model checking, a classical technique in formal verification, has been extensively employed for this purpose. Model checking systematically explores the state space of a system to determine whether a given temporal logic specification holds.

Algorithm 1 Verification by Goal-Driven Reduction

```
1: procedure CHECK($\mathbb{B}$, $\phi$, $\Gamma_0$)
2:     input: bridge $\mathbb{B}$, specification $\phi$, initial state $\Gamma_0$
3:     output: a counterexample or safe $\bot$
4:     $\phi' \leftarrow \text{reduce}(\phi; \mathbb{B})$                    ▷ reduces temporal property to safety check
5:     $\Lambda \leftarrow \text{GraphGen}(\phi'; \mathbb{B})$                ▷ generates location dependence graph
6:     while $S \leftarrow \text{Enumerate}(\Lambda)$ do       ▷ generates sketch of execution sequence by graph reachability
7:         $r \leftarrow \text{SAT}(\text{SEval}(\Gamma_0, S; \mathbb{B}) \wedge \neg\phi')$    ▷ symbolically evaluates sketch and check for violation
8:         if $r$ then return model($r$)      ▷ returns the concrete execution sequence that violates $\phi'$
9:     return $\bot$                                       ▷ no violation is found; returns safe
```

Given a system model described in Section 4.1, a classical model checking algorithm exhaustively explores the system model's state space to verify whether the specified temporal properties hold. If the properties are violated, the model checker provides a counterexample, which is a sequence of states demonstrating the violation.

While model checking provides a systematic and exhaustive approach to temporal property verification, several limitations arise when applied to cross-rollup bridges:

- **Scalability** Model checking algorithms often struggle with scalability due to the state explosion problem. Cross-rollup bridges, which integrate multiple entities and protocols, introduce a vast state space that can become computationally infeasible for classical model checking.
- **Complex Interactions and Temporal Dependencies** Cross-rollup bridges rely on complex interactions between on-chain and off-chain entities, often requiring reasoning about temporal sequences of events. Traditional model checkers may struggle to capture and verify these interactions within a single framework.
- **Over-Approximation and False Positives** Model checking's exhaustive exploration can lead to over-approximation, especially when combined with symbolic abstractions. This can result in a high number of false positives, complicating the verification process.

5 Verification by Goal-Driven Reduction

In this section, we elaborate SCUTUM's verification algorithm. We start with an overview of the verification algorithm in Section 5.1. As vanilla temporal verification usually enumerates program execution sequences over an infinite search space, to address this issue, we introduce a goal-driven approach to reduce it to a smaller subset. Following this, Section 5.2 first introduces a new graph representation that *quantifies* the search space more efficiently, which then allows a constrained enumeration of the search space that only considers execution sequences that can likely violate the provided specification. Section 5.3 elaborates such enumeration algorithm by formalizing the problem into a graph reachability problem.

5.1 Algorithm Overview

Algorithm 1 shows our top-level procedure for verification. Given a bridge $\mathbb{B}$ and an initial program state Γ_0, SCUTUM starts by converting the liveness property from the temporal specification ϕ into a safety check ϕ' (line 4), i.e., a non-temporal specification that should always hold during the entire course of execution, through a standard reduction procedure. Based on this new specification ϕ', SCUTUM infers a special graph structure that we refer to as *location dependence graph* Λ, which encodes a goal-driven and reduced search space of program execution sequence that could potentially violate the specification ϕ' (line 5). SCUTUM then keeps enumerating such candidate sub-graphs S

(line 6), and checks for the aforementioned violation (line 7) until a model (i.e., counterexample) is found and returned (line 8); otherwise, when the enumeration exhausts, it terminates asserting that the specification ϕ is safe on the given bridge $\mathbb{B}$ and initial program state Γ_0 (line 9).

5.2 Location Dependence Graph

To prune irrelevant search space during verification, we propose a new type of graph that quantify the search space. To start with, we elaborate the notion of *location*, which is the building block of the graph.

Locations. A specification for verification is constructed with predicates and expressions using Scutum's $\mathcal{S}_q$ language. To evaluate the truth value of a given specification ϕ, Scutum's evaluation rules will eventually retrieve data stored in program states and apply corresponding operations over them. We thus define those referred value accessed from program state, which the truthfulness of specification depends on.

Definition 5.1 (Location). Given a specification ϕ, we say an access path is a *location* (denoted by λ), if ϕ's truthfulness depends on it. Such dependence could be data dependence, control dependence etc. We say an access path is a location λ with regard to specification ϕ, or location λ *reflects* specification ϕ.

Specifically, if a specification ϕ depends on a location λ_1, which itself further depends on another access path λ_2, then λ_2 is also a location with regard to ϕ. In this case, λ_1 is referred to as a *goal location* as it's the most immediate dependent of specification ϕ.

Example 5.2 (Example Locations). From the motivating example shown in Figure 4 with the given specification ϕ in Equation 1, we can identify several locations with regard to ϕ. For example: `_challenges[challengeId].challengeTime` and `cleared` from function `createChallenge`, and `abortTime` from function `clearChallenge` etc.

Given a specification, We show in Figure 12 a representative set of inference rules for recursively identifying locations on a bridge system, where Δ is an environment that stores identified locations. Therefore, the judgment:

$$\phi, \Delta \Vdash e$$

denotes that expression e is a location with regard to the given specification ϕ and existing locations stored in the environment Δ. Specially, when e is an access path, the judgment becomes:

$$\phi, L \Vdash e,$$

where L is the set of locations extracted for specification ϕ.

To quantify the search space of execution sequence that could potentially violate the user-provided specification, we introduce a new type of graph called *location dependence graph*

Graph construction. We then define the location dependence graph.

Definition 5.3 (Location Dependence Graph). Given a specification ϕ, a *location dependence graph* G_ϕ is a hypergraph defined as $G_\phi = (X, E)$, where X corresponds to a set of locations L, and E corresponds to a set of hyperedges which is defined over $F \times L \times L$ where F corresponds to a set of functions available in the system.

Example 5.4 (Example Location Dependence Graph). Figure 13 shows the location dependence graph constructed from the motivating example and specification. The hyperedge that connects `abortTime` to `cleared` is labeled with `clearChallenge`, since `abortTime` controls the valuation of `cleared` via the function `clearChallenge`. That is, if the assertion of `abortTime` in line 27 fails, the value of `clear` in line 8 may change. Thus, the location dependence graph captures such dependence.

$$\frac{\phi, \Delta \Vdash \psi_0 \wedge \psi_1}{\phi, \Delta \Vdash \psi_0 \qquad \phi, \Delta \Vdash \psi_1}\ (\textsc{And}) \qquad \frac{\phi, \Delta \Vdash \neg\psi}{\phi, \Delta \Vdash \psi}\ (\textsc{Negation}) \qquad \frac{\phi, \Delta \Vdash \psi_0 \mathcal{U} \psi_1}{\phi, \Delta \Vdash \psi_0 \qquad \phi, \Delta \Vdash \psi_1}\ (\textsc{Until}) \qquad \frac{\phi, \Delta \Vdash \circ\psi}{\phi, \Delta \Vdash \psi}\ (\textsc{Next})$$

$$\frac{\phi, \Delta \Vdash \Box\psi}{\phi, \Delta \Vdash \psi}\ (\textsc{Always}) \qquad \frac{\phi, \Delta \Vdash \forall e\ .\ \psi}{\phi, \Delta \Vdash \psi \qquad \phi, \Delta \Vdash e}\ (\textsc{Univ}) \qquad \frac{\phi, \Delta \Vdash [e_0, ..., e_n]}{\phi, \Delta \Vdash e_0 \quad ... \quad \phi, \Delta \Vdash e_n}\ (\textsc{List}) \qquad \frac{\phi, \Delta \Vdash (e_0, e_1)}{\phi, \Delta \Vdash e_0 \qquad \phi, \Delta \Vdash e_1}\ (\textsc{Tuple})$$

$$\frac{\phi, \Delta \Vdash \mathsf{reverted}(\varrho)}{\phi, \Delta \Vdash e \quad e \in \mathsf{requires}(\varrho; \mathbb{B}) \cup \mathsf{conditions}(\varrho; \mathbb{B})}\ (\textsc{Reverted}) \qquad \frac{\phi, \Delta \Vdash \oplus(\varrho, e_0, ..., e_n) \qquad \oplus \in \{\mathsf{call}, \mathsf{xcall}, \mathsf{dcall}\}}{\phi, \Delta \Vdash e_0 \quad ... \quad \phi, \Delta \Vdash e_n}\ (\textsc{Call}) \qquad \frac{\phi, \Delta \Vdash i}{\phi, L \Vdash i}\ (\textsc{Id})$$

$$\frac{\phi, \Delta \Vdash \varrho.i}{\phi, \Delta \Vdash \varrho \qquad \phi, \Delta \Vdash i \qquad L \Vdash \varrho.i}\ (\textsc{Field}) \qquad \frac{\phi, \Delta \Vdash \varrho[e]}{\phi, \Delta \Vdash \varrho \qquad \phi, \Delta \Vdash e \qquad L \Vdash \varrho[e]}\ (\textsc{Key}) \qquad \frac{\phi, \Delta \Vdash \mathsf{call}(\mathsf{require}, e)}{\phi, \Delta \Vdash e}\ (\textsc{Require})$$

Fig. 12. Rules for goal-driven location inference.

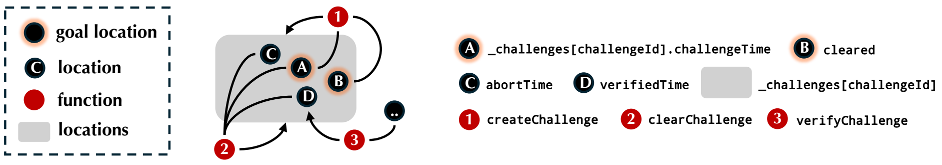


Fig. 13. Location dependence graph for the motivating example.

In addition, using the inference rules in Figure 12, we know that `cleared` is a goal location, but `abortTime` isn't.

5.3 Enumeration of Execution Sequence

The problem is reduced to a graph reachability problem over the location dependence graph. As shown in Algorithm 2, the algorithm starts by extracting a set of goal locations L from the location dependence graph Λ using the function $\mathsf{goal}(\Lambda)$. These goal locations represent the critical points in the graph that are relevant to the specification ϕ—for example, states where a security property might be violated. It then encodes the search for paths to these goal locations as Integer Linear Programming (ILP) constraints using the function $\textsc{Encode}(\Lambda, L)$. This encoding transforms the graph reachability problem into mathematical formulas, where the variables and constraints represent the presence or absence of edges and nodes in potential execution paths. A blocking constraint set κ is initialized to $\top$, indicating that no execution sketches are currently blocked.

The algorithm enters the main loop (line 7) where it attempts to find a minimal execution sketch σ that satisfies both the encoded constraints ψ and the current blocking constraints κ. This is done using the function $\mathsf{minimize}(\psi \wedge \kappa)$, which seeks the simplest (e.g., shortest) path that reaches a goal location without violating any constraints. If such a path σ is found, it represents a feasible execution sketch that could potentially lead to a violation of the specification ϕ. The algorithm immediately returns this execution sketch for further analysis. To ensure the enumeration of all possible execution sketches, the algorithm updates the blocking constraints κ by conjoining it with $\mathsf{block}(\sigma)$. This effectively blocks the current execution sketch σ from being considered in future iterations, forcing the algorithm to explore alternative paths in the next loop.

The loop continues until no new execution sketches can be found—that is, when $\mathsf{minimize}(\psi \wedge \kappa)$ returns no solution. At this point, the algorithm concludes that all feasible execution sketches have been enumerated and returns $\perp$ to indicate exhaustion.

5.4 Solving Reachability via ILP

Algorithm 2 Enumeration of Execution Sketch by Graph Reachability Analysis

```
1: procedure ENUMERATE($\Lambda, \phi$)
2:     input: location dependence graph $\Lambda$, specification $\phi$
3:     output: execution sketch $S$ or exhausted $\bot$
4:     $L \leftarrow \mathsf{goal}(\Lambda)$                    ▷ extracts a set of goal locations
5:     $\psi \leftarrow$ ENCODE($\Lambda, L$)    ▷ encodes search for goal locations as graph reachability ILP constraints
6:     $\kappa = \top$                                ▷ initializes the set of graphs blocked
7:     while $\sigma \leftarrow \mathsf{minimize}(\psi \wedge \kappa)$ do          ▷ solves for minimum reachable graph
8:         return $\sigma$
9:         $\kappa \leftarrow \kappa \wedge \mathsf{block}(\sigma)$               ▷ blocks the current proposed graph
10:    return $\bot$                                  ▷ exhausted; returns $\bot$
```

In this section, we elaborate on the details of our ILP encoding for solving the reachability problem. Given A location-dependence graph $\Lambda = (V, E)$ where 1) V is the set of nodes (locations) and E is the set of directed edges between nodes, 2) A start node $s \in V$, 3) A set of goal nodes $L \subseteq V$. We aim to find a path from s to any node in L.

Variables.

- Edge Variables (x_e): For each edge $e \in E$, define a binary variable $x_e \in \{0,1\}$. $x_e = 1$ if edge e is included in the path; $x_e = 0$ otherwise.
- Node Variables (y_v): For each node $v \in V$, define a binary variable $y_v \in \{0,1\}$. $y_v = 1$ if node v is included in the path; $y_v = 0$ otherwise.
- Goal Node Indicator Variables (z_l): For each goal node $l \in L$, define a binary variable $z_l \in \{0,1\}$. $z_l = 1$ if the path reaches goal node l; $z_l = 0$ otherwise.

Constraints.

- Flow Conservation Constraints: δ^- and δ^+ represent the set of incoming and outgoing edges, respectively.
$$\sum_{e \in \delta^-(v)} x_e = \sum_{e \in \delta^+(v)} x_e \quad \forall v \in V \backslash \{s\} \backslash L.$$
- Start Node Constraint: Ensure that exactly one edge leaves the start node s:
$$\sum_{e \in \delta^+(s)} x_e = 1.$$
- Goal Node Constraints: for each goal node $l \in L$:
$$\sum_{e \in \delta^-(l)} x_e = z_l \quad \forall l \in L.$$
- Goal Node Selection Constraint: Ensure that the path reaches exactly one goal node.
$$\sum_{l \in L} z_l = 1.$$
- Edge-Node Relationship Constraints (if using node variables): ensures that if an edge is used, both its source and target nodes are included in the path.
$$x_e \leq y_u \quad \forall e = (u \rightarrow v) \in E,$$
$$x_e \leq y_v \quad \forall e = (u \rightarrow v) \in E.$$

The *Objective Function* is to minimize the total length of the path:
$$\mathsf{minimize} \sum_{e \in E} x_e.$$

6 Related Work

In this section, we survey the closest related works with respect to our proposed techniques.

Smart contract vulnerability analysis. Existing methods for detecting and analyzing vulnerabilities in smart contracts typically fall into two categories: static analysis[2, 16, 17] and dynamic analysis[10, 19, 26]. Static analysis tools, such as symbolic execution, identify common vulnerabilities (as outlined in) without needing an in-depth understanding of a DeFi protocol. For instance, Securify[28] examines smart contract bytecode to identify pre-defined vulnerability patterns through control flow graph analysis. Slither[14], which is also integrated into Scutum, is one of the most stable and actively maintained static analysis frameworks for smart contracts. Prominent symbolic execution tools include Manticore[21], Mythril[11], Solar[15], and Halmos [1] (the current state-of-the-art). As shown in our evaluation, symbolic execution alone, without effective sketch generation and domain-specific compilation, struggles to address complex logical bugs in DeFi protocols.

On the other hand, most dynamic and hybrid analysis tools are limited to a single contract [5, 19, 22, 31]. Even tools that support cross-contract fuzzing, such as ItyFuzz [26], face challenges in generating effective results for DeFi attack synthesis without understanding the broader protocol logic.

DeFi Security. The primary challenge in DeFi security stems from the complexity and broader scope beyond individual smart contracts, involving intricate semantics and logic. While Zhou et al.[32] provide a comprehensive overview of DeFi attacks, most existing research focuses on identifying patterns from past attack instances and developing detection tools based on pattern matching. For example, DeFiRanger[30] abstracts low-level smart contract semantics into high-level patterns for detection, while FlashSyn[9] uses numerical approximations to identify patterns in attack transaction sequences, enabling real-time detection of suspicious activity. UnifairTrade[7] identifies vulnerabilities in swap pair implementations, and DeFiTainter [20] applies taint analysis by extracting sources and sinks from standard contract APIs.

However, the effectiveness and scalability of these approaches are limited by their reliance on predefined patterns, allowing them to detect only certain types of vulnerabilities, such as price manipulation through token swaps. Some newer tools have extended this methodology to other vulnerabilities. For instance, DeFiCrisis[18] explores strategies for exploiting DeFi governance by manipulating funding, and TokenScope[8] detects inconsistencies and phishing activities in token applications.

Temporal verification. Numerous studies have explored the analysis of temporal properties. For example, [25] presents an approach for testing violations of past Linear Temporal Logic (LTL) formulas. Similarly, [3] focuses on verifying LTL formulas within UML models. T2 [6] is a system designed to verify temporal properties over LLVM, specifically supporting programs with linear integer arithmetic. E-HSF [4] extends this by verifying existential Computation Tree Logic (CTL) formulas, representing them as existentially quantified Horn-like clauses and solving them using a counterexample-guided approach. Typestates [27] allow for the expression of correct usage patterns in class operations or protocols, and several of the properties evaluated in this work can be expressed as typestates. While those prior works can conceptually used to enhance Scutum, none of them targets cross-rollup bridges.

The approach most closely aligned with Scutum is VerX [24], a verifier that can automatically prove temporal safety properties of Ethereum smart contracts. Similar to Scutum, VerX reduces temporal safety verification to reachability checking, an efficient symbolic execution engine used to compute precise symbolic states within a transaction. Scutum is different in multiple aspects: first, Scutum extends both the specification and verification to reasoning about cross-rollup bridges, a crucial infrastructure in blockchains; Second, Scutum reduces the expensive temporal checking over liveness properties to static analyzing the safety properties of a novel graph representation of the original system, leading to efficient verification time without compromising much on the precision.

7 Conclusion

We have introduced a scalable verifier designed to systematically assess the security of cross-rollup bridges—vital infrastructure for seamless asset transfers across Layer 2 (L2) rollups and between L2 and Layer 1 (L1) chains. Our approach overcomes the limitations of traditional security analysis methods by employing a comprehensive multi-model framework that captures both individual behaviors and complex interactions using temporal properties. By approximating temporal safety verification through reachability analysis of a graph representation, we effectively leverage advanced program analysis techniques. The incorporation of a conflict-driven refinement loop further enhances precision by eliminating false positives.

Our evaluation on mainstream cross-rollup bridges, including Orbiter Finance, uncovered multiple zero-day vulnerabilities, demonstrating the practical utility of our method. With favorable runtime performance enabling efficient analysis suitable for real-time or near-real-time applications, our verifier addresses the urgent need for robust security assessments of cross-rollup bridges.

References

[1] a16z. 2023. Halmos: A symbolic testing tool for EVM smart contracts. https://github.com/a16z/halmos.

[2] Elvira Albert, Shelly Grossman, Noam Rinetzky, Clara Rodríguez-Núñez, Albert Rubio, and Mooly Sagiv. 2020. Taming Callbacks for Smart Contract Modularity. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 209 (nov 2020), 30 pages. https://doi.org/10.1145/3428277

[3] Luciano Baresi, Mohammad Mehdi Pourhashem Kallehbasti, and Matteo Rossi. 2015. Efficient Scalable Verification of LTL Specifications. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 711–721.

[4] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. 2013. Solving Existentially Quantified Horn Clauses. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 869–882. https://doi.org/10.1007/978-3-642-39799-8_61

[5] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, and Christopher Kruegel. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*.

[6] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. 2016. T2: Temporal Property Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9636)*, Marsha Chechik and Jean-François Raskin (Eds.). Springer, 387–393.

[7] Jiaqi Chen, Yibo Wang, Yuxuan Zhou, Wanning Ding, Yuzhe Tang, XiaoFeng Wang, and Kai Li. 2023. Understanding the Security Risks of Decentralized Exchanges by Uncovering Unfair Trades in the Wild. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*. 332–351. https://doi.org/10.1109/EuroSP57164.2023.00028

[8] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1503–1520. https://doi.org/10.1145/3319535.3345664

[9] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. 2022. FlashSyn: Flash Loan Attack Synthesis via Counter Example Driven Approximation. arXiv:2206.10708 [cs.PL]

[10] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *2021 36th IEEE/ACM International Conference*

on Automated Software Engineering (ASE). 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

[11] ConsenSys. 2020. Mythril: Security Analysis Tool for Ethereum Smart Contracts. https://github.com/ConsenSys/mythril.

[12] Byron Cook, Eric Koskinen, and Moshe Vardi. 2011. Temporal Property Verification as a Program Analysis Task. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 333–348.

[13] Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. 2013. Predicate Abstraction for Relaxed Memory Models. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 84–104.

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

[15] Yu Feng, Emina Torlak, and Rastislav Bodik. 2021. Summary-Based Symbolic Evaluation for Smart Contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1141–1152. https://doi.org/10.1145/3324884.3416646

[16] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: Surviving out-of-Gas Conditions in Ethereum Smart Contracts. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 116 (oct 2018), 27 pages. https://doi.org/10.1145/3276486

[17] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts. *Proc. ACM Program. Lang.* 2, POPL, Article 48 (dec 2017), 28 pages. https://doi.org/10.1145/3158136

[18] Lewis Gudgeon, Daniel Perez, Dominik Harz, Benjamin Livshits, and Arthur Gervais. 2020. The Decentralized Financial Crisis. arXiv:2002.08099 [cs.CR]

[19] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/3238147.3238177

[20] Queping Kong, Jiachi Chen, Yanlin Wang, Zigui Jiang, and Zibin Zheng. 2023. DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1144–1156. https://doi.org/10.1145/3597926.3598124

[21] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[22] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. SFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. https://doi.org/10.1145/3377811.3380334

[23] Orbiter Finance. [n. d.]. Orbiter Finance. https://www.orbiter.finance/. Accessed: 2024-9-10.

[24] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1661–1677.

[25] Grigore Rosu, Feng Chen, and Thomas Ball. 2008. Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008. Selected Papers (Lecture Notes in Computer Science, Vol. 5289)*, Martin Leucker (Ed.). Springer, 51–68.

[26] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 322–333. https://doi.org/10.1145/3597926.3598059

[27] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. https://doi.org/10.1109/TSE.1986.6312929

[28] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. https://doi.org/10.1145/3243734.3243780

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

[30] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. 2021. DeFiRanger: Detecting Price Manipulation Attacks on DeFi Applications. arXiv:2104.15068 [cs.CR]

[31] Valentin Wüstholz and Maria Christakis. 2020. Harvey: A Greybox Fuzzer for Smart Contracts. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1398–1409. https://doi.org/10.1145/3368089.3417064

[32] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. 2023. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2444–2461.