$2B Lessons: BRIGADE as a Defense Against Real-World DeFi Bridge Exploits

Pascal Winkler[1]✉[0009−0002−8324−4993], Jens-Rene Giesen[1][0009−0004−0685−6237], Oussama Draissi[1][0009−0005−6065−8087], Federico Badaloni[2][0009−0007−7713−740X], Sebastian Holler[2][0009−0005−2074−9187], Clara Schneidewind[2][0009−0000−5471−8454], and Lucas Davi[1][0000−0002−7322−2777]

[1] paluno - the Ruhr Institute for Software Technology, University of Duisburg-Essen, Essen, Germany
{pascal.winkler,jens-rene.giesen,oussama.draissi,lucas.davi}@uni-due.de
[2] Max Planck Institute for Security and Privacy, Bochum, Germany
{federico.badaloni,sebastian.holler,clara.schneidewind}@mpi-sp.org

Abstract. Cross-chain bridges enable decentralized finance (DeFi) applications. However, ensuring their security remains a significant open challenge. Several recent attacks caused losses exceeding 2 billion USD. In this paper, we present the design and implementation of BRIGADE, which is the first framework to prevent token losses in cross-chain transactions and their router contracts. BRIGADE employs universal policies and event detection systems to enable secure and fast token transfers in cross-chain bridges. We formally validate the general applicability of these policies using the Tamarin prover and conduct an extensive evaluation of twelve real-world cross-chain bridge attacks on Ethereum, Solana, and the Binance Smart Chain. Our results demonstrate the high efficiency and effectiveness of our novel mitigation techniques.

Keywords: Distributed Systems · Cross-Chain Applications · Blockchain · Runtime Monitoring · Security Assessment · Policy Enforcement.

1 Introduction

The rise of blockchains has fostered a vibrant ecosystem commonly referred to as decentralized finance (DeFi). This ecosystem, however, is not a unified whole but rather a collection of isolated environments. Native assets of one platform, such as Ether from Ethereum, cannot seamlessly integrate with applications on another platform, like Solana. To address this challenge, developers rely on *cross-chain bridges*: protocols designed to forge pathways between these otherwise disconnected blockchains [1–5]. The most prevalent bridge design operates on a *lock-and-mint* model [2, 6–10], where an asset is locked on one chain to serve as collateral for a new, functionally compatible token minted on another. The success of this approach has made these protocols essential to the DeFi economy, with over 6 billion USD in assets currently secured within Ethereum bridges [11].

Unsurprisingly, the large concentration of locked value makes bridges a prime target for attackers, with bridges becoming the focal point of security incidents in DeFi [12–17]. A recent study [18] revealed that attacks against cross-chain applications amounted to nearly 70% of all stolen funds in cryptocurrency, while four of the five largest hacks in the cryptocurrency ecosystem targeted cross-chain bridges [19], with losses estimated at nearly 2 billion USD. These incidents highlight critical security gaps arising from their complex design, which splits logic across two domains: on-chain smart contracts that execute transparently on the blockchain and off-chain components that observe, authorize, and forward transactions across networks. The delicate interplay between distinct components opens up a broad attack surface for attackers [2–4, 20].

Existing smart contract analysis approaches are fundamentally limited, as they inspect individual components in isolation [21–30]. This single-chain paradigm is insufficient for cross-chain systems, where security properties emerge from complex interactions across blockchains. The PolyNetwork incident clearly demonstrates this vulnerability [12, 31]. In that attack, a carefully crafted cross-chain message allowed the adversary to seize control of the bookkeeper role, authorized to approve transactions. With this control, the attacker validated their own malicious transactions. The receiving chain, lacking cross-chain context, processed these transactions as valid. This reveals a critical gap in security analysis: the lack of approaches capable of reasoning about the *holistic* behavior of cross-chain systems.

While several approaches address cross-chain attacks [3, 20, 32, 33], they lack the versatility and formal rigor for robust security assessment of real-world bridges. Runtime monitors either rely on rigid patterns that miss slight variations [3], or require comprehensive models of legitimate behavior that are practically impossible to obtain [33]. Complementary efforts using static analysis fail to provide end-to-end security. Approaches like XGuard [20] and SmartAxe [32] cannot capture protocol-level flaws as their analysis is limited to static contract code. They are constrained to EVM-based blockchains, leaving interoperability with non-EVM ecosystems (e.g., Solana) unaddressed. This research gap motivates the need for a holistic and principled approach towards the security analysis of bridges, as advocated by recent studies [34–36].

As a first step in this direction, we introduce Brigade, a system designed to provide just-in-time security assessment for cross-chain bridges by monitoring a set of *universal security policies*. Our approach is powered by Talon, a novel domain-specific language that we specifically designed for expressing these policies. This approach enables Brigade to reason about the security of the bridge as a whole, monitoring global invariants that fall out of the scope of approaches that focus on individual smart contracts. Our policies are formally validated using the Tamarin prover [37] and are designed to capture fundamental invariants of secure bridge operation. Brigade operates as a standalone runtime monitor that observes transactions to identify harmful requests, a design that preserves the bridge's trusted computing base (TCB) and avoids introducing new attack vectors. Leveraging its just-in-time detection capabilities, Brigade enables the

effective enforcement of security policies within bridge designs.

Our policies are grounded in a systematic analysis of major real-world exploits, which we classify into four distinct categories: (1) inconsistent deposit logic, (2) incomplete verification, (3) privilege escalation, and (4) private key compromise. We complement the formal validation of our policies with an extensive empirical evaluation against twelve real-world attacks, showing that Brigade outperforms state-of-the-art approaches in both detection scope and efficiency. The engineering effort behind Brigade and Talon is substantial, with the implementation consisting of approximately 9k lines of Rust code. Our results confirm that Brigade successfully detects these exploits on major blockchains like Ethereum, Solana, and the Binance Smart Chain while introducing no practical performance overhead, as its median detection latency (7ms-110ms) is orders of magnitude lower than transaction inclusion times.

In summary, this paper makes the following contributions:

- **The design and implementation of Brigade**, a novel runtime monitoring system to secure cross-chain bridges without expanding the TCB.
- **A set of universal security policies**, derived from a systematic study of real-world attacks and formally validated using the Tamarin prover.
- **An extensive evaluation** on real-world exploits across Ethereum, Solana, and BSC demonstrates that our approach is both highly effective and efficient.

To foster research on the security of cross-chain bridges, we open-source Brigade[3].

2 Background

Cross-chain bridges connect distinct blockchains, facilitating asset transfers and cross-chain function calls [38, 39]. Interactions between blockchains are not natively supported, as each blockchain is a closed system limited to processing system-specific transactions. Transactions may transfer native assets or trigger *smart contracts* registered on the blockchain. A smart contract execution may invoke other smart contracts (via function calls) within the same blockchain. Smart contracts can also be used to implement custom (non-native) tokens within the context of a blockchain. Most prominently, the ERC20 standard emerged as a unified interface for fungible tokens implemented on the Ethereum blockchain. Bridges enable *interoperability* between blockchains, supporting cross-chain asset transfers (native or non-native) and general cross-chain contract calls. Unlike centralized exchanges, bridges can be fully decentralized and trustless [38], enabling decentralized finance (DeFi) across multiple blockchains.

Figure 1 depicts the architecture of a typical bridge connecting two blockchains, $\mathbb{B}_S$ (source) and $\mathbb{B}_D$ (destination). This architecture consists of smart contracts, called *router contracts*, on each chain, along with an *off-chain component* coordinating interactions. Bridge operators develop and maintain these off-chain components, bridge networks, and router contracts. The primary goal

[3] github.com/uni-due-syssec/Brigade

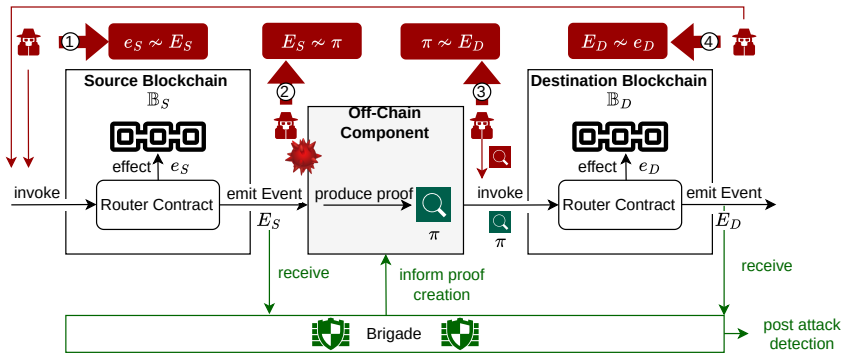

Fig. 1: Prevention of malicious cross-chain transactions with Brigade

of cross-chain interactions is to link an effect e_D on $\mathbb{B}_D$ (e.g., token transfer) to a corresponding effect e_S on $\mathbb{B}_S$ (e.g., token reception). To initiate this process, a user invokes a function on the router contract on $\mathbb{B}_S$, emitting an event E_S. The off-chain component observes this event, generates a proof π, and calls the router contract on $\mathbb{B}_D$. Upon receiving the proof, the router contract verifies it, triggers e_D, and emits a corresponding event E_D. In a cross-currency transfer, a user sends assets to the router contract on $\mathbb{B}_S$. Upon acceptance (e_S), the router contract emits an event (e.g., `Deposit(`e_S`)`). The router contract on $\mathbb{B}_D$ then releases the corresponding assets (e_D) and emits an event (e.g., `Receive(`e_D`)`).

Different mechanisms can be employed to bridge assets. A *lock-and-mint* mechanism [2, 6] involves *locking* native tokens in the source chain's bridge contract and *minting* equivalent tokens on the destination chain. Conversely, the *burn-and-unlock* mechanism [2, 6] is used to return minted tokens by *burning* them on one chain and *unlocking* the original tokens on the other chain. *Atomic swaps* [40] directly exchange assets between chains, ensuring both sides successfully transfer before completing the swap. Additionally, bridges execute *cross-chain functions*, similar to remote procedure calls (RPCs), using specific transactions to interact with contracts on the destination chain.

3 Problem Statement

The complex interplay between different components, such as router contracts and off-chain components, makes it challenging to implement secure cross-chain bridges. Unsurprisingly, existing bridge implementations suffer from severe bugs, of which many have led to massive financial damage [12–17]. The bug types causing those massive financial damages differ significantly from those found in smart contracts that are not part of bridge networks. In the following, we describe the different bug types observed in various exploits.

3.1 Bug Types

As Figure 1 illustrates, we identified bugs enabling (1) emitting events E_S on $\mathbb{B}_S$ without the corresponding effect e_S; (2) creation of proofs π without the corresponding source event E_S; (3) emitting events E_D on $\mathbb{B}_D$ without knowledge of a corresponding proof π; (4) effects e_D on $\mathbb{B}_D$ not reflected by emitted events E_D. More concretely, we find four vulnerability types: (1) inconsistent deposit logic; (2) private key compromises; (3) incomplete proof verification; and (4) privilege escalation. Bridges can also be hijacked via token contracts [41] or common smart contract bugs like reentrancy or integer overflow. These bugs are not specific to cross-chain bridges but are generic smart contract vulnerabilities addressed by existing security analysis approaches, e.g., Slither [42]. The focus of this paper is on cross-chain bridge-specific vulnerabilities, which account for most attacks in this domain.

We derived the bug classes from state-of-the-art research [34, 36]. The bug classes cover most known bugs but are not meant to be comprehensive. For instance, entire bug classes such as cross-chain front running, replay attacks, and infinite approvals have not been considered in these studies. We exclude front-running because there are no known cross-chain front-running attacks. On the other hand, we believe that this is a general open research problem on its own which requires dedicated mitigation techniques similar to classic front-running attacks [43, 44]. Further, infinite approvals are merely smart contract permission issues rather than a specific feature of cross-chain bridges. That is, infinite approvals require that the victim contract does not validate the token contracts it interacts with. Hence, this bug class belongs to the unchecked calls vulnerability class in smart contracts [45, 46]. Lastly, BRIGADE does not require a special policy for replay attacks, as the incomplete verification policy detects when a proof has been used before. Next, we elaborate on each vulnerability type.

Inconsistent Deposit Logic: An inconsistent deposit logic bug [41] occurs when a bridge's router contract fails to secure deposits properly [2, 3, 15]. This causes the router to emit a deposit event (E_S) on $\mathbb{B}_S$ and release funds (e_D) to an attacker on $\mathbb{B}_D$ without locking (e_S) the corresponding amount (①). Although the bug originates in the contract, exploiting it requires the bridge, as emitting E_S alone is harmless. Funds are only released upon authorization by the off-chain component, reacting to E_S. For example, an attacker can send a malicious deposit transaction to $\mathbb{B}_S$ using an arbitrary ERC20 token, causing the bridge to emit a deposit event without verifying the transfer. As a result, the bridge mints tokens on $\mathbb{B}_D$. This bug may stem from incorrect token implementation or treating a native token as an ERC20 token. At least 84 million USD were lost due to this bug class [15, 17].

Private Key Compromise: In centrally operated bridges (notary) bridges, the ability to generate a proof π for a source event E_S is tied to a specific private key (or key set). Data leaks or similar incidents can compromise the keys, allowing an attacker to produce a proof π even if E_S was not emitted (②) and submit a valid transaction to the router contract on $\mathbb{B}_D$ [3, 4, 13]. Ronin Bridge lost 624 million USD in such an attack, where stolen keys were used to unlock the

```
1 modifier onlyOwner(){require(owner==msg.sender);}
2 function set_owner(address newOwner) public onlyOwner {...}
3 function cross_call(address target, bytes calldata data) external {
4   emit CrossChainCall(target, data);}
5 function call_contract(address target, bytes calldata data) external {
6   (bool _, bytes memory _) = target.call(data);}
```

Listing 1.1: A simplified privilege escalation bug in a cross-chain environment.

multi-signature wallet and release locked ETH [13]. These transactions execute automatically on the blockchain and cannot be stopped. However, by analyzing all router transactions, compromised keys causing direct token transfers can be identified as they lack the corresponding E_S. Private key compromise is outside this work's scope, as it is challenging to prevent via software measures [47]. Secure key storage remains essential for bridge security.

Incomplete Verification: This resembles an integrity issue [41], allowing an attacker to falsely prove the existence of a transaction [2, 4, 14, 16] or event [3]. It often arises from faulty signature verification [16, 48] or other proof checks [14, 41] in the off-chain component or the router contract on $\mathbb{B}_D$. Attackers can exploit this by forging a valid proof without depositing assets, causing fund losses. Depending on the off-chain component's design, the attacker either tricks the off-chain component into generating a valid proof π (e.g., due to a bug) or independently generates π so that the router contract on $\mathbb{B}_D$ verifies it. In Figure 1, an attacker does this by generating or provoking a valid proof (③). The attacker does not need a transaction on the source chain $\mathbb{B}_S$, as proof verification is bypassed, making the attack unmitigable without modifying the smart contract. Recently, several attacks exploited incomplete verification [14, 16, 48].

Privilege Escalation: Bridges often have an authority, e.g., a bridge operator, to halt operations or unlock tokens. In a privilege escalation, attackers gain access to such privileged functions, for instance allowing them to release locked funds [2, 4]. For instance, if the operator forwards a cross-chain call targeting the $\mathbb{B}_D$ router contract, the router function executes with the operator's privileges but with arguments from the call initiator. Listing 1.1 provides a concrete example: an attacker initiates a cross-chain transaction (Line 3) with a payload instructing the bridge to call itself (Line 5). The `calldata` includes the `set_owner` function selector, padding, and the attacker's address. Since the bridge contract initiates the `call`, its address becomes `msg.sender` for `set_owner`, bypassing flawed permission checks that trust internal calls, resulting in unauthorized contract takeover. This may cause unintended effects e_D (e.g., additional fund releases) on the destination chain, deviating from expected effects E_D from a cross-chain call by an unprivileged user (④). The most prominent privilege escalation, PolyNetwork [12], lost 611 million USD.

3.2 Threat Model

We assume an attacker whose primary goal is to exploit logic errors and permission bugs of smart contracts deployed in a bridge. As is the case for all real-world

attacks [12–17], the adversary aims to steal assets and perform privilege escalation attacks. Our threat model aligns to prior work in the field [3, 20, 32, 33]. Further, we assume a blockchain environment where the adversary can read state and submit transactions. Consensus-layer attacks and private key theft are beyond the scope of this paper. The core attack vector involves manipulating the bridge protocol to either 1) emit a fraudulent event or 2) incorrectly validate an incoming message. Our work aims to defend against such attacks by directly intercepting and validating these potentially malicious transactions.

3.3 Challenges

Existing approaches focus on the detection of generic bugs in smart contracts residing in a single blockchain [28, 45, 49–53]. In contrast, bridge operators aim to monitor and enforce the security of the whole bridge operation which involves the synchronized execution of several router contracts across different blockchains. However, doing so requires addressing various challenges:

Challenge 1: Data Sources: Cross-chain transactions typically include a source chain, an off-chain component, and a destination chain. Each of these are critical to the transaction flow. As a result, the collection and coordination of data from diverse sources, as well as maintaining consistency and accuracy is crucial to enable effective attack detection.

Challenge 2: Bridge Heterogeneity: Data and events need to be collected from different bridge architectures and formats. Hence, the development of a homogeneous format is required. Further, data and event retrieval should be possible through a uniformed interface, allowing developers to handle distinct bridges uniformly.

Challenge 3: Detection Scope: Given the rapid developments and changes in cross-chain bridges, a security framework needs to quickly react to new attack patterns and vulnerabilities. As a result, we need to allow for modular and flexible implementation of new detection modules.

4 Brigade

The full semantics of cross-chain bridges only become evident from the interactions between the individual components. This limits static analysis approaches [54], which often focus on single components [20, 32] and, thus, have a limited view of the overall system. Machine learning approaches [33] analyze the behavior of the whole system at runtime, offering deeper introspection. However, they require learning a comprehensive transaction model, which is difficult in practice, and the bridge remains vulnerable during the training phase.

We introduce Brigade, the first framework designed to automatically detect and counteract malicious transactions in bridges. In contrast to previous approaches [3, 20, 32, 33], Brigade formulates universal security properties specific to bridges. This enables just-in-time detection of cross-chain bridge bugs, and ensures the maintenance of essential security even in the presence of logical

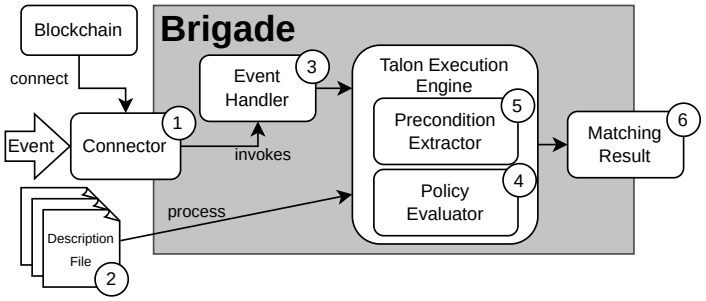

Fig. 2: BRIGADE's components and execution flow to confirm a transaction.

bugs within underlying smart contracts. BRIGADE operates as a non-invasive, parallel process by subscribing to router contract events and analyzing the resulting transactions. This architecture preserves the bridge's trusted computing base and allows for seamless integration without altering the existing codebase. We apply both universal and custom policies to each transaction, flagging any violations as malicious. In contrast to static approaches [20, 32], we achieve a high level of introspection by analyzing cross-chain bridges at runtime. Further, BRIGADE's policies do not require learning or training phases, and often require no configuration at all (cf. Section 4.4). Thus, there is no point in time at which the bridge is open to attack.

4.1 Architecture

BRIGADE's architecture (Figure 2) is designed to address key challenges specific to the analysis of cross-chain bridges (cf. Section 3.3). BRIGADE connects to the target blockchain ① and subscribes to relevant events. By decoupling the event handling from the transaction analysis, BRIGADE offers two key advantages. First, it enables extensibility: implementing a new blockchain connector ① allows BRIGADE to support additional blockchains. Second, event handlers ③ can integrate with various sources to replay historical transactions or simulate transactions. This modularity addresses Challenge 1, while initial data type conversions by the event handler address Challenge 2. Once an event is received, BRIGADE's execution engine validates the transaction using policies defined in the *description files* ②. The TALON execution engine, comprising the policy evaluator ④ and the precondition extractor ⑤, evaluates policies based on the extracted data. BRIGADE provides a core set of formally verified *universal policies*, which can be extended with custom logic for application-specific requirements, addressing Challenge 3. Individual policy checks are aggregated into a final verdict ⑥, which can trigger an automated response. Integrating BRIGADE into a bridge requires only a one-time adaptation to the bridge's design, as details like event

topics and asset descriptors differ. Next, we describe each component of BRIGADE in detail.

4.2 Blockchain Connector

The blockchain connector ① links blockchains and off-chain components to BRIGADE. Each time BRIGADE connects to a new blockchain, a new connector must be created. In BRIGADE, we provide the implementation of two blockchain connectors: an EVM blockchain connector to cover all EVM-based blockchains and a dedicated Solana blockchain connector. Each connector establishes a WebSocket connection to the blockchain to subscribe and listen for relevant events. Event data is forwarded to the event handler ③, which adds contextual information such as the transaction hash, event type, and the payer's balance before and after the transaction. Note that our connectors are capable of querying arbitrary blockchain data. We also support private blockchains to simulate transactions in controlled environments or replay historical transactions by issuing RPC requests for past blocks and forwarding the resulting event to the event handler ③.

4.3 Event Handler

Depending on the connected blockchain, the event handler fetches data from the blockchain and transforms it into a uniformed format (Challenge 2). For each event, the event handler maps the associated policies. Our policies are specified in description files in a domain-specific language (DSL), called TALON, that we designed for this purpose. Our TALON execution engine executes the policy checks. When multiple policies are specified for an event, the event handler sequentially executes all policies and consolidates the results. If a single policy prohibits a transaction, the event will be blocked, regardless of the success of other policies.

4.4 Universal Policies

As a core contribution, we developed three formally verified universal policies that detect the attack classes described in Section 3.1. These policies enforce fundamental security invariants applicable to bridge designs. We provide an overview of these foundational policies in this section and refer to Appendix A for their TALON implementation.

Inconsistent Deposit Logic: To detect this bug class, our policy checks the payer's account balance before and after the deposit transaction to verify that the deposit event E_S released by the source contract coincides with the source effect e_S (the transfer of the correct amount of funds from the claimed sender to the source contract). In BRIGADE, this policy is implemented by checking upon receipt of a transaction $\mathbf{tx_{deposit}}$, which triggers a deposit event, that the sender's balance change aligns with the amount recorded in the event. More precisely, the following condition is checked:

$$\mathbf{tx_{deposit}}.\text{balance}_{\text{before}} = \mathbf{tx_{deposit}}.\text{balance}_{\text{after}} + \mathbf{tx_{deposit}}.\text{amount} + \text{fee} \quad (1)$$

where $\mathbf{tx_{deposit}}.\text{balance}_{\text{before}}$ and $\mathbf{tx_{deposit}}.\text{balance}_{\text{after}}$ denote the sender's balance before and after executing $\mathbf{tx_{deposit}}$ and $\mathbf{tx_{deposit}}.\text{amount}$ denotes the deposit amount recorded in the event. The check additionally considers that senders need to pay some fee for executing $\mathbf{tx_{deposit}}$ that are also deducted from the sender's account. As fees differ based on the configuration of the bridge, we simplify this by checking that the left-hand side of the equation is greater than the right-hand side. Thus, we take arbitrary bridge fees into account.

Incomplete Proof Verification: Incomplete proof verification enables an attacker to produce a valid (deposit) proof without the prior execution of the corresponding source event E_S. To mitigate this, upon receipt of a transaction $\mathbf{tx_{withdraw}}$ that would trigger a withdrawal event E_D on the destination chain, BRIGADE checks whether there has been a transaction $\mathbf{tx_{deposit}}$ in the past that corresponds to the proof that is provided with $\mathbf{tx_{withdraw}}$:

$$\mathbf{tx_{withdraw}}.\text{proof} = (\pi,\ \text{id}) \Rightarrow \exists \text{tx}_{\text{deposit}} \in \text{tx}_{\text{past}} : \text{proof}(\text{tx}_{\text{deposit}}) = (\pi,\ \text{id}) \quad (2)$$

Here, $\mathbf{tx_{withdraw}}.\text{proof}$ denotes the proof-relevant data $(\pi,\ \text{id})$ provided as argument to $\mathbf{tx_{withdraw}}$. π is the proof object itself (which is to be verified), and id denotes relevant proof metadata (e.g., the receiver of the withdrawal). The set tx_{past} denotes the set of all past transactions recorded on the blockchain, and $\text{proof}(\text{tx}_{\text{deposit}})$ denotes the proof data created when executing $\text{tx}_{\text{deposit}}$. The check ensures that BRIGADE would only permit the bridge to send withdrawal transactions with proofs that have been created for proper deposit events.

BRIGADE monitors and detects any withdrawal transaction. This means that even attack transactions with accepted proofs—such as by compromised bridge keys or a contract vulnerability—BRIGADE would promptly detect such unauthorized activity.

Privilege Escalation: Privilege escalation occurs if cross-chain function calls target the destination router contract and enable unintended effects e_D on the destination chain.

$$\mathbf{tx_{execute}}.\text{target} \neq \text{address}_{\text{router contract}} \quad (3)$$

To mitigate this, upon retrieval of a transaction $\mathbf{tx_{execute}}$ that executes a cross-chain function call (on the destination router contract) with target $\mathbf{tx_{execute}}.\text{target}$, BRIGADE checks that $\mathbf{tx_{execute}}.\text{target}$ is not the router contract (with address $\text{address}_{\text{router contract}}$) on the destination chain.

Non-Standard Behavior: Non-standard use cases, such as non-ERC-20 tokens, are supported as long as they implement a balanceOf function—the only function required by the policies. Any other non-standard behavior can be accommodated by adapting the policies.

4.5 TALON Execution Engine and Report

To facilitate policy design, we developed the DSL TALON, which allows for the intuitive specification of policies while abstracting low-level blockchain complexities like networking and data type handling. Further, TALON policies

```
1 "precondition": {
2  "balance_after": "call(eth,invoke,[$data,$addr,$block_num])",
3  "balance_before": "call(eth,invoke,[$data,$addr,($block_num-1)])",
4  "self_after": "call(eth,invoke,[$self_call,$addr,$block_num])",
5  "self_before": "call(eth,invoke,[$self_call,$addr,($block_num-1)])"},
6 "policy": ["$balance_before - $token_amount == $balance_after",
7  "$self_before + $token_amount == $self_after"]
```

Listing 1.2: Preconditions and policies to prevent an Inconsistent Deposit bug.

can be created and modified at runtime, enabling development without downtime for Brigade.

Policies in Talon consist of two components, *preconditions* which specify the data required for evaluation and the *policy* logic itself. As an example, Listing 1.2 shows Brigade's policy preventing inconsistent deposit errors for ERC20 tokens. The preconditions fetch the user's balance in the token contract and the router contract's balance. Line 7 defines that after the transaction, the balance must equal the previous balance plus the bridged amount, while the same amount must be added to the router contract's balance.

For expressing policies, Talon supports various data types (e.g., EVM's uint256) and includes typecasting to transform representations, such as converting a hexadecimal string from transaction data into an integer or address. It also provides standard operations, a global map for maintaining state across transactions, and a call function for live data retrieval from smart contracts.

To evaluate policies, the Talon Execution Engine integrates a policy evaluator ④ and a precondition extractor ⑤. The extractor gathers necessary information from the transaction and the blockchain. Preconditions may invoke custom functions to obtain data from on- and off-chain sources. Talon generalizes these results into a common representation, easing the conversion of blockchain-dependent formats, such as Ethereum addresses, into unified hexadecimal representations and vice versa. Finally, the policy evaluator checks compliance and generates a *Matching Report* (⑥). If all policy checks for a transaction pass, it is deemed compliant. If any check fails, the report identifies the violation and Brigade recommends halting the transaction.

4.6 Policy Enforcement

Brigade primarily functions as a runtime monitor that reliably identifies harmful transactions. Whether such transactions can be halted depends on the bridge design and the type of violation. For instance, attacks exploiting inconsistent deposit logic bugs, which rely on wrongly emitted events on the source chain, can be stopped by ensuring the off-chain component only processes events marked compliant by Brigade. However, bugs that manifest in malicious transactions on the target chain (e.g., incomplete verification bugs) may only be detected afterward. Since transactions in blockchains like Ethereum are public before execution, and Brigade's execution is decoupled from event retrieval, it can also simulate future events to detect attacks before harmful transactions execute. Still,

as noted in various studies [34, 36], no reliable solution currently exists to prevent such transactions from executing. One option is to redesign the target chain router contracts to require off-chain confirmation before processing transactions, but this adds cost, complexity, and centralization risks. Thus, effective bridge-design independent runtime enforcement of general security properties upon attack detection remains an interesting but orthogonal research question.

4.7 Replaying Historical Transactions

In addition to real-time enforcement, we support historical analysis by replaying past transactions, allowing Brigade to be used throughout a bridge's entire lifecycle. This capability is critical in practice for three reasons: (1) It enables detailed forensic analysis for incident response. By replaying an attack's event sequence, operators can inspect transaction payloads and correlate activities across chains to reconstruct the exploit path. (2) It facilitates safe policy development by allowing operators to backtest new Talon policies against historical data to validate efficacy and avoid false positives. (3) It enables proactive threat hunting. When a new vulnerability class is disclosed, operators can quickly instantiate a corresponding Talon policy and scan the bridge's transaction history to check for exposure.

4.8 Implementation

We implemented Brigade and Talon in 9,031 lines of Rust, with the core system comprising 2,894 lines and the language implementation accounting for 6,137. The connector integrates with Ethereum and Solana using the *reqwest* (HTTP) [55] and *WS* (WebSocket) [56] libraries, while we use *Ethnum* [57] for Ethereum data types and *base58* [58] for Solana addresses. Our universal policies are expressed in only 86 lines of Talon, including customizations for our evaluation setup, and the Tamarin model was proven in 250 lines.

5 Evaluation

This section presents our evaluation of Brigade, which combines formal analysis with empirical testing. We begin by formally verifying our universal policies from Section 4.4 in Tamarin to show they mitigate the threats described in Section 3.1. Next, we demonstrate Brigade's practical effectiveness against twelve real-world attacks and compare its detection scope against state-of-the-art systems. We conclude by measuring performance, confirming that Brigade executes well within the block time and without false positives. Our experiments were run on a 3.3GHz AMD Ryzen 8-core processor with 64GB RAM and a Linux-based OS.

5.1 Validation of Universal Policies

We validate the effectiveness of our universal policies in preventing attacks within our threat model Section 3 by modeling core bridge functionalities using the

Tamarin protocol analysis tool [59]. Our bridge model incorporates the vulnerabilities outlined in Section 3.1, and we demonstrate that enforcing BRIGADE's universal policies mitigates these attacks. Tamarin models security protocols as multi-set rewrite rules representing state transitions. The resulting labeled transition system describes all possible executions and can be automatically verified against temporal first-order properties, such as ensuring critical events do not occur during execution.

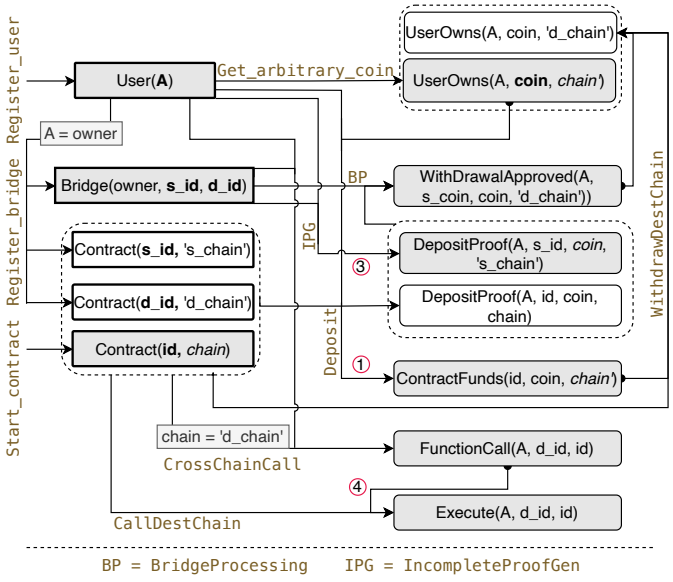

Fig. 3: Cornered/rounded boxes model persistent/volatile facts. Multi arrows represent rules (circular endings consume source facts). Italicized/bold variables indicate fresh/unbound identifiers. Constants are written in quotation marks. Gray boxes determine variable names.

We illustrate our model in Figure 3, where *facts* (in boxes) represent protocol state. For instance, `fBridge`(`vOwner`, `vScontract`, `vDcontract`) denotes a bridge owned by `vOwner`, established through router contracts `vScontract` and `vDcontract`. The *rules* dictate how facts are consumed and produced; the rule `rDeposit` requires a user `vUser`, a contract `vContract` on blockchain `vChain`, and a coin `vCoin` owned by `vUser` on another blockchain `vChainprime`. When these conditions are satisfied, it generates two new facts: `fContractFunds`(`vContract`, `vCoin`, `vChainprime`) and `fDepositProof`(`vUser`, `vContract`, `vCoin`, `vChain`), while consuming `fUserOwns`(`vUser`, `vCoin`, `vChainprime`). Different rules may produce the same facts; for instance, the `rIPG` rule generates deposit proofs

without restrictions for existing bridges, creating a fact `fDepositProof`(`vUser`, `vScontract`, `vCoin`, `cSchain`) for arbitrary coins.

This model deliberately underspecifies bridge operations, highlighting the susceptibility to the bugs discussed in Section 3.1. For example, the `rDeposit` rule can generate a deposit proof for blockchain `vChain`, even if the user deposited the coin on another blockchain (`vChainprime`), modeling **Inconsistent Deposit Logic** bugs (cf. ① in Figure 3). Similarly, the `rIPG` rule allows for the generation of arbitrary deposit proofs, reflecting the potential for forging valid proofs for coins that have not been deposited, as seen in **Incomplete Proof Verification** bugs (cf. ③ in Figure 3). Additionally, the model captures **Privilege Escalation** bugs (cf. ④ in Figure 3) by modeling unrestricted cross-chain function calls (rules `rCrossChainCall` and `rCallDestChain`), which may enable the invocation of a router contract within such calls.

We validate our model by formulating temporal first-order properties that characterize secure bridge operation and demonstrate that, without enforcement, the model violates these properties. Specifically, a secure bridge must ensure: (1) any coin withdrawal by a user `vUser` on the destination chain is backed by a prior deposit on the source chain, and (2) if the router contract on the destination chain executes a cross-chain function call, it must not be directed to the router contract or must originate from the bridge owner. Subsequently, we formulate the universal policies implemented by Brigade in our Tamarin model and demonstrate that enforcing these policies ensures compliance with properties (1) and (2).

We enforce the **Inconsistent Deposit Logic** policy by restricting the `rDeposit` rule to execute only if the source contract received the payment for which the deposit proof is generated, thereby preventing inconsistent proofs while abstracting from concrete amounts to simplify encoding of the universal policy. Compliance with the **Incomplete Proof Verification** policy requires that a withdrawal transaction (rule `rBP`) is approved only if a corresponding deposit (rule `rDeposit`) was conducted beforehand, ensuring attackers cannot bypass the bridge to send transactions accepted by the destination router. The **Privilege Escalation** policy is enforced by forbidding cross-chain function calls (rule `rCrossChainCall`) to the bridge's destination router. These universal security policies are implemented as logical restrictions in Tamarin, showing that all executions of the original bridge model—previously permitting bugs—comply with them and thus satisfy bridge security properties (1) and (2). This theoretical validation confirms the effectiveness of the universal policies in preventing the bug classes outlined in Section 3.1.

5.2 Exploit Detection Capabilities

We evaluate Brigade's exploit detection capabilities by deploying a cross-chain bridge with a router contract on Ethereum. The bridge implements functions that contain exploitable bugs within the scope of our threat model (see Section 3). Then, we execute malicious transactions to demonstrate Brigade's ability to detect and prevent these exploits. For each bug, we deploy an exploitable smart contract

```
function bridge_to(uint256 chain_id, string memory dest, uint256 amount)
    external payable{
  require(amount > 0, "Amount <= 0");
  bank[msg.sender] += amount;
  emit Deposit(chain_id, dest, amount);}
function bridge_token(address token_address, address to, uint256 amount)
    external{
  require(amount > 0, "Amount <= 0");
  IERC20(token_address).transferFrom(msg.sender, address(this), amount);
  emit DepositToken(to, amount, token_address);}
```

Listing 1.3: A contract vulnerable to inconsistent deposit logic.

```
function deposit(address to) external payable{...
  bytes32 proof=keccak256(abi.encodePacked(to,msg.value));... }
function withdraw(bytes32 proof, uint256 amount, address to) external {
  require(amount > 0, "Withdraw <= 0");
  bytes32 _proof = keccak256(abi.encodePacked(to,amount));
  require(_proof == proof, "Wrong Proof");
  emit Withdraw(amount, proof);}
```

Listing 1.4: A contract vulnerable to incomplete verification.

and a universal policy to detect the exploitation of the bug. The experiments validate that these universal policies are sufficient to detect exploitation.

Inconsistent Deposit Logic: An *inconsistent deposit logic* bug (Section 3.1) emerges when the depositing logic fails to verify that the sender's payment equals the bridged amount. Deposits can involve either (1) native tokens (e.g., ETH), or (2) ERC20 tokens (e.g., WETH), which differ in implementation: native tokens are held directly by accounts, while ERC20 tokens are tracked via smart contracts. Due to this technical difference, we deploy separate policies for native and ERC20 tokens described in Section 4.4.

Listing 1.3 illustrates a vulnerable contract where the functions `bridge_to` and `bridge_token` transfer assets without verifying that the bridged amount equals the received payment. This enables exploitation by supplying fewer funds than specified or spoofing valid tokens.

To address this, BRIGADE enforces separate policies for native and token deposits. For native tokens, it records the balance before and after execution, accounts for gas cost, and verifies that the received value matches the expected deposit. For ERC20 tokens, BRIGADE retrieves the token address and bridged amount from the transaction input data and queries the `balanceOf` function on both the sender and the contract to ensure balance changes are consistent with the bridged amount. Transactions violating these checks are denied, effectively preventing this class of bug.

Incomplete Verification: Listing 1.4 illustrates an incomplete verification bug (see Section 3.1). The issue is that the attacker controls all components required to generate the proof hash, enabling them to create a valid proof. The bridge mitigates this by verifying that `msg.sender` is the actual proof owner. BRIGADE employs a two-stage universal policy to protect against this bug.

When the contract creates the proof and emits the corresponding event, like a deposit or swap event, the first part of the policy fetches and stores the proof and recipient. The second part of the policy waits for a retrieval event, such as an `Withdraw` event in Listing 1.4, and checks whether the map contains the proof and the correct receiver. If the proof is non-existent or belongs to a different receiver, BRIGADE rejects the transaction. Once a correct transaction uses a proof, the proof is removed from the map to prevent multiple retrievals of the same transaction. By handling the payout in a separate function initiated through a trusted bridge account, BRIGADE can stop the payout and mitigate the attack.

Privilege Escalation: The contract in Listing 1.1 is vulnerable to privilege escalation because any account can become the owner by calling `set_owner` through a cross-chain transaction (`call_contract()`), bypassing the `onlyOwner` modifier and gaining control over the contract.

To mitigate this, BRIGADE implements an *address denylist policy* (see Section 4.4), which tracks uncallable addresses, i.e., router and token contracts, and ensures that functions do not interact with these addresses. This policy effectively prevents unauthorized interactions with critical bridge components.

5.3 Real-World Attacks

To evaluate BRIGADE's effectiveness in a real-world environment, we integrated BRIGADE into twelve recently exploited open-source cross-chain bridges connecting Ethereum, Solana, and Binance Smart Chain (BSC). We selected the bridges from the rekt Leaderboard [19] and incident reports [48, 60, 61] based on their high rankings in terms of total value lost. We excluded closed-source bridges, and exploits limited to smart contract or front-end vulnerabilities. While a detection of compromised private keys is not possible for Brigade, profiting from compromised private keys depends on the usage of the key. Signing cross-chain transactions like withdraw or unlock is detected by the incomplete proof verification, as the source chain is missing the associated transaction. However, signing refund transactions or using the key to bypass access control checks cannot feasibly be covered by BRIGADE, as it is undecidable whether the stolen keys are used by a legitimate user or not.

We group the bridges by their primary vulnerability class and the reported total value lost (in parentheses): *Privilege Escalation*: PolyNetwork ($611M) [12]; *Incomplete Verification*: Ronin ($600M) [13], BNB Bridge ($586M) [14], Wormhole ($326M) [16], Nomad ($190M) [62], Harmony Horizon ($100M) [63], ChainSwap ($8M) [48], Force Bridge ($4M) [64]; and *Inconsistent Deposit Logic*: Qubit Bridge ($80M) [17], ThorChain ($4.9M) [65], Meter.io ($4.3M) [15], Multichain ($3M) [60, 61]

We executed the original attack transactions 500 times, and 500 other benign transactions. Each malicious transaction represents a working exploit that would result in a loss of funds for the bridge. BRIGADE identified every malicious transaction and allowed all benign transactions. Our findings demonstrate that BRIGADE could have prevented the loss of 2,517 billion USD across these bridges. Here, we exemplarily discuss exploits from the three different categories,

explaining how BRIGADE's universal policies protect against them. We refer to Appendix C for a detailed analysis for all real-world attacks.

Qubit's *QBridge* lost 80 million USD due to an *inconsistent deposit logic* issue [17]. The `deposit` function failed to burn ETH, allowing an attacker to trigger a deposit event without actual token transfer. We apply our universal inconsistent deposit logic policies, adjusting preconditions to retrieve the token address from a resource ID. Calling `balanceOf` on `0x00...00` returns an empty result, failing the policy, and thereby denying the exploit.

The *ChainSwap* bridge suffered from an *incomplete verification* bug, resulting in an 8 million USD loss. The attacker exploited a flaw allowing quota reset for each new signatory, bypassing withdrawal verification by changing the signatory and nonce on each transaction. BRIGADE detects this bug using the universal incomplete verification policy, which marks repeated withdrawals as malicious after the first successful transaction.

A *privilege escalation* in PolyNetwork allowed an attacker to exploit the router contract's execution logic, appointing any account as administrator. The attacker changed the owner of the *EthCrossChainData* contract, bypassing permission checks and withdrawing all funds from the bridge's router contract. This exploit was repeated across connected blockchains, resulting in a 611 million USD loss. The bug would have been prevented by BRIGADE's *address denylisting* policy. This policy stores the *EthCrossChainData* address key and denies transactions involving this contract to prevent ownership changes. Developers can adjust this address, with options to customize it to ensure privileged owner access. To prevent the attack, the router contract should only emit an event in cases that privileged entities, such as BRIGADE or the off-chain component, initiate the function. Our experiment shows that BRIGADE reports a policy violation when the protected contract is called, while still allowing arbitrary cross-chain transactions that do not involve the protected contract.

5.4 Comparison with State of the Art

We compare BRIGADE with state-of-the-art approaches in Table 1, evaluating their capacity to detect the impactful real-world attacks from Section 5.3. To ensure a fair evaluation, we attempted to reproduce the results of each approach. We successfully executed XGuard [20] and compared its findings against ours. Since artifacts for SmartAxe [32], XScope [3], and Hephaestus [33], are either unavailable or defective, an empirical comparison is impossible. Hence, we base our comparison on the design choices and capabilities presented in their respective publications. Our analysis reveals significant limitations in these approaches, which BRIGADE overcomes.

Our experiments show that XGuard fails to detect any of the critical cross-chain bugs found by BRIGADE. While XGuard flags common smart contract issues using its Slither-based scanner, it misidentifies the root cause in cases like the Qubit [17] and Meter.io [15] bridges, pointing to a missing event instead of the core address mismatch vulnerability. Similarly, Hephaestus [33] trains a model on historic transactions, an approach inherently unable to detect attacks

Table 1: Comparison with state-of-the-art approaches on real-world exploits. BRIGADE is the only approach to detect all listed vulnerabilities.

	PolyNetwork [12]	BNB Bridge [14]	Wormhole [16]	Nomad [62]	Qubit Bridge [17]	ChainSwap [48]	Multichain [61]	Meter.io [15]
XGuard [20]	✗	✗	✗	✗	✗	✗	✗	✗
XScope [3]	✓	✗	✗	✗	✓	✓	✓	✓
SmartAxe [32]	✗	✗	✗	✗	✓	✓	✓	✓
Hephaestus [33]	✗	✗	✗	✗	✗	✗	✗	✗
BRIGADE	✓	✓	✓	✓	✓	✓	✓	✓

that exploit logic with validly formed transactions like the PolyNetwork [12] and Nomad [62] exploits. Unlike BRIGADE, which is blockchain agnostic, Hephaestus is purpose-built for a specific HyperLedgerFabric/Ethereum environment. Thus, neither XGuard nor Hephaestus can detect the critical exploits detailed in Table 1.

XScope [3] and SmartAxe [32] exhibit different but equally critical limitations. XScope relies on a fixed, predefined rule set, which prevents the detection of unknown or zero-day vulnerabilities. Its analysis is also restricted to EVM-compatible chains, excluding increasingly important ecosystems like Solana. Consequently, major exploits on other platforms, such as the Wormhole hack on Solana, would go undetected by any of these prior systems. In contrast, BRIGADE introduces universal, extensible policies via a powerful DSL. This approach enables BRIGADE to overcome the limitations of prior work by covering all analyzed bugs and uniquely providing support for non-EVM blockchains.

5.5 Performance

We evaluate BRIGADE's performance to confirm its suitability for real-time detection, using Ethereum (12s) and Solana (0.4s) block times as baselines. Our experiment measured the latency of enforcing twelve real-world policies over 1,000 transactions for each target bridge (Section 5.3). Our Results (Figure 4) confirm that BRIGADE operates with negligible overhead, as execution times are significantly lower than native block production times.

Execution times vary: the privilege escalation policy in PolyNetwork has a median of 7ms (no RPCs), while the inconsistent deposit logic policy on QBridge reaches a median of 77 ms (four RPCs). All medians remain well below Ethereum's 12s block time. The highest times (69–155ms) occur for the inconsistent deposit logic policy, due to blockchain node queries. Even for Solana's 400ms block time, the incomplete verification policy for Wormhole executes faster. Since BRIGADE runs independently of block processing and completes within one block time, it is effective for Ethereum, Solana, and BSC bridges. Our results confirm BRIGADE's

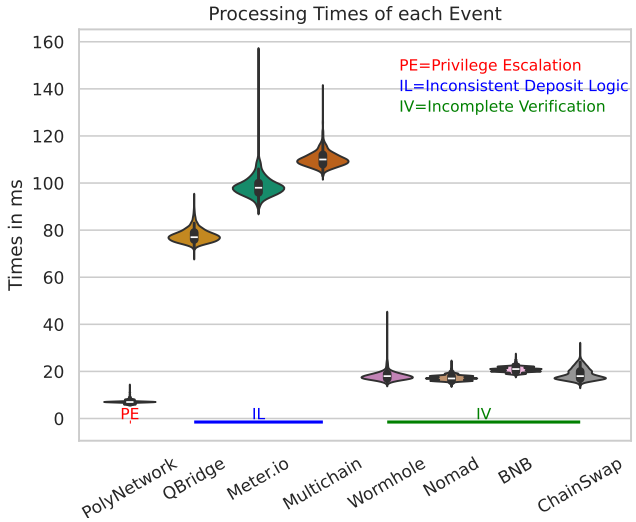

Fig. 4: Violin plot of the time used to process an event.

suitability for real-time deployment on high-throughput bridges, providing robust security while imposing negligible overhead compared to native block production.

6 Discussion

Deploying Brigade within bridges enhances security by enabling effective management of policy violations through integration with the bridge's off-chain components. In centralized setups, Brigade connects to a single event-forwarding node while in decentralized environments, it interfaces with relayers for each blockchain, ensuring comprehensive transaction coverage.

Integrating Brigade into router contracts allows for the enforcement of universal policies on various transactions, including withdrawals executed directly by the router on the destination chain. Direct withdrawals—those relying on proof π and not transiting the bridge—are executed immediately upon block inclusion. While in theory, Brigade could front run such attacks by building a defensive transaction, as prior work has shown [66], in practice this requires rapid detection from the transaction pool and adequate defensive capabilities, such as pausing the bridge or revoking access for specific addresses. We recommend revising router contracts to disallow unlocking tokens solely based on user input. Instead, contracts should emit events to Brigade, simulating the transaction to verify policy compliance. Brigade then unlocks the tokens directly, or invokes a trusted authority to release them.

For transactions transiting the bridge—such as deposits—bridge nodes are responsible for processing. Enforcing Brigade's decisions depends on the protocol and whether honest nodes access its output in time to halt malicious propagation.

Detection, Prevention, and Continuous Monitoring. BRIGADE's limitations arise from the design of bridges and blockchain semantics. First, while BRIGADE reliably detects policy violations, it cannot identify the underlying cause of a vulnerability as BRIGADE merely reports malicious behavior, e.g., incorrect balances, incomplete transaction chains, missing events.

Second, BRIGADE has limited influence over bridge execution, especially for operations executed within a single atomic transaction. Failures such as flawed ERC20 behavior or single-chain inconsistencies cannot be intercepted or halted mid-execution; BRIGADE detects them after they occurred. Preventing token losses in these cases requires explicit protocol support, e.g., BRIGADE as a pre-unlock or pre-burn guard. A potential extension is a rescue transaction that front-runs an exploit, but this would necessitate real-time mempool monitoring and substantially greater computational resources.

Third, BRIGADE provides continuous monitoring with practical constraints. Policies are hot-loaded, enabling real-time updates without downtime, but adding or removing blockchains requires a brief restart. This affords near-continuous coverage but does not overcome inherent limits imposed by bridge architectures or transaction atomicity.

Covering Blockchains. As demonstrated in Section 5 (Harmony, Ronin, Nervos), extending BRIGADE to additional blockchains is straightforward: new chains require only the configuration of additional RPC endpoints. While the EVM connector works for all EVM-compatible chains, non-EVM ecosystems may expose events only as raw logs. In such cases—e.g., Solana—developers must provide a lightweight trigger mechanism (such as Anchor events or structured log messages) to provide event data for BRIGADE. Any chain offering RPC-based event subscriptions works out of the box with the generic or Ethereum connector, which we confirmed on Harmony, Ronin, Polygon, Snowtrace, and others. Policy calls are chain-agnostic; BRIGADE issues arbitrary RPC requests to any target. For example, we integrated Nervos (a non-EVM blockchain) to verify Force Bridge proofs without modifying BRIGADE, requiring only an additional API endpoint. Overall, apart from event subscription differences on certain non-EVM chains, we encountered no obstacles when extending BRIGADE to new environments.

7 Related Work

Smart Contract Security. Bridge operators may consider existing approaches to secure router contracts and monitor cross-chain transactions. Static analysis approaches [21–28] analyze smart contract code to identify vulnerabilities. Some of these approaches rely on formal verification [23, 24, 26], burdening operators with sound and correct specifications of contract behavior. Other approaches leverage static program analysis [25] or symbolic execution [21, 22, 27, 28] to identify vulnerabilities. While these approaches can identify vulnerabilities automatically, they imply trade-offs between precision and scalability, and often require manual interpretation of the results.

Dynamic smart contract analysis is dominated by fuzzing approaches [50, 51, 67–69], which generate test cases to trigger vulnerabilities but focus only the contract under test, ignoring interplay of components.

Runtime monitoring approaches [29, 70] detect attacks or anomalies at runtime. Sereum [29] performs dynamic taint analysis for reentrancy attacks, while ÆGIS [70] is extensible to other vulnerabilities through patterns-based detection but requires substantial modifications to the validator ecosystem. Both are limited to the Ethereum platform and Ethereum specific vulnerabilities.

Most of these approaches cover only Ethereum-specific vulnerabilities such as reentrancy, overflows, and access control, with VRust [25] and FuzzDelSol [50] being the only exceptions. Both, VRust and FuzzDelSol are limited to Solana and cover only Solana-specific issues. Regardless of technique, supported platforms, or inter-contract analysis, these approaches only address vulnerabilities within individual contracts and ignore off-chain components of cross-chain bridges. Hence, we consider these works orthogonal to our work.

Cross-Chain Bridge Security. Existing runtime monitoring approaches for cross-chain bridges are limited in scope and flexibility. XScope [3] can detect harmful transactions but lacks safeguards for operators to block them, relying on predefined patterns that limit adaptability to new bridges or ecosystems. It also fails against subtle attack variations, as it cannot capture semantic links between transaction patterns and malicious intent. Addressing this limitation leads to strict patterns with false alarms or broad ones with false negatives. XScope collects transaction data across blockchains to detect attacks through predetermined properties and patterns. This approach is inherently limited, as the security properties are rigid and require formal specification. Furthermore, the lack of adaptability of these properties means that new attack schemes cannot be detected, and XScopes rigid design hampers operators from incorporating new rules into the system. Hephaestus [33] flags suspicious transactions as outliers but assumes a complete bridge transaction model, nearly impossible in practice. Complementarily, XGuard [20] and SmartAxe [32] apply static analysis to router contracts. While this can rule out certain contract attacks, end-to-end bridge security lies beyond static analysis. Further, both are limited to Solidity contracts—XGuard built on Slither, SmartAxe on *GigaHorse*—constraining them to EVM blockchains and excluding architectures like Solana.

One major milestone in cross-chain bridge security is the development of zero-knowledge proofs (ZKPs) [71–73]. For instance, ZKPs have been applied to bridge contracts [71–73] to ensure that only authorized transactions are processed. Yet, ZKPs often require a trusted setup [73] and need complex calculations to build the proofs which is costly on-chain [71].

8 Conclusion and Summary

Cross-chain bridges play a pivotal role in connecting blockchain ecosystems for decentralized finance. BRIGADE is the first framework for detecting and mitigating cross-chain bridge attacks on popular blockchains like Ethereum, Solana, and

BSC. We introduce universal policies to detect malicious transactions targeting cross-chain vulnerabilities. In addition, a key achievement of our work is to model the working of bridges using the protocol analysis tool Tamarin, allowing formal verification of universal policies. Given its high performance and effectiveness against attacks, deploying BRIGADE in cross-chain bridges significantly increases trust in cross-chain bridges and DeFi applications.

Acknowledgements. This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—SFB 1119 (CROSSING) 236615297 within project T1 and S2, EXC 2092 (CASA) 39078197—, the Heinz Nixdorf Foundation through a Heinz Nixdorf Research Group (HN-RG), and the European Union (ERC, CONSEC, No. 101042266, and Horizon 2020 R&I, DYNABIC, No. 101070455). The views and opinions expressed are those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

A Appendix Proof of Concept Evaluation

Inconsistent Deposit Logic. In Listing 1.5 the policy to mitigate inconsistent deposit logic for ERC20 tokens is shown. The logic can be simplified to work with native currencies, e.g., ETH by getting the payer's balance instead of the token balance.

```
"precondition": {
    "balance_after": "call(ethereum, call_method, [$calldata, $token_address,
     $ethereum_block_number]).get(result)",
    "balance_before": "call(ethereum, call_method, [$calldata, $token_address, (
     $ethereum_block_number - 1)]).get(result)",
    "self_after": "call(ethereum, call_method, [$self_call, $token_address,
     $ethereum_block_number]).get(result)",
    "self_before": "call(ethereum, call_method, [$self_call, $token_address, (
     $ethereum_block_number - 1)]).get(result)"},
"policy": ["$balance_before - $token_amount == $balance_after",
    "$self_before + $token_amount == $self_after"]
```

Listing 1.5: An excerpt of preconditions and policies to prevent an Inconsistent Deposit bug for token assets.

Incomplete Verification. Checking for the incomplete verification bug needs two different policies. The first policy (Listing 1.6) checks for proof creations and stores these in a `$map`. The second policy (Listing 1.7) compares the msg.sender with the proof's actual recipient. Calling the function `withdraw_with_proof()` with the proof and the correct destination address and amount but, from the wrong message sender, results in a denial of the transaction.

Privilege Escalation. The policy defined in Listing 1.8 first assigns a new

```
"event": "ProofCreated(bytes32,address)", "chain_name": "ethereum",
"precondition": {
    "receiver": "call(ethereum, get_transaction_by_hash, [$transaction_hash]).get(result).
     get(input).slice(34, 74).as(address)",
    "calldata": "0x89ee...0000.push($payer_address.slice(2,42))",
    "proof": "call(ethereum, call_method, [$calldata, $poc2_contract,
     $ethereum_block_number.as(hex)]).get(result)"},
"policy": ["$map.insert($receiver, $proof)"]
```

Listing 1.6: An excerpt of a description file executed upon receiving the ProofCreated() event.

```
"event": "IncompleteVerification(uint256,bytes32)", "chain_name": "ethereum",
"precondition": { "input_data": "call(ethereum, get_transaction_by_hash,
    [$transaction_hash]).get(result).get(input)",
    "receiver": "$input_data.slice(162, 202).as(address)",
    "proof": "$input_data.slice(10, 74).as(hex)"},
"policy": [ "$receiver == $payer_address",
    "require($map.get($receiver) == $proof, $map.remove($receiver))"]
```

Listing 1.7: In this description file, we check whether the correct proof was provided by the correct msg.sender

variable that references the bridge's router contract, and then, checks whether the cross-function target is not the router contract. Note that there are other ways to specify the uncallable contracts ranging from filling the inbuilt keystore or putting the router in the inbuilt map.

B Custom Functions

BRIGADE is capable of implementing additional functions within the precondition dictionary at runtime. The namespace in which a function is valid is indicated by placing it in a subdirectory of the functions' folder. Every function consists of a JSON object that contains an RPC. Listing 1.9 displays the custom function used to invoke Ethereum smart contracts with arbitrary call data. The *$* symbol denotes a changeable parameter. In this scenario, `$param` substitutes the call data, `$to` replaces the called contract, and `$blocknumber` is the most recent block. Calling a custom function uses the function `call(connection, function, [args])`. For example, `call(ethereum, eth_call, [calldata, contract_address, latest])` calls the function in Listing 1.9. The first parameter is the endpoint where BRIGADE executes the function. Specifying different connections enables BRIGADE to call functions from arbitrary HTTP endpoints, like blockchains or off-chain

```
"event": "CrossFunctionCall(address)", "chain_name": "ethereum",
"precondition": { "uncallable_account": "0xa58A...1763"},
    "target": "$event_data.slice(2,66).as(address).toLower()"
"policy": ["$target != $uncallable_account.toLower()"]
```

Listing 1.8: A description file to check if cross-chain functions invoke the right contract

```
"method": "eth_call",
"params":[{"data":"$param","to":"$bridge_address"}, "$block_number"],
```

Listing 1.9: Custom Functions are RPCs with user-defined parameters.

components, to retrieve data from any source. The second parameter denotes the called function. Thereafter, an array contains the arguments to the function. Note that Ethereum supports the use of `latest` as a block number rather than the actual block number. When one places a $ sign in front of a value, it becomes a parameter for the function. Actual values replace the parameters in the description file. The function call yields a JSON map as the RPC response. The `get(field)` function retrieves the specified field from the map. For example, to obtain values in the *balance* field from the RPC response, we use `get(result).get(balance)`. Custom Functions are applicable in both the preconditions and the policies field.

C Full Reports on Real-World Attacks

In this section we provide the full details about the real-world Attacks.

Inconsistent Deposit Logic. We analyze four *inconsistent deposit logic* bugs. Qubit's *QBridge* lost 80 million USD due to an *inconsistent deposit logic* issue [17]. The `deposit` function failed to burn ETH, allowing an attacker to trigger a deposit event without actual token transfer. We apply the inconsistent deposit logic policies, adjusting preconditions to retrieve the token address from a resource ID. Calling `balanceOf` on `0x00...00` returns an empty result, failing the policy and denying the exploit.

The *inconsistent deposit logic* vulnerability in *Meter.io* resulted in a loss of 4.3 million USD [15]. Attackers exploited the bridge's assumption that wrapped native tokens are already unwrapped by the bridge handler. However, using `0x00...00` as the token address causes the unwrapping to fail, but the deposit still succeeds. BRIGADE's universal inconsistent deposit logic policy mitigates this attack while still processing valid transactions.

An auditor found a bug worth 3 million USD in the AnySwap *Multichain* bridge [60, 61]. Multichain wraps any ERC20 token into AnySwap tokens. When users transfer the token through the bridge, the AnySwap token receives the balance of the sender and the sender receives a new token on the destination chain. However, users that approved the bridge to take assets from a token contract were vulnerable. By calling the `anySwapOutUnderlyingWithPermit` function, an attacker stole their balances. To transfer the underlying token into the bridge, the function calls the `permit` function in the token contract. The exploited token lacked a permit function and triggered a fallback, continuing the execution and transferring approved amounts to the attacker. BRIGADE mitigates this attack using a universal policy for token deposits, with preconditions adjusted to the AnySwap token receiving the balance instead of the router contract.

The blockchain ThorChain was exploited for 8 million USD [65]. The attacker deployed a contract which executed a deposit with zero deposition value. However,

the contract itself was invoked with a value. The off-chain component of ThorChain falsely picked up the root transactions value instead of the deposition value. BRIGADE finds the exploit transactions.

Incomplete Verification. We analyze four bridges that were exploited through an *incomplete verification* bug. Additionally, we analyze 3 incomplete verification bugs where the transactions were signed with compromised private keys. The Ronin Bridge [13] ($600M) and Harmony Horizon ($100M) were exploited through compromised private keys. BRIGADE detects both exploits as the corresponding source chain transactions were missing.

The *BNB Bridge* suffered from an *incomplete verification* error, which allowed attackers to exploit a flaw in the Merkle proof verification and steal 586 million USD. BRIGADE's incomplete verification policy (see Section 5.2) detects this attack. To mitigate the attack, the router contracts are instrumented to stop transactions that BRIGADE finds to be malicious. For BNB Bridge, the sequence number serves as identifier, due to the lack of a receiver in the sending transaction.

The *Wormhole* bridge lost 326 million USD on Solana due to an *incomplete verification* bug. Solana programs require all context for execution in each transaction. Wormhole expected a *keccak256* proof and a Solana system program address but failed to verify the supplied program's validity. An attacker exploited this by deploying a program approving any proof, allowing unauthorized minting of wrapped Ether. BRIGADE's incomplete verification policy detects this by storing the correct proof at transaction initiation and retrieving it during withdrawal.

The *Nomad* bridge suffered from an *incomplete verification* bug, resulting in a 190 million USD loss, when a bridge operator set `committedRoot` to `0x00...00`, inadvertently validating all messages. Normally, Nomad uses the root hash to verify messages before processing transactions. However, now all messages bypassed verification. BRIGADE's universal incomplete verification policy detects this by matching corresponding deposit transactions.

The *ChainSwap* bridge suffered from an *incomplete verification* bug, resulting in an 8 million USD loss. The attacker exploited a flaw allowing quota reset for each new signatory, bypassing withdrawal verification by changing the signatory and nonce on each transaction. The incomplete verification policy marks the repeated withdrawals as malicious after the first successful transaction.

The Force Bridge ($4M) was exploited by unknown means [64] (likely a compromised private key). Force connects the Nervos blockchain with Ethereum and BSC. We connected the Nervos blockchain with BRIGADE to query for the associated source transactions. So, BRIGADE also detects the exploit.

Privilege Escalation. A *privilege escalation* in PolyNetwork allowed an attacker to exploit the router contract's execution logic, appointing any account as administrator. The attacker changed the amdinistrator in the *EthCrossChainData* contract, bypassing permission checks and withdrawing all funds from the bridge's router contract across all blockchains, causing losses of 611 million USD.

BRIGADE would have prevented this attack through its *address denylisting* policy, which protects designated high-privilege contracts. By storing the *EthCrossChainData* address as a protected key, BRIGADE blocks any transaction

```
"precondition": { "to_contract": "$event_data.slice(322, 362).as(address)",
    "protected": "0x07...bf.as(address)"},
"policy": ["$to_contract != $protected"]
```

Listing 1.10: Address allowlisting for the PolyNetwork bridge verifying that the target contract is not the protected contract.

attempting to alter its ownership while allowing legitimate cross-chain activity. Developers can refine or update the protected address as needed. Listing 1.10 shows the mitigating policy. Our experiment confirm that BRIGADE reports a policy violation when the protected contract is called, still allowing arbitrary cross-chain transactions not involving protected contracts.

References

[1] Yue Li et al. "POLYBRIDGE: A Crosschain Bridge For Heterogeneous Blockchains". In: *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2022, pp. 1–2. DOI: 10.1109/ICBC54727.2022.9805525.

[2] Sung-Shine Lee et al. "SoK: Not Quite Water Under the Bridge: Review of Cross-Chain Bridge Hacks". In: *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2023, pp. 1–14. DOI: 10.1109/ICBC56567.2023.10174993.

[3] Jiashuo Zhang et al. "Xscope: Hunting for Cross-Chain Bridge Attacks". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. Rochester, MI, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3551349.3559520.

[4] Li Duan et al. "Attacks Against Cross-Chain Systems and Defense Approaches: A Contemporary Survey". In: *IEEE/CAA Journal of Automatica Sinica* 10.8 (2023), pp. 1647–1667. DOI: 10.1109/JAS.2023.123642.

[5] Wei Ou et al. "An overview on cross-chain: Mechanism, platforms, challenges and advances". In: *Computer Networks* 218 (2022), p. 109378. ISSN: 1389-1286. DOI: doi.org/10.1016/j.comnet.2022.109378. URL: sciencedirect.com/science/article/pii/S1389128622004121.

[6] Rongjian Lan et al. *Horizon: A Gas-Efficient, Trustless Bridge for Cross-Chain Transactions*. 2021. arXiv: 2101.06000 [cs.CR].

[7] *Wormhole*. 2025. URL: wormhole.com/ (visited on July 10, 2025).

[8] *LayerZero*. 2025. URL: layerzero.network/ (visited on July 10, 2025).

[9] *Allbridge*. 2025. URL: allbridge.io/ (visited on July 10, 2025).

[10] *Nomad Protocol*. 2025. URL: nomad.xyz/ (visited on July 10, 2025).

[11] Elias Simos. *Bridge away (L1 Ethereum) - Dune*. 2021. URL: dune.com/eliasimos/Bridge-Away-(from-Ethereum) (visited on Apr. 10, 2025).

[12] RektHQ. *Poly Network*. 2021. URL: rekt.news/polynetwork-rekt/ (visited on Apr. 10, 2025).

[13] RektHQ. *Ronin*. 2022. URL: rekt.news/ronin-rekt/ (visited on Apr. 10, 2025).

[14] RektHQ. *BNB Bridge*. 2021. URL: rekt.news/bnb-bridge-rekt/ (visited on Apr. 10, 2025).

[15] RektHQ. *Meter*. 2022. URL: rekt.news/meter-rekt/ (visited on Apr. 10, 2025).

[16] RektHQ. *Wormhole*. 2022. URL: rekt.news/wormhole-rekt/ (visited on Apr. 10, 2025).

[17] RektHQ. *Qubit Finance*. 2022. URL: rekt.news/qubit-rekt/ (visited on Apr. 10, 2025).

[18] Chainalysis Team. *Cross-chain bridge hacks emerge as top security risk*. Aug. 2022. URL: blog.chainalysis.com/reports/cross-chain-bridge-hacks-2022/ (visited on Apr. 10, 2025).

[19] Julien Bouteloup. *Leaderboard*. 2023. URL: rekt.news/leaderboard/ (visited on Apr. 10, 2025).

[20] Ke Wang et al. "XGuard: Detecting Inconsistency Behaviors of Crosschain Bridges". In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, pp. 612–616. ISBN: 9798400706585. DOI: 10.1145/3663529.3663809. URL: dl.acm.org/doi/10.1145/3663529.3663809 (visited on Aug. 26, 2024).

[21] Asem Ghaleb et al. "AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 945–956. DOI: 10.1109/ICSE48619.2023.00087.

[22] Zhijie Zhong et al. "PrettySmart: Detecting permission re-delegation vulnerability for token behaviors in smart contracts". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24: IEEE/ACM 46th International Conference on Software Engineering (Lisbon Portugal). Vol. 54. ACM, Apr. 12, 2024. DOI: 10.1145/3597503.3639140. URL: http://dx.doi.org/10.1145/3597503.3639140.

[23] Clara Schneidewind et al. "eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 621–640.

[24] Petar Tsankov et al. "Securify: Practical Security Analysis of Smart Contracts". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 67–82.

[25] Siwei Cui et al. "VRust: Automated Vulnerability Detection for Solana Smart Contracts". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022.

[26] Sukrit Kalra et al. "ZEUS: Analyzing safety of smart contracts". In: *Proceedings 2018 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2018.

[27] Syed Badruddoja et al. "Making Smart Contracts Smarter". In: *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2021, pp. 1–3.

[28] Mark Mossberg et al. "Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 1186–1189. DOI: 10.1109/ASE.2019.00133.

[29] Michael Rodler et al. "Sereum: Protecting existing smart contracts against re-entrancy attacks". In: *Proceedings 2019 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2019.

[30] Michael Rodler et al. "EVMPatch: Timely and Automated Patching of Ethereum Smart Contracts". In: *USENIX Security Symposium*. 2021, pp. 1289–1306.

[31] Merkle Science. *Hack Track: An Analysis of Poly Network Hack and Latest Related Events*. Aug. 12, 2021. URL: merklescience.com/blog/hack-track-an-analysis-of-poly-network-hack-and-latest-related-events (visited on July 15, 2025).

[32] Zeqin Liao et al. "SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis". In: *Proc. ACM Softw. Eng.* 1.FSE (2024), 12:249–12:270. DOI: 10.1145/3643738. URL: dl.acm.org/doi/10.1145/3643738 (visited on Aug. 26, 2024).

[33] Rafael Belchior et al. "Hephaestus: Modeling, Analysis, and Performance Evaluation of Cross-Chain Transactions". In: *IEEE Transactions on Reliability* (2023). Conference Name: IEEE Transactions on Reliability, pp. 1–15. ISSN: 1558-1721. DOI: 10.1109/TR.2023.3336246. (Visited on Apr. 4, 2024).

[34] Ningran Li et al. "Blockchain Cross-Chain Bridge Security: Challenges, Solutions, and Future Outlook". In: *Distrib. Ledger Technol.* 4.1 (Feb. 2025). URL: doi.org/10.1145/3696429.

[35] Deepa Mishra et al. "Blockchain Security in Focus: A Comprehensive Investigation Into Threats, Smart Contract Security, Cross-Chain Bridges, and Vulnerabilities Detection Tools and Techniques". In: *IEEE Access* 13 (2025), pp. 60643–60671. DOI: 10.1109/ACCESS.2025.3556499.

[36] Mengya Zhang et al. "Security of Cross-chain Bridges: Attack Surfaces, Defenses, and Open Problems". In: *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '24. Padua, Italy: Association for Computing Machinery, 2024, pp. 298–316. ISBN: 9798400709593. DOI: 10.1145/3678890.3678894. URL: doi.org/10.1145/3678890.3678894.

[37] Simon Meier et al. "The TAMARIN prover for the symbolic analysis of security protocols". In: *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer. 2013, pp. 696–701.

[38] Ethereum Foundation. *Introduction to blockchain bridges*. ethereum.org/en/bridges/. 2022.

[39] Pascal Akunne. *Blockchain bridges: Guide to cross-chain data sharing*. blog.logrocket.com/blockchain-bridges-cross-chain-data-sharing-guide/. 2022.

[40] Maurice Herlihy. "Atomic Cross-Chain Swaps". In: *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. PODC '18. Association for Computing Machinery, 2018, pp. 245–254. ISBN: 9781450357951. DOI: 10.1145/3212734.3212736.

[41] Mengya Zhang et al. *SoK: Security of Cross-chain Bridges: Attack Surfaces, Defenses, and Open Problems*. 2023. arXiv: 2312.12573 [cs.CR].

[42] Josselin Feist et al. "Slither: A Static Analysis Framework for Smart Contracts". In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 2019, pp. 8–15.

[43] Wuqi Zhang et al. "Nyx: Detecting Exploitable Front-Running Vulnerabilities in Smart Contracts". In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024, pp. 2198–2216. DOI: 10.1109/SP54263.2024.00146.

[44] Yuheng Zhang et al. "FRAD: Front-Running Attacks Detection on Ethereum Using Ternary Classification Model". In: *Ubiquitous Security*. Ed. by Guojun Wang et al. Singapore: Springer Nature Singapore, 2024, pp. 63–75. ISBN: 978-981-97-1274-8.

[45] Jens-Rene Giesen et al. "HCC: A Language-Independent Hardening Contract Compiler for Smart Contracts". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2025, pp. 123–152.

[46] BlockSec. *LI.FI Attack: a Cross-chain Bridge Vulnerability? No, It's Due to Unchecked External Call!* blocksecteam.medium.com/li-fi-attack-a-cross-chain-bridge-vulnerability-no-its-due-to-unchecked-external-call-c31e7dadf60f. 2022. (Visited on Jan. 9, 2025).

[47] Amanda Davenport et al. "Air Gapped Wallet Schemes and Private Key Leakage in Permissioned Blockchain Platforms". In: *2019 IEEE International Conference on Blockchain (Blockchain)*. 2019, pp. 541–545. DOI: 10.1109/Blockchain.2019.00004.

[48] TeraBlock. *The ChainSwap Exploit Explained: A Detailed Overview and Next Steps*. myterablock.medium.com/the-chainswap-exploit-explained-a-detailed-overview-and-next-steps-84a70327278c. 2021. (Visited on Jan. 9, 2025).

[49] Shaun Azzopardi et al. "Monitoring Smart Contracts: ContractLarva and Open Challenges Beyond". In: *Runtime Verification*. Ed. by Christian Colombo et al. Cham: Springer International Publishing, 2018, pp. 113–137. ISBN: 978-3-030-03769-7.

[50] Sven Smolka et al. "Fuzz on the Beach: Fuzzing Solana Smart Contracts". In: *Proc. of the ACM Conference on Computer and Communications Security (CCS)*. Kopenhagen, Denmark, 2023.

[51] Michael Rodler et al. "EF/CF: High Performance Smart Contract Fuzzing for Exploit Generation". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2023.

[52] Gustavo Grieco et al. "Echidna: effective, usable, and fast fuzzing for smart contracts". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 557–560.

[53] Jingxuan He et al. "Learning to Fuzz from Symbolic Execution with Application to Smart Contracts". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS. ACM, 2019. DOI: 10.1145/3319535.3363230.

[54] Luca Olivieri et al. "Cross-chain Smart Contracts and dApps Verification by Static Analysis: Limits and Challenges". In: *CEUR Workshop Proceedings*. Vol. 3962. CEUR-WS. 2025.

[55] *reqwest*. 2025. URL: crates.io/crates/reqwest (visited on July 17, 2025).

[56] *WS-RS*. 2025. URL: crates.io/crates/ws (visited on July 17, 2025).

[57] *ethnum*. 2025. URL: crates.io/crates/ethnum (visited on July 17, 2025).

[58] *base58*. 2025. URL: crates.io/crates/base58 (visited on July 17, 2025).

[59] Benedikt Schmidt et al. "Automated analysis of Diffie-Hellman protocols and advanced security properties". In: *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE. 2012, pp. 78–94.

[60] Neptune Mutual. *Taking a Closer Look At Multichain Exploit*. neptunemutual.com/blog/taking-a-closer-look-at-multichain-exploit/. 2023. (Visited on Jan. 19, 2025).

[61] Rob Behnke. *Explained: The Multichain Hack (January 2022)*. halborn.com/blog/post/explained-the-multichain-hack-january-2022. 2022. (Visited on Jan. 19, 2025).

[62] Julienne Bouteloup. *Nomad Bridge - REKT*. July 1, 2022. URL: rekt.news/nomad-rekt (visited on July 18, 2025).

[63] Elliptic Enterprises Limited. *The Harmony Horizon Bridge Hack*. Elliptic. URL: elliptic.co/resources/harmony-horizon-bridge-hack (visited on Dec. 11, 2025).

[64] Rob Behnke. *Explained: The Force Bridge Hack (June 2025)*. Halborn. Oct. 6, 2025. URL: halborn.com/blog/post/explained-the-force-bridge-hack-june-2025 (visited on Dec. 11, 2025).

[65] Rob Behnke. "Explained: The THORChain Hack (July 2021)". In: (July 20, 2021). URL: halborn.com/blog/post/explained-the-thorchain-hack-july-2021 (visited on Dec. 11, 2025).

[66] Zhuo Zhang et al. "Your Exploit is Mine: Instantly Synthesizing Counterattack Smart Contract". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, 2023, pp. 1757–1774. URL: usenix.org/conference/usenixsecurity23/presentation/zhang-zhuo-exploit.

[67] Christof Ferreira Torres et al. "ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts". In: *IEEE European Symposium on Security and Privacy*. EuroS&P. IEEE, 2021. DOI: 10.1109/EuroSP51992.2021.00018.

[68] Mingxi Ye et al. "Midas: Mining Profitable Exploits in On-Chain Smart Contracts via Feedback-Driven Fuzzing and Differential Analysis". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, pp. 794–805. DOI: 10.1145/3650212.3680321.

[69] Chaofan Shou et al. "ItyFuzz: Snapshot-Based Fuzzer for Smart Contract". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 322–333. DOI: 10.1145/3597926.3598059.

[70] Christof Ferreira Torres et al. "ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20. Taipei, Taiwan: Association for Computing Machinery, 2020, pp. 584–597. DOI: 10.1145/3320269.3384756.

[71] Tiancheng Xie et al. "zkBridge: Trustless Cross-chain Bridges Made Practical". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 3003–3017. ISBN: 9781450394505. DOI: 10.1145/3548606.3560652. URL: doi.org/10.1145/3548606.3560652.

[72] Xiaoxuan Hu et al. "Optimized Cross-Chain Transactions With Aggregated Zero-Knowledge Proofs: Enhancing Efficiency and Security". In: *IEEE Internet of Things Journal* 12.9 (2025), pp. 11495–11510. DOI: 10.1109/JIOT.2024.3516036.

[73] Zahary Karadjov Rafael Belchior Dimo Dimov. *Harmonia: Securing Cross-Chain Applications Using Zero-Knowledge Proofs*. Tech. rep. 2025. DOI: 10.36227/techrxiv.170327806.66007684. URL: doi.org/10.36227/techrxiv.170327806.66007684/v4.