

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The Perils of Flexibility: Uncovering Application-Layer Vulnerabilities in Cosmos SDK Customization Points

Pengxiang Ma and Ningyu He, *The Hong Kong Polytechnic University;* Zhongchun Cao, *ChainFlag;* Zihao Li, *The Hong Kong Polytechnic University and University of Electronic Science and Technology of China;* Xiapu Luo, *The Hong Kong Polytechnic University*

https://www.usenix.org/conference/usenixsecurity26/presentation/ma

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

The Perils of Flexibility: Uncovering Application-Layer Vulnerabilities in Cosmos SDK Customization Points

Pengxiang Ma[1] Ningyu He[1] Zhongchun Cao[2] Zihao Li[1,3] Xiapu Luo[1,*]
[1]*The Hong Kong Polytechnic University*
[2]*ChainFlag*
[3]*University of Electronic Science and Technology of China*

Abstract

The Cosmos SDK is a premier blockchain development framework that facilitates the construction of application-layer logic through reusable modules and extensible customization interfaces. Currently, the Cosmos ecosystem encompasses over one hundred production blockchains, reaching a peak total market capitalization exceeding $32 billion. While this modular architecture significantly lowers development barriers, it simultaneously expands the application-layer attack surface, as developer-accessible customization points are susceptible to misuse or incorrect composition.

In this work, we present a security analysis of application-layer pitfalls within Cosmos-based blockchains. By analyzing the transaction execution lifecycle and developer-accessible customization points, we formalize four execution invariants essential for correct transaction processing. Guided by these invariants, we identify eight recurring classes of error-prone development patterns in which violations compromise transaction safety, resulting in severe consequences ranging from asset theft to chain-halting failures. To assess the prevalence and impact of these pitfalls, we conducted an ecosystem-wide empirical analysis, uncovering 65 vulnerabilities across various Cosmos-based blockchains. Following responsible disclosure, 26 of these vulnerabilities were acknowledged by the affected project teams, with 19 subsequently receiving public fixes and two resulting in assigned CVE identifiers. These disclosures yielded $51,000 in bug bounty rewards and prevented potential asset theft affecting assets valued in the tens of millions of dollars. Our findings demonstrate that application-layer vulnerabilities are widespread in the Cosmos ecosystem and underscore the critical need for increased rigor in application-layer design, development, and auditing.

1 Introduction

The Cosmos ecosystem has emerged as a prominent framework for building application-specific blockchains, enabling developers to design sovereign chains with customized application logic. A wide range of production systems are built atop the Cosmos SDK, including Cosmos Hub [16] (inter-chain coordination), Osmosis [35] (DEX), Sei [36] (high-performance EVM compatibility), Secret Network [48] (TEE-based privacy), and Evmos [22] and Injective [34] (EVM compatibility). Together, these systems support diverse domains (DeFi, privacy-preserving computation, EVM-based contracts), and at their peak, they collectively reached a market capitalization of over $32 billion [12].

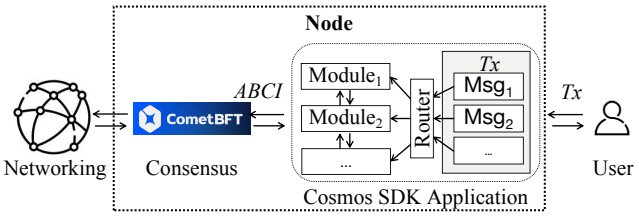


Figure 1: The architecture of a Cosmos-based blockchain.

Unlike smart-contract-centric platforms, the Cosmos SDK adopts a fundamentally different development model. Instead of expressing application logic in a domain-specific smart contract language, the Cosmos SDK allows developers to implement blockchain logic directly in a general-purpose programming language (Golang) by composing reusable modules. Fig. 1 illustrates the architecture of a Cosmos-based blockchain. End users submit transactions to nodes through wallets or RPC endpoints. Each node consists of two logically separated components: the *consensus layer* and the *application layer*. The consensus layer is implemented by Tendermint [2], now maintained as CometBFT, which is responsible for peer-to-peer networking, transaction dissemination, and deterministic transaction ordering. The application layer is implemented using the Cosmos SDK, which interprets ordered transactions and applies state transitions. These two components communicate through the ABCI interface [20], ensuring

*Corresponding author.

that all honest nodes execute transactions in the same order and reach consensus on the resulting state. The transaction may contain one or more messages, which are dispatched by the message router to corresponding module handlers, where handlers may further interact with other modules through synchronous keeper calls or internal message routing (see §2.4). While the SDK enforces a fixed transaction processing pipeline, it exposes several powerful *customization points*.

CometBFT (formerly Tendermint) consensus has been extensively studied, with prior work analyzing its safety and liveness guarantees [37, 50, 52]. These studies primarily focus on the correctness of the consensus protocol itself and typically assume that the application layer behaves as expected. Although the SDK is designed to preserve global application guarantees, subtle misuses of application semantics can still introduce security risks. Existing work has begun to explore such risks mainly from the perspective of non-determinism, leveraging static and information-flow analysis to detect non-deterministic behaviors in Cosmos SDK applications [45, 49]. However, these approaches focus narrowly on non-determinism and do not capture a broader class of application-layer pitfalls. More broadly, application-layer vulnerabilities have been shown to cause severe security incidents across blockchain systems. For example, prior research on non-Cosmos-based platforms, such as Ethereum [3], has extensively studied smart contract vulnerabilities, revealing critical issues including reentrancy, integer overflows, and logic bugs [4, 23, 53]. Beyond smart contracts, several studies have analyzed gas-related issues in blockchain application layers, such as gas-based denial-of-service attacks and inconsistent gas-cost accounting [29, 33, 40]. The Cosmos ecosystem is no exception. Several incidents demonstrate that flaws in the application layer can have a severe, critical impact. For example, the Juno network suffered a denial-of-service attack that halted the chain due to transactions triggering a non-deterministic error [11]. More recently, a vulnerability exploited on Saga in January 2026 resulted in losses exceeding $7 million [47]. In addition, industrial security reviews and community efforts have documented recurring vulnerability patterns in Cosmos SDK applications [5, 26, 44]. Despite these industrial efforts, new application-layer vulnerability patterns continue to emerge and propagate widely. In particular, the lack of cohesive security guidance has resulted in development pitfalls being embedded in widely referenced tutorials and code, which are subsequently reused across many Cosmos-based projects.

This Work. We take the first step toward a systematic security analysis of application-layer pitfalls in Cosmos-based chains by conducting an in-depth analysis of the Cosmos transaction execution pipeline. We first distill the transaction end-to-end lifecycle and identify key *customization points* exposed to application developers (see §2). We then summarize the adversary model and a set of fundamental execution invariants (see §3). Against these invariants, we identified eight types of *development pitfalls*. For each pitfall, we propose a detection method, construct proof-of-concept (PoC) exploits to empirically validate exploitability, and present concrete mitigation strategies (see §4). Beyond individual vulnerabilities, we further analyze how these pitfalls propagate across the ecosystem through code reuse, module composition, and developer tutorials. Our empirical analysis reveals 65 vulnerabilities across real-world Cosmos-based blockchains. In particular, we identify arbitrary-asset-stealing vulnerabilities in *Evmos* and *Secret Network*, as well as denial-of-service vulnerabilities in *Mantra*, *Injective*, and other production chains (see §5). After responsible disclosure to affected projects, a total of 19 vulnerabilities have been patched, and our disclosures resulted in bug bounty rewards totaling over $51,000.

Our contributions can be summarized as follows:

- We present an analysis of application-layer security risks in Cosmos-based chains.
- We characterize eight Cosmos development pitfalls, six of which are first described as security issues, as we know.
- We conduct a large-scale empirical analysis and uncover 65 real-world vulnerabilities across Cosmos-based blockchains, with 26 acknowledged and 19 publicly fixed.
- Our results show significant real-world security impact, including asset theft worth tens of millions of dollars and chain halts on multiple production projects.

2 Background

2.1 Blockchain

Blockchain is a distributed system that maintains a replicated and append-only state among a set of mutually untrusted nodes. Instead of relying on a single trusted database, blockchain nodes collectively maintain a shared ledger by validating transactions and agreeing on a single canonical ordering through a consensus protocol. This ordering is crucial because the effects of a transaction depend on the current state (e.g., balances and nonces), and different orderings may yield different outcomes. State transitions are driven by transactions, which are disseminated over the peer-to-peer network, checked against protocol rules, and then executed by all honest participants in the agreed order [43]. From an execution perspective, a blockchain functions as a deterministic replicated state machine in which each transaction encodes a sequence of operations applied to the current state, and every honest node must obtain the same post-state and the same externally visible outputs (e.g., events and receipts). This determinism requirement makes the execution layer a critical component of consensus, since any divergence in execution can lead to state inconsistency, chain forks, or even a safety violation.

2.2 Cosmos SDK

The Cosmos SDK is a modular framework for building application-specific blockchains [17]. The Cosmos SDK allows developers to implement blockchain application logic directly in native code by composing or customizing reusable modules. Each Cosmos-based blockchain, therefore, could define its own logic while relying on shared infrastructure for consensus and transaction delivery. As illustrated in Fig. 1, a Cosmos-based blockchain separates consensus and application logic into two parts. CometBFT, the consensus engine, is responsible for network communication and transaction ordering, while the Cosmos SDK, the application layer, interprets ordered transactions and performs state transitions. These two components communicate through a standardized interface, *i.e.,* ABCI, ensuring that transaction execution remains deterministic across all nodes [14, 20].

Application logic in the Cosmos SDK is organized into *modules*, each encapsulating a specific functional domain and its corresponding state. These modules are designed to interact with one another (see §2.4). Moreover, transactions are composed of one or more *messages*, which are dispatched to the appropriate modules for execution. The framework enforces a fixed transaction-processing structure while exposing well-defined customization points for developers to implement application-specific behavior [17].

Cosmos SDK application logic is implemented by composing modules, each responsible for a distinct state domain and a set of message handlers. For example, the `bank` module maintains account balances and implements token transfers (e.g., `SendCoins`) that are widely used by other modules and application code. The `auth` and `authz` modules manage accounts, signature verification, and authorization, while `staking`, `distribution`, and `gov` implement core governance workflows. Many production chains further integrate the `x/wasm` module [18], which embeds the CosmWasm virtual machine and enables smart contract execution as part of the same transaction pipeline. A CosmWasm contract is uploaded as bytecode, instantiated with an initial state, and then invoked through contract messages that are routed to `x/wasm`. During execution, the contract can read/write its own storage, emit events, and call into other modules' message handlers.

2.3 Transaction Execution Pipeline

In the Cosmos SDK, application state advances through the ordered execution of transactions. A transaction may contain one or more messages (`sdk.Msg`), which are executed sequentially as a single atomic unit. Each transaction is executed within a dedicated execution Context, denoted as `Ctx`, which encapsulates all transaction-scoped execution state throughout the pipeline. A `Ctx` consists of three key components:

- **Storage.** The `Ctx` provides access to application state via `MultiStore`. During transaction execution, this storage is typically wrapped in a cached layer, ensuring that state updates are applied atomically at the commit boundary.
- **Gas.** The `Ctx` carries a `GasMeter` that tracks and enforces gas consumption. Each `GasMeter` consists of two fields, `consumed` and `limit`, which record the gas used so far and the maximum allowable gas, respectively, ensuring that transaction execution does not exceed its declared limit at any stage of the transaction lifecycle. In addition, the gas consumed by transactions is later charged against a block-level `GasMeter` during deferred gas accounting, ensuring that the block maximum gas limit is also not exceeded.
- **Event.** The `Ctx` maintains an `EventManager` for recording events emitted during transaction execution. These events are buffered within the `Ctx` and are finalized only if the state change is accepted. Thus, `EventManager` shares the same commit boundary with `Storage`.

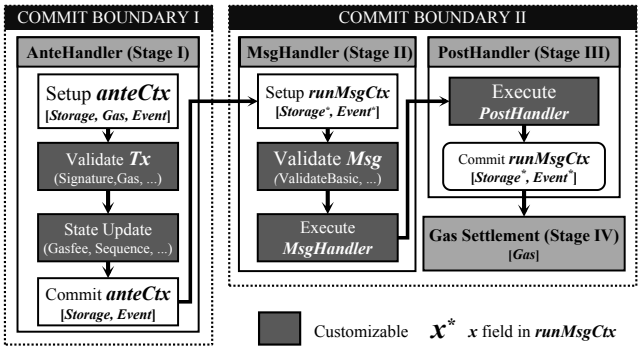


Figure 2: Transaction execution pipeline

Transaction execution follows a fixed pipeline, which could be divided into four stages, i.e., *AnteHandler*, *MsgHandler*, *PostHandler*, and *Gas Settlement*, as illustrated in Fig. 2.

Stage I (AnteHandler) Stage I performs transaction admission control before any message logic runs. The term *ante* indicates that this stage runs *before* message execution. As illustrated in Fig. 2, the SDK derives an ante-specific cached context, denoted as `anteCtx`, from global `Ctx` with cached `Storage` and an ante-scoped `EventManager`. It also installs a transaction-wide `GasMeter`. The `GasMeter` uses the declared transaction gas limit, and the SDK rejects transactions that exceed the configured maximum, which is the block gas limit. This `GasMeter` is also inherited by the subsequent execution `Ctx`. As a result, all transaction execution is bounded by the gas limit configured by the *AnteHandler*. The *AnteHandler* then executes a fixed sequence of steps, including basic field validation, signature verification, fee deduction, and account sequence increment. If all steps succeed, the SDK commits `anteCtx` and persists the state updates and the ante events. Otherwise, all cached effects are discarded. Thus, *Stage I* forms a *commit boundary*.

Stage II (MsgHandler). Stage II executes transaction messages and produces application state transitions. The SDK creates a new cached execution `Ctx` for message handling, while continuing to use the same transaction-wide `GasMeter` installed in *Stage I*. This cached `Ctx` provides a cached `Storage` and installs a fresh `EventManager`. All messages in the transaction are executed sequentially within this shared execution `Ctx`. For each message, the SDK locates the corresponding module's message handler and performs basic message validation, including `ValidateBasic` and the `circuitBreaker` check. Both the handler and the validation can be customized. The SDK executes the handler logic while charging gas against the transaction-wide `GasMeter`. All state mutations performed by message handlers are applied to the cached `Storage` and are not immediately committed. If any message handler returns an error or triggers a panic, execution is aborted, and all cached effects introduced during message execution are discarded.

Stage III (PostHandler). Stage III runs optional post-processing after all messages finish. In this stage, an optional *PostHandler* is invoked as a transaction-level hook on the cached execution `Ctx` produced by *Stage II*, as illustrated in Fig. 2. The *PostHandler* provides a customizable extension point for post-transaction execution logic (e.g., refunding unused gas). If the *PostHandler* returns an error, execution aborts and all cached state changes and events introduced during *Stage II* and *Stage III* are discarded. Otherwise, the SDK commits the cached execution `Ctx`, persists state updates, and emits the accumulated events. Thus, *Stage II* and *Stage III* collectively form a second commit boundary.

Stage IV (Gas Settlement). At this stage, gas consumption is finalized regardless of the transaction outcome. Specifically, gas settlement is handled by a `defer` block, which in Go is guaranteed to execute upon function return or during a `panic`. As a result, whether *Stage III* completes normally or a failure or `panic` occurs in *Stages I*, *II*, or *III*, control is always transferred to *Stage IV*. In *Stage IV*, the transaction's `GasMeter` is charged, and the consumed gas is accounted against the block-level `GasMeter`. Consequently, the cumulative gas consumption of all transactions in a block is strictly bounded by the configured maximum block gas limit.

2.4 Customization

Developers typically do not modify the transaction execution pipeline itself. Instead, the Cosmos SDK exposes well-defined *customization points* for injecting application-specific logic across execution stages. While *Stage II* message handlers serve as the primary extension mechanism, developers may also customize *Stage I* via the *AnteHandler* (e.g., overriding gas and signature semantics for EVM compatibility) and optionally extend *Stage III* using a *PostHandler* for post-execution processing, as shown in Fig. 2.

Listing 1: Example message handler

```
1  func (k Keeper) RequestLoan(
2      goCtx context.Context,
3      msg *types.MsgRequestLoan,
4  ) (*types.MsgRequestLoanResponse, error) {
5   ctx := sdk.UnwrapSDKContext(goCtx)
6  // Gas operation (Orange)
7   ctx.GasMeter().ConsumeGas(gasCostPerIteration,
8   " check msg")
9  // Inter-module call (Purple)
10  sdkError := k.bankKeeper.SendCoins(
11   ctx, borrower, types.ModuleName, collateral,
12  )
13 ...
14  sdkError = k.router.Handler(ctx, innerMsg)
15 // Storage operation (Blue)
16  store := prefix.NewStore(
17   ctx.KVStore(k.storeKey),
18   types.KeyPrefix(types.LoanKey),...)
19  store.Set(GetLoanIDBytes(loan.Id), appendedValue)
20 // Event operation (Green)
21  ctx.EventManager().EmitEvent(
22   sdk.NewEvent(sdk.EventTypeHello,
23   sdk.NewAttribute(
24   types.AttributeKeySender, from)))
25     return &types.MsgRequestLoanResponse{}, nil
26 }
```

As a representative example, Listing 1 shows a typical message handler implementation. As introduced in §2.3, all application logic executes under a shared execution `Ctx`. The `Ctx` encapsulates core execution resources, including gas, persistent state, and events. The SDK exposes a concise API surface over `Ctx`, enabling developers to read and write state, consume gas, emit events, and invoke logic across module boundaries, thereby allowing control flow to extend beyond a single module. Below, we discuss each category in turn.

Gas accounting. Gas is accounted through the `GasMeter` embedded in the `Ctx` (Listing 1 orange lines). Message handlers and keeper methods explicitly charge gas by invoking `ConsumeGas`, which increments the `consumed` field of the `GasMeter` and checks it against the `limit` field, aborting execution by triggering an out-of-gas `Panic` once the `limit` is exceeded. The `GasMeter` also exposes read-only queries such as `GasConsumed` and `GasRemaining`. In addition, `GasMeter` can be locally rebound via `WithGasMeter`, which installs a new `GasMeter` into a derived `Ctx` to track gas consumption for a specific execution scope. Gas consumed under the derived `Ctx` is charged to the new `GasMeter` and does not automatically propagate to the original `GasMeter`.

Persistent state. Persistent state is maintained in the underlying `MultiStore` and accessed through `KVStore` obtained from the `Ctx` (blue-highlighted region in Listing 1).

Read operations (e.g., `Get`) observe the current cached view of state, while write operations (e.g., `Set` and `Delete`) apply updates to a cache-backed store. These writes remain buffered during execution and are committed to the database only if execution reaches a commit point. To separate commit boundaries and support explicit commit or rollback of state changes, the Cosmos SDK enables Storage forking via `WithMultiStore`, which creates a derived `Ctx` backed by a new `MultiStore`. In practice, this functionality is most commonly accessed through the higher-level `CacheContext` API, which constructs a cached `MultiStore` and binds it to a derived `Ctx` using `WithMultiStore`. State updates in the derived `Ctx` are buffered and applied to the parent `MultiStore` only upon explicit commitment, allowing the SDK to isolate execution stages and enforce well-defined commit boundaries.

Event recording. Execution events are recorded through the `EventManager` (green-highlighted region in Listing 1). Handlers append events using `EmitEvent` or `EmitEvents`, while read-only access is provided via `Events`. Similar to gas and storage, the event scope can also be rebound by installing a fresh `EventManager` into a derived `Ctx` via `WithEventManager`, which is also used to separate *Stage I* events from *Stage II* events.

Cross-module interaction. Modules interact through two patterns (purple-highlighted region in Listing 1). In the first pattern, a module invokes another module's exported method directly (e.g., `otherKeeper.Method(ctx, ...)`). In the second pattern, a module could forward an `sdk.Msg` to the message router by invoking `Handler(ctx, msg)`. This handler resolves the target module based on the message type and dispatches execution to the corresponding message handler in *Stage II*. In both patterns, execution continues under the same `Ctx`, meaning that gas consumption, state change, and event recording are consistently propagated across inter-module interactions.

3 Adversary Model

In this section, we begin with a motivating example that illustrates how a seemingly simple implementation mistake can have severe consequences, and how such an error can further propagate across the Cosmos ecosystem through code reuse (see §3.1). We then define the threat model by characterizing the attacker's capabilities and objectives (see §3.2). Finally, based on our analysis of the Cosmos SDK transaction execution pipeline, we summarize the four invariants that govern the correct execution of transactions (see §3.3).

3.1 Motivating Example

Many Cosmos-based blockchains reuse code patterns from other projects and tutorials, thereby propagating subtle yet severe vulnerabilities. For example, Listing 2 shows an official tutorial where a cross-module call to `SendCoins` fails to check its returned `error`, allowing an attacker to acquire ownership without payment by invoking `BuyName()` from a zero-balance account. As this tutorial is widely copied, the same flaw has propagated into multiple production Cosmos projects.

Listing 2: A motivating example in official tutorial

```
1  func (k msgServer) BuyName(
2      goCtx context.Context,
3      msg *types.MsgBuyName,
4  ) (*types.MsgBuyNameResponse, error) {
5      ...
6  -   k.bankKeeper.SendCoins(ctx, buyer...)
7  +   err := k.bankKeeper.SendCoins(ctx, buyer...)
8  +   if err != nil {
9  +    return nil, err
10 +   }
11     k.SetWhois(ctx, types.Whois{Owner: buyer})
12     return &types.MsgBuyNameResponse{}, nil
13 }
```

This example highlights a core motivation of our work. While the Cosmos SDK provides strong structural guarantees, oversights can still lead to incorrect state transitions and concrete security risks, which can then be widely propagated to affect many downstream projects. The issue was first identified and reported by us to the Cosmos team in 2022 [10] and was subsequently fixed. We later added this issue to the industrial checklist [31] to alert other developers.

3.2 Threat Model

We consider a primary adversary, *malicious transaction senders*, who can freely construct and submit transactions to invoke vulnerable code through all externally accessible execution paths. Depending on the chain configuration, adversaries can also deploy arbitrary *CosmWasm* contracts [18]. Although several Cosmos-based blockchains restrict contract deployment through governance approval or whitelisting, we assume that adversaries can nevertheless deploy malicious contracts. This assumption is justified because governance- or whitelist-based code review does not constitute a reliable security barrier, as bytecode-level review cannot reliably rule out subtle vulnerabilities [41]. Within these capabilities, adversaries aim to exploit vulnerabilities at all customization points (see §2.4), including gas handling, event emission, storage updates, and cross-module call patterns.

3.3 Execution Invariants

Based on our analysis of the transaction execution pipeline in the Cosmos SDK (see §2.3), we summarize the key invariants across gas accounting, event emission, state storage, and the execution sequence. These invariants ensure that customization does not compromise global correctness properties, even

when application logic spans multiple messages, modules, or execution stages.

Gas Invariant (I_1). Transaction execution is subject to a bounded gas limit that must be consistently enforced throughout the entire lifecycle. All computations triggered during transaction processing must be charged against this limit. Exceeding the gas limit (both the block gas limit and the transaction gas limit) must deterministically abort execution, thereby preventing the DoS attack vector.

Event Invariant (I_2). Events emitted during transaction execution must be consistent with the committed state updates. There should be a faithful and complete log of the final state transition. Concretely, for each successful state mutation that becomes persistent, the corresponding events must be emitted exactly once (no missing or duplicated events), and the event order should match the execution order observed by the state machine. Conversely, if an execution path is reverted or aborted, any events produced along that path must not appear in the final transaction result. This invariant ensures that off-chain consumers (e.g., indexers and relayers) observe an execution history that is both correct and unambiguous.

Storage Invariant (I_3). Storage updates must satisfy *atomicity* at the commit boundary. Execution either commits all writes or rolls back all writes, with no partial effects. Violating this invariant can leave the chain in an inconsistent state. Since *Stage II* and *Stage III* share a commit boundary, an abort in *Stage III* must roll back from both stages.

Execution Sequence Invariant (I_4). Transaction execution must comply with the validation design imposed by both the framework and external modules, ensuring that no constraints can be bypassed. Concretely, the `AnteHandler` checks (e.g., signature verification and gas validation) must be enforced before any state mutation, and under cross-module interaction via direct calls to another module's exported methods (e.g., `otherKeeper.Method(ctx, ...)`), the callee's exported-method checks must also be respected.

4 Pitfall Development Patterns

Guided by the four execution invariants in §3.3, we distill eight recurring pitfall development patterns. These error-prone practices stem from ambiguous Cosmos SDK specifications and misunderstandings of the transaction pipeline, and they concretely violate these invariants. Table 1 links each violated invariant to its corresponding pitfall pattern and the source of our discovery. We derived seven risks from code auditing. To the best of our knowledge, except for $\mathcal{P}_8$ (derived from a known vulnerability report [32]) and $\mathcal{P}_1$ (for which we later identified a disclosed vulnerability report with the same root cause [19]), the remaining pitfalls have not been described in prior academic or industry reports, although we did observe scattered fixes for $\mathcal{P}_2$ and $\mathcal{P}_4$ issues in some codebases. Notably, $\mathcal{P}_6$ can be viewed as a Cosmos-specific security instance of generic unchecked error handling in Go. While such warnings are often ignored because many return values have negligible effects, we focus on Cosmos APIs whose failures are security-critical. For example, ignoring errors from asset-transfer APIs such as `SendCoins` (e.g., insufficient balance) can invalidate subsequent balance-update logic and threaten asset safety.

Table 1: Summary of broken execution invariants, pitfall pattern, and sources.

Invariant	Pitfall Pattern	Source
Gas (I_1)	$\mathcal{P}_1$	Auditing
	$\mathcal{P}_2$	Auditing†
Event (I_2)	$\mathcal{P}_3$	Auditing†
	$\mathcal{P}_4$	Auditing†
Storage (I_3)	$\mathcal{P}_5$	Auditing†
Execution Sequence (I_4)	$\mathcal{P}_6$	Auditing†
	$\mathcal{P}_7$	Auditing†
	$\mathcal{P}_8$	History Report

† pitfall risks first described in this work.

4.1 Gas Invariant Violation

Two pitfall patterns ($\mathcal{P}_1$ and $\mathcal{P}_2$) violate the I_1 gas invariant, allowing attackers to execute unbounded computation and thereby mount denial-of-service attacks.

4.1.1 Block Gas Accounting Loss ($\mathcal{P}_1$)

$\mathcal{P}_1$ arises when a module executes logic under a child `GasMeter` but fails to reliably propagate the child `consumed` amount back to the global `GasMeter`. Although the developer may explicitly merge the child `GasMeter` usage to parent `GasMeter`, exceptional control flow, such as an out-of-gas `Panic` (see §2.4), can interrupt execution prior to the explicit merge step, preventing the child `GasMeter` usage from being reflected in the parent `GasMeter` and thereby violating the I_1 gas invariant (see §3.3). Because block-level gas accounting in *Stage IV* is derived from the parent `GasMeter`, exploiting $\mathcal{P}_1$ allows blocks to incur actual execution costs beyond the block gas limit. In practice, such behaviors could compress block proposer and voting time windows, increasing the likelihood of missed proposals or votes, leading to liveness-related slashing and violations of protocol fairness.

Listing 3: Gas accounting loss from Non-deferred child `GasMeter` reconciliation.

```
childCtx:=ctx.WithGasMeter(NewGasMeter(limit))
+ defer ctx.GasMeter().ConsumeGas(defaultGasCost)
err := SubCall(childCtx)//cost default gas
- ctx.GasMeter().ConsumeGas(defaultGasCost)
```

Detection. First, we identify application logic that derives a new `Ctx` with a child `GasMeter` using `WithGasMeter` and later reconciles the child gas usage with the parent `GasMeter` through an explicit merge operation that is not guarded by a `defer`. Second, we validate the presence of the vulnerability by executing a proof-of-concept that triggers exceptional control flow and checking that the parent `GasMeter` fails to account for the gas actually consumed.

Mitigation. To preserve the I_1 gas invariant, reconciliation of child gas consumption must execute regardless of how the nested call terminates. A standard mitigation is to move the reconciliation logic into a `defer` block, ensuring that the canonical `GasMeter` is always charged. As shown in Listing 3, the vulnerable version (red) reconciles gas only on the normal return path, whereas the fixed version (green) ensures reconciliation is executed on all exit paths including panic.

4.1.2 Infinite Reentrancy Execution ($\mathcal{P}_2$)

Listing 4: Reentrant message handler in `SubCall` with fixed-limit gas rebinding.

```
func SubCall(..., msg sdk.MsgCall) error {
-   childMeter := sdk.NewGasMeter(500_000)
-   childCtx := ctx.WithGasMeter(childMeter)
+   gasLimit := min(GasRemaining(),500_000)
+   submeter := storetypes.NewGasMeter(gasLimit)
+   childCtx := ctx.WithGasMeter(submeter)
    err := k.router.Handler(childCtx, msg.SubMsg)
    if err != nil {
        return err
    }
    return nil
}
```

$\mathcal{P}_2$ arises when reentrancy repeatedly triggers `WithGasMeter` rebinding, allowing each reentrant invocation to execute under a freshly allocated child `GasMeter` with a fixed `limit` field allowance and thereby incrementally gain additional gas. As introduced in §2.4, the Cosmos SDK supports internal message forwarding via the message router, enabling recursive execution of the same type of `Msg`. When such a routing-based recursion occurs within a handler that rebinds a child `GasMeter`, gas consumed under child meters may not be consistently charged against the global `GasMeter` before sub-message handling. As shown in Listing 4, a handler installs a child `GasMeter` and synchronously forwards a sub-message back to `k.router.Handler(..)` within the same transaction, causing successive reentries to execute under new child meters while the global meter is never exhausted. By repeatedly routing through this handler, a single transaction can extend execution beyond the intended gas bound by borrowing gas across reentrant invocations. If such reentrant paths are attacker-controlled and repeatedly triggerable, execution is no longer bounded by the original gas limit, violating the I_1 gas invariant (see §3.3) and causing transaction execution to fail to terminate.

Detection. We identify $\mathcal{P}_2$ by detecting the co-occurrence of two features: (i) invocation of the `k.router.Handler(..)` interface, including APIs that wrap `k.router.Handler(..)` (e.g., `Wasm.Sudo`, a chain-side contract invocation interface that synchronously dispatches contract-returned messages via the router), and (ii) a fixed gas limit `WithGasMeter` rebinding. Furthermore, we validate this pattern through a proof-of-concept execution that triggers reentrant routing and assesses whether execution terminates normally. The PoC shows that, under fixed-limit gas rebinding, handler invocations can execute indefinitely, potentially leading to a chain halt.

Mitigation. To mitigate $\mathcal{P}_2$, a handler must not grant each reentrant invocation a fresh, fixed gas allowance. Instead, when a child `GasMeter` is needed, the child's gas limit should be bounded by the parent `GasMeter`'s remaining gas, so that reentrant routing cannot increase the total gas available within a transaction. Listing 4 shows an example fix that sets `gasLimit := min(parentRemaining, hookLimit)` before constructing the child `GasMeter`.

4.2 Event Invariant Violation

Two pitfall patterns ($\mathcal{P}_3$ and $\mathcal{P}_4$) can lead to violations of the I_2 event invariant, i.e., inconsistencies between emitted events and committed state transitions. Although they do not affect consensus, they constitute the primary interface through which off-chain systems observe execution results. Both $\mathcal{P}_3$ and $\mathcal{P}_4$ pitfalls introduce observable inconsistencies for off-chain consumers such as wallets, explorers, and bridges [56]. Missing events prevent state changes from being detected, while duplicated events may be treated as repeated actions.

4.2.1 Event Loss ($\mathcal{P}_3$)

Event loss occurs when user-defined customization points install a new `EventManager` in the current `Ctx` via `WithEventManager`, but the events emitted under this new manager are not merged back into the original `EventManager`. In such cases, events within the child `EventManager` are ultimately discarded. This form of rebinding, which operates below the intended lifecycle, violates the guarantee that all events produced along successful execution paths are preserved and observable. As a consequence, events generated during execution would be silently omitted from the final transaction or block results, even though the corresponding state transitions have been successfully committed.

Detection. We flag $\mathcal{P}_3$ when application logic initializes a fresh `EventManager` to `Ctx`, but the events emitted under this new manager are not propagated to the parent `EventManager` and thus do not appear in the final transaction result. For example, this can happen when a derived `Ctx` is created via

WithEventManager or wrapper APIs such as CacheContext and router.Handler(..), yet the child events are never merged back (e.g., via EmitEvent(s)). We validate suspected cases by executing the mock execution path, enumerating the expected event set for the triggered state transition, and verifying that the final result contains no such events.

Listing 5: Fixing event loss by propagating child events back to the parent EventManager.

```
childCtx := ctx.WithEventManager(sdk.
  NewEventManager())
...
- return nil
+ events := childCtx.EventManager().Events()
+ ctx.EventManager().EmitEvents(events)
+ return nil
```

Mitigation. To preserve the I_2 event invariant, developers should ensure that any events emitted under a derived EventManager are propagated to the parent context on all successful paths. As Listing 5 illustrates, the vulnerable version (red) installs a fresh EventManager but never merges its events back, whereas the fixed version (green) explicitly emits the child events into the parent manager before returning.

4.2.2 Duplicate Event Emission ($\mathcal{P}_4$)

This pitfall arises from semantic drift in the SDK's CacheContext implementation between versions 0.46.3 and 0.46.4. In earlier versions, events emitted under a cached context were not automatically propagated to the parent EventManager, requiring developers to manually merge events upon calling writeCache. Starting from 0.46.4, the SDK guarantees that events produced in the CacheContext are automatically emitted to the parent Ctx when the cache is written (see Listing 6). When application code written under the older assumption is executed against newer SDK versions without being updated, events are merged twice, once implicitly by the SDK and once explicitly by the application. As a result, a single logical state transition may produce duplicated event records. External systems that interpret events as triggers, such as exchanges or indexers, may process these duplicates as repeated actions, for example, treating a single token transfer action as two independent transfers.

Detection. We flag $\mathcal{P}_4$ when application logic (i) manually merges events from the newly created EventManager from CacheContext into the parent EventManager via EmitEvent or EmitEvents, and (ii) builds under an SDK version where CacheContext already performs this merge implicitly, as shown in Listing 6 (green part). We validate suspected cases by exercising the mock execution path that emits a deterministic event set and checking whether the final event log contains duplicates, even though the underlying state transition is executed only once.

Listing 6: CacheContext implementation in different SDK versions (v0.46.3 and v0.46.4).

```
func (c Context) CacheContext() (..) {
    cms := c.ms.CacheMultiStore()
    cc = c.WithMultiStore(cms).WithEventManager(
     NewEventManager())
    writeCache = func() {
+    c.EventManager()
+    .EmitEvents(cc.EventManager().Events())
        cms.Write()
    }
    return cc, writeCache
}
```

Mitigation. Developers should ensure that application-level event handling aligns with the semantics of the currently used SDK version. When upgrading to SDK versions that automatically propagate events from CacheContext, any event-merging logic should be removed.

4.3 Storage Invariant Violation

Violating the storage invariant compromises transactional atomicity, allowing state updates made before a failure to be committed even after the transaction reverts, which can lead to severe security exploits, such as asset theft.

4.3.1 Out-of-Context Mutable State ($\mathcal{P}_5$)

$\mathcal{P}_5$ is caused by persisting mutable state outside the official Cosmos SDK storage design, i.e., Ctx.MultiStore. Cosmos SDK assumes that state updates are applied through the cache-backed MultiStore and are discarded entirely if a transaction fails. However, when a module maintains additional mutable state beyond the Context.MultiStore, updates to such a state may survive after transaction failure. As a result, a transaction can fail but still leave behind state changes that later transactions can observe and build upon. Listing 7 shows an example adapted from the Evmos/evm module, where a CommitStateDB maintained by the Keeper contains an in-memory storage cache. Evmos rolls back and clears the CommitStateDB-backed state upon message handler execution failure. However, as discussed in §2.3, a single transaction may contain multiple messages. If an earlier Msg1 executes successfully, the corresponding CommitStateDB state is finalized and retained, whereas a subsequent Msg2 can still cause the transaction to abort. In this case, the SDK rolls back the MultiStore updates introduced by Msg1 and Msg2, but the CommitStateDB state produced by Msg1 is not reverted. Consequently, storage mutations originating from a logically failed transaction can persist and leak into later executions, breaking storage atomicity.

Listing 7: Keeper-owned `CommitStateDB` maintaining out-of-context storage state.

```
type Keeper struct {
        ...
        CommitStateDB *types.CommitStateDB
        Bloom   *big.Int
}
```

Detection. We detect $\mathcal{P}_5$ by examining whether a module `Keeper` defines unexpected mutable fields. `Keeper` is each module's core long-lived data structure, containing essential components (e.g., storage keys and the message router interface). It is initialized at node startup and persists until node termination, making it a natural place to store mutable state that may violate storage atomicity across the transaction execution pipeline. After excluding standard and expected Keeper fields, we identify non-standard fields that could be modified. To validate whether a project is affected by $\mathcal{P}_5$, we execute a single transaction containing two messages. `Msg1` mutates keeper-owned state outside the `MultiStore`, while `Msg2` is designed to fail and abort the transaction. Although the `MultiStore`-backed state is rolled back, the keeper-owned state modified by `Msg1` remains. If this state is still observable after transaction failure, the I_3 storage invariant is violated.
Mitigation. To preserve storage atomicity, modules should ensure that mutable state is confined to the `MultiStore` in `Ctx` and participates in the SDK's commit and rollback semantics.

4.4 Execution Sequence Invariant Violation

I_4 requires that all message execution paths traverse the validation and admission checks enforced by the transaction pipeline, including signature verification, fee deduction, gas initialization, and module-specific validation. As illustrated in Fig. 3, violations of this invariant arise when messages are executed through non-standard paths that bypass parts of the pipeline. We identify three pitfall patterns: (i) bypassing module-internal checks via cross-module invocation, (ii) bypassing signer verification via internally routed messages, and (iii) bypassing customized gas validation via inner messages. Each pattern allows a message to be executed without the intended checks, thereby breaking the security assumptions of the affected modules.

4.4.1 Missing Error Propagation ($\mathcal{P}_6$)

$\mathcal{P}_6$ arises when a module invokes another module's exported method but fails to propagate the returned `error`. In the Cosmos SDK, cross-module exported-method interactions (e.g., `k.bankKeeper.SendCoins(...)`) explicitly encode critical admission checks in their return values. If the caller discards the `error` and continues execution, it can commit state updates that semantically depend on a check that never actually passed, thereby violating the I_4 check invariant. Fig. 3(a) shows the exploit structure of $\mathcal{P}_6$. An attacker triggers another module's exported method that fails an essential admission check (e.g., insufficient balance), but the caller still reaches and commits a dependent state update in the message handler. We already introduced a concrete, real-world instance of this pitfall in our motivating example (see §3.1), where a handler ignores the `error` returned by `bank.SendCoins` and unconditionally commits a subsequent ownership update.
Detection. We detect $\mathcal{P}_6$ by identifying call sites where (i) a keeper call may fail and returns an `error`, but (ii) the caller ignores the returned error. To validate exploitability, we trigger the failing branch (e.g., bidding with an insufficient balance so that the bank transfer returns an `error`) and observe whether the dependent state transition (e.g., recording the bid as highest) is still committed.
Mitigation. All cross-module exported methods that return an `error` must be explicitly checked. Application logic should abort immediately upon failure.

4.4.2 Missing Signature Verification ($\mathcal{P}_7$)

$\mathcal{P}_7$ arises when internally routed messages bypass signer validation. In the Cosmos SDK, signature verification is enforced by the `AnteHandler` and applies only to messages carried directly by a transaction. When a module constructs an inner message during execution and dispatches it via the message router (i.e., `router.Handler(...)`), the routed message does not undergo signer verification again. If no explicit authorization check is enforced at the routing site, the inner message may specify an attacker-chosen signer and still execute successfully. This behavior violates the I_4 check invariant and enables full signer impersonation. Downstream modules execute the routed message as authenticated, even though no valid signature exists for the effective signer. An attacker can therefore forge privileged messages such as `Bank` transfers from arbitrary accounts or `Gov` messages to submit or approve a malicious software upgrade, thereby achieving arbitrary control over the chain. As illustrated in Listing 8, the vulnerable implementation routes the inner message without verifying that its signer is authorized by the current execution context.

Listing 8: Vulnerable message routing without signer authorization guard.

```
- return k.router.Handler(ctx, sdkMsg)
+ if h.cdc.GetSigner(sdkMsg) != originAddr {
+  return error}
+ return k.router.Handler(ctx, sdkMsg)
```

Detection. We detect $\mathcal{P}_7$ by identifying routing sites where (i) a module constructs an inner `sdk.Msg` and executes it via the message router (i.e., `k.router.Handler(..)`), but (ii) the routing site does not enforce any explicit signer-authorization check that binds the inner message's signer set to the authenticated caller (e.g., no use of signer-extraction

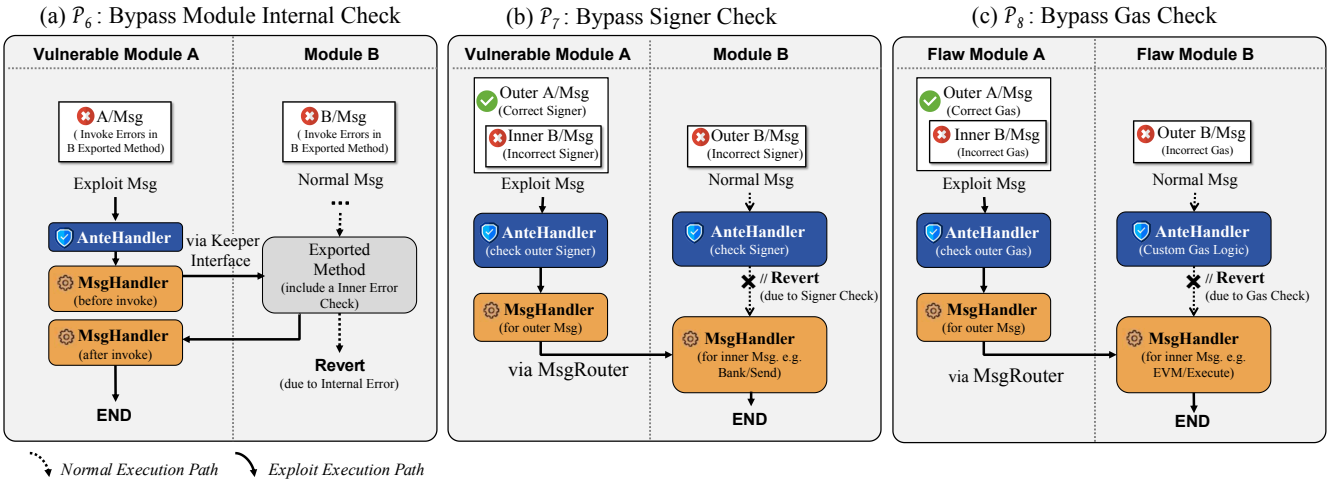


Figure 3: Attack paths illustrating pitfalls $\mathcal{P}_6$, $\mathcal{P}_7$, and $\mathcal{P}_8$.

APIs such as `GetSigner` / `GetSignerV2` before routing). We validate suspected cases via constructing the exploit path shown in Fig. 3(b). We submit an outer message that passes `AnteHandler` signature checks, but whose handler internally builds an inner message with an attacker-chosen signer and dispatches it via `k.router.Handler(..)`. If the routed inner message reaches the target module handler and executes successfully, we confirm that the routing site bypasses signer validation and the system is affected by $\mathcal{P}_7$.

Mitigation. Modules must not treat internally routed messages as authenticated. All routing paths must enforce explicit signer authorization checks.

4.4.3 Missing Gas-Limit Verification ($\mathcal{P}_8$)

In the Cosmos SDK, gas limits and fee deductions are enforced at the `AnteHandler` stage prior to message execution. To support specialized semantics, certain modules such as EVM-compatible modules deploy customized `AnteHandler` logic to adjust gas accounting outside of the Cosmos SDK's standard `GasMeter` tracking. However, this design introduces a flaw when gas-intensive logic is triggered by inner messages constructed from an outer message and executed via `k.router.Handler()`. In this setting, an attacker can bypass the `AnteHandler`'s validation of transaction-level gas limits by embedding an inner message with an arbitrarily large gas limit that is never validated. As a result, the customized `AnteHandler` is skipped, allowing the inner message to execute without the intended upper-bound gas enforcement. Notably, this issue does not arise when a module relies on the Cosmos SDK's official `GasMeter` for gas accounting, since gas consumption remains constrained by the outer message's `GasMeter` during execution.

Detection. We identify candidates based on the following two conditions. (i) the application contains a module that invokes `k.router.Handler()` to execute inner messages such as the `group`, `wasm`, and `authz` modules, and (ii) the application contains a module that is guarded by customized gas validation logic in the *AnteHandler*. To validate the issue, we construct a transaction that triggers the *exploit* path in Fig. 3(c). We submit an outer message `A/Msg` that passes *AnteHandler*, and its handler executes an over–gas-limit inner `B/Msg` via `k.router.Handler()`, so that `B/Msg` executes without entering Module B's customized *AnteHandler*. Under the intended design, the same `B/Msg` should be rejected on the normal (right) path by Module B's gas validation. Therefore, if `B/Msg` succeeds only when routed as an inner message in our exploit, we confirm missing gas-limit verification.

Mitigation. Developers should prohibit routing arbitrary messages into modules whose gas checks are implemented in customized *AnteHandlers* rather than the standard `GasMeter`.

5 Evaluation

In this research, we answer the following research questions:

RQ1: How prevalent are these pitfalls in real-world Cosmos–based blockchains?

RQ2: How do these pitfalls propagate across projects in the Cosmos SDK ecosystem?

RQ3: What is the real-world security impact of these pitfalls in Cosmos-based blockchains?

5.1 Dataset collection

We use two sources for our evaluation. First, we collect 150 chains from the official Cosmos *chain-registry* [15]. Second, we mine GitHub for Cosmos SDK projects via keyword and

Table 2: Summary of discovered vulnerabilities.

Type	#Val	#Ack (Fixed)	Type	#Val	#Ack (Fixed)
$\mathcal{P}_1$	8	4 (2)	$\mathcal{P}_5$	1	1 (1)
$\mathcal{P}_2$	7	3 (1)	$\mathcal{P}_6$	11	5 (5)
$\mathcal{P}_3$	10	1 (1)	$\mathcal{P}_7$	1	1 (1)
$\mathcal{P}_4$	24	9 (6)	$\mathcal{P}_8$	3	2 (2)
#Total Validated: 65			**#Total Ack: 26 (19 Fixed)**		

dependency-based matching (e.g., imports of SDK modules and dependency signals).

5.2 RQ1: Prevalence

Detection & Validation. We identify vulnerabilities through a two-stage process that combines pattern matching with manual validation. First, based on the pitfall taxonomy and detection strategies in §4, we implement pitfall-specific pattern matchers using regex-based API keyword rules, and apply them by matching against large-scale code corpora, including public repositories on *GitHub* and the chain data in the chain registry [15]. For example, for $\mathcal{P}_1$ type pitfalls, we collect all projects whose code invokes the `WithGasMeter` API, but lacks any occurrence of gas consumer in `defer` block, according to regular API keyword rules. We adopt this lightweight, API keyword matching approach to enable scalable ecosystem-wide analysis, as many vulnerable projects are scattered across *GitHub* and often hidden in less visible or forked repositories. While precise static analysis offers stronger semantic guarantees, it does not scale well to broad ecosystem searches, which runs counter to our goal of providing a wide-coverage assessment rather than building a fully precise vulnerability detector. Using this lightweight pattern-matching approach, we identified $\mathcal{P}_6$ type vulnerabilities during the testnet development of projects such as OLLO, Jackal, and Ununifi, which could have enabled free lending or unpaid asset acquisition after mainnet deployment. Early identification allowed these issues to be fixed before production. In the second stage, we manually inspect flagged code and construct PoCs for unique pitfall triggers, selecting the tier that most directly confirms each vulnerability. For single-module flaws, we use *unit-test* PoCs by invoking the relevant entry point. When full execution output is too noisy to isolate the defect, we use *mock* PoCs by extracting the vulnerable code pattern into a controlled harness, such as directly demonstrating missing or duplicated events. When the exploit spans multiple modules, we use *full-chain* PoCs by deploying a local chain and submitting a sequence of attack transactions.

Vulnerability Results. Table 3 reports the validation results for *chains with economic value*, which we treat as a *highlight* subset of the ecosystem. For some of these projects, we further discuss the practical impact (see §5.4). Table 2 complements this view by summarizing the complete discovery results across all analyzed projects, including inactive or test-only chains. Across the analyzed projects, we confirm 65 vulnerabilities. Each independent root-cause instance is backed by a corresponding PoC validation. Most validated cases are confirmed via at least unit-level testing, while pitfalls that are difficult to reproduce in unit tests (e.g., $\mathcal{P}_7$ and $\mathcal{P}_8$) are validated by deploying a local chain and constructing attack transactions. Appendix B provides a per-finding table that records each PoC validation type and remediation status.

Table 3: $\mathcal{P}_1$–$\mathcal{P}_8$ pitfall matrix across economically relevant Cosmos-based chains. Each cell denotes the disclosure status of a pitfall: (–) not observed, (○) observed but unconfirmed, (◐) acknowledged by the project team and committed to remediation, but without an identifiable public fix at the time of writing, (●) publicly fixed through a deployed patch or merged commit. Market capitalization data is sourced from CoinMarketCap at each chain's January 2026 peak [13]. For chains without CoinMarketCap coverage, our artifact provides evidence that they had economic value at the time of writing.

Chain (Market Cap)	$\mathcal{P}_1$	$\mathcal{P}_2$	$\mathcal{P}_3$	$\mathcal{P}_4$	$\mathcal{P}_5$	$\mathcal{P}_6$	$\mathcal{P}_7$	$\mathcal{P}_8$
Secret ($57.4M)	–	–	–	–	–	–	●	–
Evmos ($0.64M)	–	–	–	–	●	–	–	–
Mantra ($93.62M)	●	●	●	–	–	–	–	–
Osmosis ($49.45M)	●	–	○	–	–	–	–	–
J**	◐	◐	–	–	–	–	–	–
N**	◐	◐	○	–	–	–	–	–
Jackal ($1.27M)	–	–	–	–	–	●	–	–
Injective ($563.77M)	–	–	–	–	–	–	–	●
Stride ($1.28M)	–	–	–	●	–	–	–	–
Kava ($99.82M)	–	–	–	◐	–	–	–	–
Irisnet ($10.33M)	–	–	–	●	–	–	–	–
Helios	–	–	–	◐	–	–	–	–
Sei ($842.62M)	–	–	○	–	–	–	–	–
Saga	–	–	–	○	–	–	–	–
Planq($85.71K)	–	–	–	○	–	–	–	–
Mitosis ($13.46M)	–	–	○	–	–	–	–	–
Guru	–	–	–	○	–	–	–	–
Mezod	–	–	–	○	–	–	–	–
Hetu	–	–	–	●	–	–	–	–
UptickNetwork	–	–	–	●	–	–	–	●
Ununifi	–	–	–	–	–	●	–	–
PellNetwork ($0.51M)	–	–	–	–	–	–	–	○

5.3 RQ2: Propagation

Vulnerability Propagation in the Ecosystem. Beyond isolated vulnerable projects, our analysis reveals the *propagation effect* in the Cosmos SDK ecosystem, where execution pitfalls spread across projects through module reuse, composition, and developer-facing documentation. Fig. 4 summarizes three propagation patterns observed in our study, denoted as Type a, Type b, and Type c, while Table 4 summarizes the propagation scope of each risk by reporting its source module

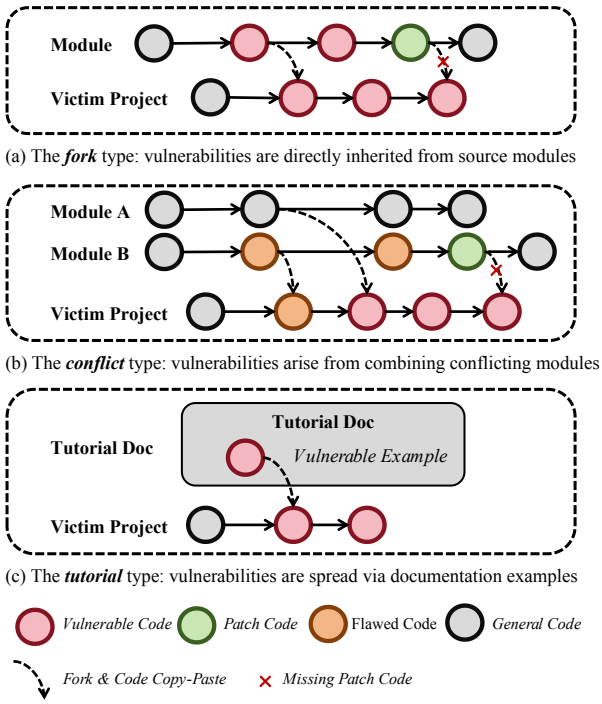


Figure 4: Vulnerability propagation patterns.

Table 4: Instances of pitfalls introduced by reusing flawed modules.

Source Module	Risk ID	Risk Type	#Affected Items.
`osmosis/tokenfactory` Module	$\mathcal{P}_1$	Type a	7
	$\mathcal{P}_2$	Type a	6
	$\mathcal{P}_3$	Type a	6
`evmos/evm` Module	$\mathcal{P}_4$	Type a	21
	$\mathcal{P}_8$	Type b	3
`ignite` Tutorial	$\mathcal{P}_6$	Type c	3

and the number of affected items, with source items counted when applicable. Beyond manual analysis, we also performed code-similarity analysis on vulnerable code. The results are included in the artifact.

- **Fork-based propagation (Type a).** In Type a cases, the victim directly imports code from a flawed source module, thereby inheriting the same execution pitfall. As shown in Table 4, the `evmos/evm` module introduces one Type a risk, $\mathcal{P}_4$. After the *import time*, downstream projects incorporate the corresponding vulnerable logic from the EVM module. For $\mathcal{P}_4$, this logic is subsequently observed in 21 affected items, indicating widespread reuse of the flawed implementation. A similar pattern is observed for the `osmosis/tokenfactory` module, where Type a risks $\mathcal{P}_1$, $\mathcal{P}_2$, and $\mathcal{P}_3$ are imported after their respective import times and appear in 7, 6, and 6 affected items, respectively.
- **Conflict-based propagation (Type b).** Type b propagation corresponds to a vulnerability introduced by the composition of multiple modules with incompatible assumptions. Risk $\mathcal{P}_8$, originating from the `evmos/evm` module, is classified as Type b. In this case, the vulnerable behavior is absent in the source module in isolation and manifests only after the imported code interacts with other modules within the victim project, affecting 3 items.
- **Tutorial-driven propagation (Type c).** Type c propagation occurs when execution pitfalls are introduced through tutorial code, with a representative example being the `Ignite` (the official Cosmos scaffolding framework) tutorials. Specifically, both the `Loan` and `Nameservice` example modules provided by `Ignite` tutorials contain $\mathcal{P}_6$. The `Loan` flaw enables cost-free borrowing, while the `Nameservice` flaw allows unpaid asset acquisition. We identify three affected items associated with these tutorial examples, including the source tutorial itself and downstream projects that reuse its code. For instance, OLLO incorporates the `loan` module, and Jackal adopts the `nameservice` example. As a result, $\mathcal{P}_6$ propagates from tutorial documentation into deployed systems.

5.4 RQ3: Impact

Asset impact. We observe that $\mathcal{P}_5$ and $\mathcal{P}_7$ can lead to asset theft. Appendix A summarizes our conservative direct asset-loss estimate and propagation-scale context. We next elaborate on the severe cases observed in Evmos and Secret.

- **Evmos.** Evmos (formerly Ethermint) exhibits a $\mathcal{P}_5$ flaw caused by a keeper-local `CommitStateDB` that caches EVM state across message executions. During EVM execution, state reads prioritize `CommitStateDB` and fall back to `Ctx`'s `MultiStore` only on cache miss. However, `CommitStateDB` updates from successful EVM messages are not reverted when the transaction fails, resulting in a persistent divergence between `CommitStateDB` and `Ctx`'s `MultiStore`. To illustrate the exploit, consider an ERC20 contract and three attacker-controlled accounts A, B, C. Initially, both `CommitStateDB` and `Ctx`'s `MultiStore` record balances $(A{=}1, B{=}0, C{=}0)$. The attacker then submits a multi-message transaction where the first message transfers 1 token from A to B, updating the `CommitStateDB` view to $(A{=}0, B{=}1, C{=}0)$, while the second message is forced to fail. The transaction is reverted at the SDK layer, leaving `Ctx`'s `MultiStore` unchanged at $(A{=}1, B{=}0, C{=}0)$, but the polluted `CommitStateDB` state persists. Next, within the same block, a normal transaction transfers 1 token from B to C; this execution observes the forged `CommitStateDB` balance and succeeds, resulting in a `Ctx`'s `MultiStore` state of $(A{=}1, B{=}0, C{=}1)$. At block finalization, the keeper state is committed to the SDK state, resulting in an additional token and enabling arbitrary token minting and direct asset theft. This vulnerability was originally discovered by

one of our authors and assigned CVE-2021-25837. Notably, Evmos was later widely reused as an EVM stack within the Cosmos ecosystem, including in projects such as Injective, Cronos, and ZetaChain.

- **Secret.** In Secret Network, we identified a $\mathcal{P}_7$ class vulnerability that allows a `Wasm` contract to emit internally routed messages with an arbitrary sender, without possessing the private key. To demonstrate exploitability, we constructed a proof-of-concept in the test environment by deploying a malicious contract that emits a `bank.MsgSend` with a victim account as the sender, successfully transferring one token from the victim to the attacker. As a result, approximately $50 million worth of tokens were at risk of theft. Beyond direct asset theft, forged-signer execution also enables additional attacks, including *governance manipulation* and *consensus interference*, such as voting on a malicious chain upgrade proposal or withdrawing staked funds under an impersonated identity. Notably, the Secret team demonstrated exceptional professionalism throughout the disclosure process, delivering a fix and completing a mainnet upgrade within four hours of our report.

Denial-of-service (DoS). We find that $\mathcal{P}_2$ and $\mathcal{P}_8$ can directly cause *chain halts*. In contrast, $\mathcal{P}_1$ mainly enables *resource-exhaustion DoS* attacks with comparatively limited impact. We next elaborate on two high-impact cases in Injective and Mantra where these pitfalls resulted in chain halts.

- **Mantra.** In Mantra, we discovered a $\mathcal{P}_2$ class vulnerability that allows a single transaction to trigger unbounded execution. By exploiting recursive `WithGasMeter` calls, an attacker can force validators to execute transactions in an effectively unbounded manner, leading to chain-halting liveness failures. Notably, although the module enforces an explicit bound on recursion depth (e.g., a maximum depth of 500), this limit does not prevent the attack. To bypass the depth constraint, our PoC avoids unbounded recursion depth and instead performs two recursive invocations at each level. As a result, a single transaction expands to $2^{500}-1$ total executions, a scale that exceeds any practical execution capacity. This issue has been assigned CVE-2025-61595. In follow-up discussions, we worked with Mantra developers to propose Cosmos SDK-level mitigations, particularly a new `ProxyGasMeter`[1] design for $\mathcal{P}_1$ and $\mathcal{P}_2$.

- **Injective.** In Injective, we identified a $\mathcal{P}_8$ vulnerability that allows EVM execution paths to be reached without proper gas-limit enforcement. By setting the EVM gas parameter to $2^{64}-1$, an attacker can trigger transactions with gas consumption far exceeding any practical block limit (about 10^7), enabling a chain halt.

[1] https://github.com/cosmos/cosmos-sdk/pull/25303

Off-chain misleading. Event-related pitfalls such as $\mathcal{P}_3$ and $\mathcal{P}_4$ can lead to inconsistent event semantics that mislead off-chain infrastructure. For instance, $\mathcal{P}_4$ in Stride may cause a single state transition to emit duplicated events, leading indexers to misinterpret one action as multiple actions. In addition, in Sei we observe $\mathcal{P}_3$, where events are silently dropped, causing off-chain infrastructure to miss real state changes and resulting in incomplete indexing and inconsistent accounting.

6 Related Work

Blockchain Consensus Security. A large body of prior work studies the security and correctness of blockchain systems [2,7,9,30,37,38,46,50,52,54,55]. These works typically model the blockchain as a deterministic state machine and assume that the application layer faithfully implements the expected state transition function. As a result, application logic is treated as a black box, and potential misuse of developer-defined customization points at the application layer is mainly outside the scope of existing analyses.

Blockchain Application Security. Blockchain application security has been widely studied, particularly in smart contract platforms such as Ethereum. Prior research has identified vulnerabilities and has proposed static analysis to detect them [1, 4, 6, 8, 21, 23–25, 27, 28, 33, 39, 42, 51, 53, 57, 58]. These approaches target VM environments in which application logic runs within a virtual machine, with a fixed, isolated execution model. In contrast, the Cosmos-based projects are implemented directly in native code and outside the scope of existing smart contract analyses.

Cosmos-based Blockchain Security. Within the Cosmos ecosystem, prior work has primarily focused on non-deterministic issues [45,49]. While effective for preventing violations of determinism, these methods address only a narrow class of application-layer risks.

7 Discussion

While this work identifies eight recurring pitfall classes and uncovers several high-impact vulnerabilities in Cosmos-based blockchains, our findings are not exhaustive. Our analysis focuses on the transaction execution lifecycle and does not cover block-level execution semantics (e.g., `BeginBlocker`). However, such hooks are typically high-privilege and not directly reachable by external adversaries, which often constrains their exploitability. We therefore deliberately scope our study to the transaction level. Moreover, the Cosmos SDK evolves rapidly, making it difficult to enumerate all pitfalls in a stable manner, as new issues may emerge over time (e.g., $\mathcal{P}_4$ introduced in version `0.46.4`). In addition, unknown exploitation paths and customized risky APIs, such as overrides or unseen variants introduced by SDK forks, may affect recall, requiring manual analysis. Despite these limitations, the large number

of real-world vulnerabilities we uncover demonstrates the impact of our work and reduces their recurrence in future projects. We hope that this work can serve as a data-driven foundation to guide the design of static analysis frameworks by clarifying domain-specific models, abstraction levels, and security-sensitive API variants. More importantly, we hope these findings could provide insights for Cosmos framework developers to implement safer customization mechanisms.

8 Conclusion

We explored application-layer security pitfalls in Cosmos-based chains, an understudied direction beyond consensus security and non-determinism. We characterized the transaction execution pipeline, summarized four execution invariants, and observed eight recurring development pitfalls. Our study further revealed the prevalence and propagation of these pitfalls across the Cosmos ecosystem, and their real-world impact on deployed projects. We uncovered 65 validated vulnerabilities in Cosmos-based blockchains, including high-impact asset-theft and denial-of-service threats. Our study helps shift attention toward application-layer correctness in blockchains and motivates broader efforts from both researchers and practitioners to build a more robust ecosystem.

Acknowledgments

This work was supported in part by the Hong Kong RGC Projects (No. PolyU15224121 and PolyU15231223), the Hong Kong ITF Project (No. PRP/005/23FX), and the Hong Kong RGC Grant for Theme-based Research Scheme Project (No. T43-513/23-N).

Ethical Considerations

This work follows a stakeholder-based ethical analysis consistent with the conference ethics guidelines and established norms in computer security research. As our study examines real-world vulnerabilities in financial- and infrastructure-critical blockchain software, we explicitly considered how our methods and disclosures could affect relevant parties and adopted measures to minimize foreseeable harm while preserving scientific value.

Stakeholders considered. We considered multiple stakeholder groups that may be affected by this research. (1) core maintainers of Cosmos-based chains, including validators and governance participants, (2) dependent third-party services such as exchanges, wallets, bridges, block explorers, and other infrastructure providers, and (3) the broader Cosmos development and research community, whose security practices and assumptions may be influenced by our findings.

Potential impacts on stakeholders. The intended positive impact of this research is to improve the security of decentralized systems by identifying recurring pitfall classes and concrete vulnerabilities, thereby prompting timely remediation across the ecosystem and mitigating the risk of exploitation. We also considered potential negative impacts across stakeholder groups. For project maintainers and operators, premature or overly detailed disclosure could enable attackers to reproduce exploitation techniques before patches are deployed, leading to financial loss or service disruption. For dependent third-party services and end users, exploitation of unpatched vulnerabilities may result in indirect harm, including asset loss or reduced system availability.

Mitigations for negative impacts. To mitigate these risks, we followed responsible disclosure practices. High-impact vulnerabilities were reported privately to the relevant project teams, with sufficient detail to support triage and remediation, and with a reasonable remediation window prior to publication. For projects that committed to remediation but for which we could not identify a public fix PR, we anonymize their project names. When projects were unresponsive or declined remediation, and the issue was assessed as critical, we intentionally excluded such cases from our published results, retaining only high-level lessons that do not provide a clear blueprint for exploitation. Throughout the study, we did not exploit vulnerabilities on public mainnets, access real user assets, or disrupt network operations.

Decision-making and justification. We decided to conduct and publish this research by weighing its scientific and societal benefits against foreseeable risks. We believe this work is ethically justified because the systematic understanding of these pitfalls can meaningfully improve user safety and ecosystem resilience. We prioritize harm minimization through responsible disclosure, avoidance of mainnet exploitation, and selective withholding of sensitive details, while maintaining sufficient transparency to support reproducibility and future defensive research.

Open Science

We provide artifacts in https://doi.org/10.5281/zenodo.20339215 and github [2] to support the evaluation of our work. The `poc/` directory contains proof-of-concept exploits and keyword patterns used to match candidate projects. The `highlight-findings/` directory documents high-impact vulnerabilities, including exploitation workflows and impact analysis. The `record/acknowledgments/` directory and `README.md` provide responsible disclosure evidence, including private acknowledgments, CVE assignments, and fix commit links. The `record/negative-record/` directory documents projects that matched keyword patterns but were excluded after manual analysis. The `record/similarity/` directory contains the pairwise similarity script and raw data supporting the fork-propagation analysis.

[2] https://github.com/cosmos-bug-finder/Cosmos-Pitfalls/

References

[1] Sefa Akca, Ajitha Rajan, and Chao Peng. Solanalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 482–489. IEEE, 2019.

[2] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of blockchains. 2016. URL: https://api.semanticscholar.org/CorpusID:59082906.

[3] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1(22-23):5–7, 2013.

[4] Jie Cai, Jiachi Chen, Tao Zhang, Xiapu Luo, Xiaobing Sun, and Bin Li. Detecting reentrancy vulnerabilities for solidity smart contracts with contract standards-based rules. *IEEE Transactions on Information Forensics and Security*, 2025.

[5] Jorge Castillo. Cosmos unmasked: A security guide to review cosmos applications, December 2024. URL: https://medium.com/@jorgecastillot2017/cosmos-unmasked-a-security-guide-to-review-cosmos-application-cfc9efbdd205.

[6] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. Soda: A generic online detection framework for smart contracts. In *NDSS*, 2020.

[7] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *Information Security Practice and Experience: 13th International Conference, ISPEC 2017, Melbourne, VIC, Australia, December 13–15, 2017, Proceedings 13*, pages 3–24. Springer, 2017.

[8] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. Wasai: uncovering vulnerabilities in wasm smart contracts. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 703–715, 2022.

[9] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jia-Guang Sun. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2517–2532. IEEE, 2023. doi:10.1109/SP46215.2023.10179386.

[10] Ignite CLI. Unchecked error in cli tutorial, 2022. URL: https://github.com/ignite/cli/issues/2828.

[11] CoinDesk. Juno pushed offline incident, 2022. URL: https://www.coindesk.com/tech/2022/04/06/cosmos-based-juno-blockchain-pushed-offline-in-apparent-attack/.

[12] CoinMarketCap. Cosmos ecosystem, November 2024. URL: https://web.archive.org/web/20241126221545/https://coinmarketcap.com/view/cosmos-ecosystem/.

[13] CoinMarketCap. Cosmos ecosystem, 2026. URL: https://coinmarketcap.com/.

[14] Cosmos. Cometbft, 2024. URL: https://github.com/cometbft/cometbft.

[15] Cosmos. Cosmos Chain Registry. https://github.com/cosmos/chain-registry, 2026.

[16] Cosmos. Cosmos hub documentation, 2026. URL: https://docs.cosmos.network/hub/v25.

[17] Cosmos. Cosmos sdk documentation, 2026. URL: https://docs.cosmos.network/.

[18] Cosmos. Cosmwasm: smart contract platform for the multi-chain world, February 2026. URL: https://cosmwasm.com/.

[19] CosmWasm. Sub-context gas not consumed on non-outofgas panics, 2025. URL: https://github.com/CosmWasm/advisories/blob/main/CWAs/CWA-2025-004.md.

[20] CometBFT Developers. Cometbft abci specification, 2026. URL: https://docs.cometbft.com/v0.38/.

[21] Thomas Durieux, João F Ferreira, Rui Abreu, and Pedro Cruz. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, pages 530–541, 2020.

[22] Evmos. Evmos documentation, 2025. URL: https://github.com/evmos/evmos/.

[23] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

[24] João F. Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. Smartbugs: A framework to analyze solidity smart contracts. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 1349–1352. IEEE, 2020. doi:10.1145/3324884.3415298.

[25] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 557–560, 2020.

[26] Hacken. Cosmos appchain security, 2025. URL: https://hacken.io/discover/cosmos-appchain-security/.

[27] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*, pages 161–179. Springer, 2020.

[28] Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. Eosafe: Security analysis of eosio smart contracts. In *USENIX Security Symposium*, pages 1271–1288, 2021.

[29] Zheyuan He, Zihao Li, Ao Qiao, Jingwei Li, Feng Luo, Sen Yang, Gelei Deng, Shuwei Song, Xiaosong Zhang, Ting Chen, and Xiapu Luo. Maat: Analyzing and optimizing overcharge on blockchain storage. In Haryadi S. Gunawi and Vasily Tarasov, editors, *23rd USENIX Conference on File and Storage Technologies, FAST 2025, Santa Clara, CA, February 25-27, 2025*, pages 123–137. USENIX Association, 2025. URL: https://www.usenix.org/conference/fast25/presentation/he.

[30] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 129–144, 2015.

[31] Hellobloc. Missing error handler commit, 2023. URL: https://github.com/crytic/building-secure-contracts/commit/84d54ae043773c.

[32] Jumpcrypto. Stealing gas: Bypassing ethermint ante handlers, 2023. URL: https://jumpcrypto.com/resources/stealing-gas-bypassing-ethermint-ante-handlers.

[33] Que-Ping Kong, Zi-Yan Wang, Yuan Huang, Xiang-Ping Chen, Xiao-Cong Zhou, Zi-Bin Zheng, and Gang Huang. Characterizing and detecting gas-inefficient patterns in smart contracts. *Journal of Computer Science and Technology*, 37(1):67–82, 2022.

[34] Injective Labs. Injective protocol documentation, 2026. URL: https://injective.com/.

[35] Osmosis Labs. Osmosis documentation, 2026. URL: https://docs.osmosis.zone/.

[36] Sei Labs. Sei whitepaper, 2026. URL: https://www.sei.io/.

[37] Nicolas Lagaillardie, Mohamed Aimen Djari, and Önder Gürcan. A computational study on fairness of the tendermint blockchain protocol. *Inf.*, 10(12):378, 2019. URL: https://doi.org/10.3390/info10120378, doi:10.3390/INFO10120378.

[38] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Richard Tang, XiaoFeng Wang, and Xiapu Luo. As strong as its weakest link: How to break blockchain dapps at rpc service. In *NDSS*, 2021.

[39] Shang-Wei Lin, Palina Tolmach, Ye Liu, and Yi Li. Solsee: a source-level symbolic execution engine for solidity. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1687–1691, 2022.

[40] Feng Luo, Huangkun Lin, Zihao Li, Xiapu Luo, Ruijie Luo, Zheyuan He, Shuwei Song, Ting Chen, and Wenxuan Luo. Towards automatic discovery of denial of service weaknesses in blockchain resource models. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1016–1030, 2024.

[41] Pengxiang Ma, Ningyu He, Yuhua Huang, Haoyu Wang, and Xiapu Luo. Abusing the ethereum smart contract verification services for fun and profit. *arXiv preprint arXiv:2307.00549*, 2023.

[42] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, 2019.

[43] Arvind Narayanan, Joseph Bonneau, Edward W. Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction*. Princeton University Press, 2016. URL: http://press.princeton.edu/titles/10908.html.

[44] Trail of Bits. Not so smart contracts: Cosmos, 2023. URL: https://github.com/crytic/building-secure-contracts/tree/master/not-so-smart-contracts/cosmos.

[45] Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, and Fausto

Spoto. Information flow analysis for detecting non-determinism in blockchain. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, Seattle, Washington, United States, July 17-21, 2023*, LIPIcs, pages 23:1–23:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.23, doi:10.4230/LIPICS.ECOOP.2023.23.

[46] Zhiniang Peng and Yuki Chen. All roads lead to rome: Many ways to double spend your cryptocurrency. *CoRR*, abs/1811.06751, 2018. URL: http://arxiv.org/abs/1811.06751, arXiv:1811.06751.

[47] Protos. Saga becomes latest victim in defi hacking spree, 2026. URL: https://protos.com/saga-becomes-latest-victim-in-defi-hacking-spree/.

[48] Secret. Secret network documentation, 2026. URL: https://scrt.network/.

[49] Jasper Surmont, Weihong Wang, and Tom Van Cutsem. Static application security testing of consensus-critical code in the cosmos network. In *2023 5th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pages 1–8. IEEE, 2023.

[50] Wai Yan Maung Maung Thin, Naipeng Dong, Guangdong Bai, and Jin Song Dong. Formal analysis of a proof-of-stake blockchain. In *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*, pages 197–200. IEEE Computer Society, 2018. doi:10.1109/ICECCS2018.2018.00031.

[51] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In Roberto Tonelli, Giuseppe Destefanis, Steve Counsell, and Michele Marchesi, editors, *1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018*, pages 9–16. ACM, 2018. doi:10.1145/3194113.3194115.

[52] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized testing of byzantine fault tolerant algorithms. *Proc. ACM Program. Lang.*, 7(OOPSLA1):757–788, 2023. doi:10.1145/3586053.

[53] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[54] Karl Wüst and Arthur Gervais. Ethereum eclipse attacks. Technical report, ETH Zurich, 2016.

[55] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 349–365. USENIX Association, 2021. URL: https://www.usenix.org/conference/osdi21/presentation/yang.

[56] Jiashuo Zhang, Jianbo Gao, Yue Li, Ziming Chen, Zhi Guan, and Zhong Chen. Xscope: Hunting for cross-chain bridge attacks. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–4, 2022.

[57] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 615–627. IEEE, 2023.

[58] Li Zihao, He Zheyuan, Luo Xiapu, Chen Ting, and Zhang Xiaosong. Unveiling financially risky behaviors in ethereum erc20 token contracts. *Chinese Journal of Electronics*, 35(2):668–686, 2026. URL: https://cje.ejournal.org.cn/en/article/doi/10.23919/cje.2025.00.156, doi:10.23919/cje.2025.00.156.

Appendix A. Asset-Loss Impact Estimate

Table 5 summarizes direct asset-loss exposure and Evmos-based propagation context. We count Secret and Evmos as direct asset-theft cases because their PoCs can transfer or mint native assets. Secret's estimate uses its CoinMarketCap [13] market capitalization at disclosure time. Evmos had no CoinMarketCap [13] record when P_5 was identified in 2021. But its native token valuation later peaked at roughly $2.2B in early 2022. We exclude later Evmos-based projects such as ZetaChain, Cronos, and Injective because the reused EVM code had already been fixed.

Table 5: Asset-loss exposure and market-cap context.

Case	Market-Cap Context
P_7 in Secret Network	$\sim$ \$50M
P_5 in Evmos (formerly Ethermint)	$\sim$ \$2.2B in 2022

Appendix B. Disclosure Status of Findings

Table 6: Structured disclosure-status table of all identified findings.

● denotes findings with a public patch, merged commit, release, or CVE-level remediation record, ◐ denotes findings acknowledged by the project team and committed to remediation but without an identifiable public fix at the time of writing, ○ denotes findings for which no project confirmation or remediation response was available.

Type	ProjectName	PoC Type	Status	Type	ProjectName	PoC Type	Status
$\mathcal{P}_1$	Mantra	UnitTest	●	$\mathcal{P}_4$	MTT Chain	Mock	○
$\mathcal{P}_1$	Osmosis	UnitTest	●	$\mathcal{P}_4$	Hetu Chain	Mock	●
$\mathcal{P}_1$	N**	UnitTest	◐	$\mathcal{P}_4$	Helios	Mock	◐
$\mathcal{P}_1$	Tower	UnitTest	○	$\mathcal{P}_4$	LambdaVM	Mock	○
$\mathcal{P}_1$	Phoenix	UnitTest	○	$\mathcal{P}_4$	Guru	Mock	◐
$\mathcal{P}_1$	JMES	UnitTest	○	$\mathcal{P}_4$	Bridgeless Core	Mock	●
$\mathcal{P}_1$	Symphony	UnitTest	○	$\mathcal{P}_4$	Egochain	Mock	○
$\mathcal{P}_1$	J**	UnitTest	◐	$\mathcal{P}_4$	CVN	Mock	○
$\mathcal{P}_2$	Mantra	UnitTest	●	$\mathcal{P}_4$	DE-EVM	Mock	○
$\mathcal{P}_2$	N**	UnitTest	◐	$\mathcal{P}_4$	Rarimo Core	Mock	○
$\mathcal{P}_2$	Tower	UnitTest	○	$\mathcal{P}_4$	Cosmos EVM	Mock	●
$\mathcal{P}_2$	Phoenix	UnitTest	○	$\mathcal{P}_4$	BRC Chain	Mock	○
$\mathcal{P}_2$	JMES	UnitTest	○	$\mathcal{P}_4$	Rollup EVM	Mock	○
$\mathcal{P}_2$	Symphony	UnitTest	○	$\mathcal{P}_4$	Artela	Mock	○
$\mathcal{P}_2$	J**	Mock	◐	$\mathcal{P}_4$	OLLO	Mock	○
$\mathcal{P}_3$	Sei	Mock	○	$\mathcal{P}_5$	Evmos	Full Chain	●
$\mathcal{P}_3$	Mitosis	Mock	○	$\mathcal{P}_6$	Jackal	UnitTest	●
$\mathcal{P}_3$	tgrade	Mock	○	$\mathcal{P}_6$	Ignite	UnitTest	●
$\mathcal{P}_3$	Symphony	Mock	○	$\mathcal{P}_6$	Side Protocol	UnitTest	●
$\mathcal{P}_3$	Osmosis	Mock	○	$\mathcal{P}_6$	OLLO	UnitTest	●
$\mathcal{P}_3$	Mantra	Mock	●	$\mathcal{P}_6$	Ununifi	UnitTest	●
$\mathcal{P}_3$	N**	Mock	○	$\mathcal{P}_6$	Stateset	UnitTest	○
$\mathcal{P}_3$	Tower	Mock	○	$\mathcal{P}_6$	Haven	UnitTest	○
$\mathcal{P}_3$	Phoenix	Mock	○	$\mathcal{P}_6$	ICPlaza	UnitTest	○
$\mathcal{P}_3$	JMES	Mock	○	$\mathcal{P}_6$	hero	UnitTest	○
$\mathcal{P}_4$	Stride	Mock	●	$\mathcal{P}_6$	decoGit	UnitTest	○
$\mathcal{P}_4$	Saga	Mock	○	$\mathcal{P}_6$	HalbornSecurity	UnitTest	○
$\mathcal{P}_4$	UptickNetwork	Mock	●	$\mathcal{P}_7$	Secret Network	Full Chain	●
$\mathcal{P}_4$	Kava	Mock	◐	$\mathcal{P}_8$	Injective	Full Chain	●
$\mathcal{P}_4$	Irisnet	Mock	●	$\mathcal{P}_8$	PellNetwork	Full Chain	○
$\mathcal{P}_4$	XteLabs	Mock	○	$\mathcal{P}_8$	UptickNetwork	Full Chain	●
$\mathcal{P}_4$	Planq	Mock	○				
$\mathcal{P}_4$	Mezod	Mock	○				
$\mathcal{P}_4$	Tenet-Evmos	Mock	○				