Revealing the Dark Side of Smart Accounts: An Empirical Study of EIP-7702 Incurred Risks in Blockchain Ecosystem

Mingyuan Huang[1], *Han Liu*[2,✉], *Shuo Yang*[3], *Daoyuan Wu*[4], *Shuai Wang*[1]

[1]*The Hong Kong University of Science and Technology, Hong Kong, China*
[2]*College of Cryptology and Cyber Science, Nankai University, Tianjin, China*
[3]*Sun Yat-sen University, Guangdong, China*
[4]*Lingnan University, Hong Kong, China*

Abstract

The introduction of smart accounts by EIP-7702 represents a major advancement for blockchain account abstraction, enabling externally owned accounts (EOAs) to be upgraded into programmable accounts while still preserving their original addresses. This advancement significantly enhances both account functionality and usability, but also redefines blockchain trust boundaries between EOAs and smart contract accounts (CAs), thereby altering security assumptions and creating opportunities for novel types of attack.

To systematically examine these risks, we classify smart account-based risks into three categories according to the type of victim accounts: EOA-targeted, CA-targeted, and composite attacks. We then develop specialized detection tools that combine large-scale transaction analysis with cross-contract static analysis to identify malicious behaviors. Applying these tools across seven blockchains that support EIP-7702, we detect 924 malicious contract accounts, including several previously unreported zero-day cases. These attacks have led to more than $2.3 million in losses and exposed over $10 million to potential compromise. We uncover multiple key insights into attacker behaviors. Specifically, we find that over 63% of EIP-7702 authorization transactions are associated with malicious EOA-targeted attacks, and nearly half of the most frequently authorized contracts are controlled by attackers. In addition, we identify existing evasion tactics that attackers use to circumvent detection, attack impacts observed in real-world incidents, and potential risks that may emerge in future deployments, underscoring the urgency of addressing smart account security in blockchain ecosystems.

1 Introduction

Blockchain technology, popularized by Bitcoin, has evolved to support smart contracts-programmable agreements enabling complex decentralized applications such as lending and asset management [41, 51]. As the first platform to support smart contracts, Ethereum reached a $381 billion market cap by July 2025 [23]. In such platforms, there are two main account types: Externally Owned Accounts (EOAs), controlled by users via private keys, and Contract Accounts (CAs), which are smart contracts with programmable logic that can manage assets and execute operations autonomously [19]. However, both EOAs and CAs have inherent limitations that restrict user flexibility. EOAs are limited to simple, atomic operations and lack native support for more advanced behaviors (for example, batching multiple calls). CAs, on the other hand, enable richer functionality but require users to move assets into a separate contract address before they can be managed programmatically. In practice, users must still control these contracts via an EOA: the EOA must hold ETH to pay gas for invoking the contract, whereas the contract address holds the assets being managed. This separation of asset ownership from execution logic makes both wallet management and gas provisioning more complex and confusing for users.

To address these limitations, EIP-7702 emerged [10], proposing a novel type of account abstraction, Smart Accounts (SAs). With EIP-7702, EOAs can be upgraded to SAs through authorization transactions, attaching other contracts to them. Once upgraded, SAs can delegate calls to the target smart contracts while preserving the context of the original EOAs. This mechanism enables SAs to retain their original addresses while supporting advanced transaction logic. Recognizing its significance, EIP-7702 was officially introduced in the Ethereum Pectra upgrade in 2025 [21]. As shown in Figure 1, we observe a rapid increase in the daily creation of newly upgraded SAs across seven EIP-7702-enabled blockchains. The total number of SAs has exceeded 3 million, indicating widespread and fast adoption of this new account abstraction.

However, SAs also introduce new security risks to the blockchain ecosystem, impacting both EOAs and CAs. On the one hand, upgrading an EOA to an SA enables it to execute contract code, but also exposes it to risks from malicious attached logic [10]. If an EOA accepts a malicious EIP-7702 authorization request, the embedded contract code may execute with full account privileges, leading to asset loss or

✉Corresponding author.

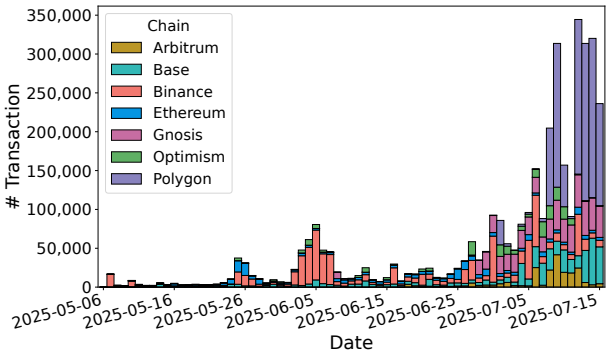

Figure 1: Daily SAs upgrades across seven EIP-7702-enabled blockchains. (# Transaction denotes the daily number of EIP-7702 authorization transactions.)

unauthorized operations. On the other hand, prior to the Pectra upgrade, CAs were designed to interact only with EOAs or other contracts. The introduction of SAs breaks this security assumption. Consequently, existing contracts may misidentify SAs as EOAs, allowing attackers to bypass access control mechanisms with malicious SAs [38]. To date, the security implications of SAs have not been systematically examined in existing research. Real-world attacks leveraging SAs also remain largely unexplored from a systematic perspective. Nevertheless, multiple real-world incidents exploiting EIP-7702 SAs have already been observed and reported [6, 29], affecting several decentralized finance (DeFi) projects and causing approximately one million USD losses.

In this paper, we present the first empirical study on risks introduced by SAs. To systematically examine these risks, we classify the risks according to the type of victim accounts, including: EOA-targeted, CA-targeted, and Composite attack:

EOA-targeted attack: This attack leverages missing validation during EIP-7702 authorization, allowing attackers to link user EOAs to malicious contracts. This enables unauthorized actions, such as transferring assets or making arbitrary calls, within the victim's account.

CA-targeted attack: Attackers take advantage of upgraded SAs retaining their EOA identity (`tx.origin`), making SAs appear as normal EOAs. This allows them to bypass legacy smart contract security assumptions, e.g., `msg.sender == tx.origin`, enabling contract-to-contract interactions that were previously restricted.

Composite attack: It exploits both EOAs and CAs. Attackers can construct EOA-targeted attacks on victim EOAs who hold admin privileges over smart contracts. The attacker inherits the victim admin identity and gains indirect control over EOA-owned contracts, thereby extracting contract assets.

We illustrate these three attack types with real-world code examples, demonstrating how SAs redefine blockchain trust boundaries and enable real-world attacks.

To uncover these risks in practice, we develop detection tools to identify real-world malicious attacks in EOA-targeted, CA-targeted, and composite scenarios. Each tool adopts a unified architecture with three components: a *Transaction analyzer* that extracts suspicious transactions from large-scale blockchain data, a *Decompiler* that reconstructs semantic facts from contract bytecode, and a *Contract analyzer* that performs cross-chain static analysis to detect malicious patterns. To validate the effectiveness of these tools, we build a comprehensive dataset covering all seven mainstream blockchain platforms that support SAs: Ethereum [23], Binance [9], Polygon [44], Optimism [43], Arbitrum [3], Base [5], and Gnosis [28], covering over 22,841,000 real-world historical transactions and 3,664,166 EIP-7702 authorization transactions before July 15th, 2025. Following this pipeline, we systematically identify malicious contracts across these seven blockchains, confirming 793 malicious contracts in EOA-targeted attacks, 124 in CA-targeted attacks, and 7 in composite attacks, resulting in a total asset loss of 2,362,848 USD. We report the 10 most popular 0-day attack addresses that we first revealed and further double-checked by other security companies [25, 26]. These results demonstrate the existence of real-world SA risks and highlight their impact across multiple blockchain ecosystems.

Finally, we conduct an in-depth analysis of the detection results, focusing on transaction behaviors and code patterns observed in real-world attacks. Among our findings, we observe that over 63% of EIP-7702 authorization transactions involve malicious EOA-targeted attacks rather than legitimate uses, and that nearly half of the top 30 most popular authorized contracts for EIP-7702 upgrades are malicious. We find that most attackers do not employ sophisticated techniques such as code obfuscation or contract-level concealment, with some even exposing explicit malicious keywords in function names. However, a subset of attackers evade detection by rebinding their SAs to benign contracts after exploitation, which necessitates analyzing full authorization histories rather than relying solely on current on-chain states. In addition, the introduction of EIP-7702 has unintentionally broken the security assumptions of many vulnerable contracts, placing assets worth approximately 10,140,000 USD at risk. These findings further demonstrate the urgency and importance of our work in filling the research gap on EIP-7702 security and in developing effective detection approaches. In summary, the main contributions of this paper are as follows:

- We present the first empirical study of security risks introduced by EIP-7702 SAs, systematically defining three real-world attack categories: EOA-targeted, CA-targeted, and composite attacks.
- Through large-scale cross-chain analysis, we detect 924 malicious smart contracts and first report the 10 most

popular previously unknown zero-day attackers.

- We reveal key insights into attacker behaviors, detection-evasion techniques, and vulnerabilities in legacy contracts, demonstrating how the programmability of EIP-7702 can be abused as a novel attack surface.

- We publish the first public cross-chain dataset of EIP-7702 attacks, establishing a foundation for future research on detecting and reducing SA risks.

2 Preliminaries

2.1 External Owned Account (EOA)

EOAs are the most common accounts in the blockchain system. EOAs are directly controlled by users, and each EOA is managed by an off-chain cryptographic key pair, including a private key for signing transactions and a public key used to identify the account address [19]. EOAs can be generated from cryptographic algorithms alone without requiring additional identity information, which provides users with some level of anonymity. However, this property also facilitates malicious activities, as attackers can easily create anonymous EOAs for malicious purposes [48]. EOAs contain two on-chain states: the balance and the nonce. The balance denotes the amount of native cryptocurrency (e.g., ETH in the Ethereum blockchain) held by the account, which can be used for value transfer and gas payment. The nonce is a sequential counter that increases with each transaction sent from the EOA, ensuring transaction ordering and preventing replay attacks.

2.2 Contract Account (CA)

CAs [20] are special-purpose accounts in the blockchain system. Like EOAs, they maintain a balance and nonce, but they also include immutable code (i.e., the smart contracts) and persistent storage. Contracts in CAs are typically written in high-level programming languages (e.g., Solidity) and compiled into bytecode for deployment. Once deployed, a contract's public functions can be invoked by EOAs or other contracts to provide services such as lending, swaps, and other DeFi primitives. Additionally, CAs can be utilized to implement tokens on the blockchain. Unlike protocol-level assets, tokens are defined entirely by contract code and storage that specify their rules and behavior. To promote interoperability, Ethereum standardizes token interfaces via Ethereum Improvement Proposals (EIPs) [22]. A canonical example is EIP-20 [50], which specifies the interface for fungible tokens, including balance queries, transfers, and approvals for third-party spenders, enabling broad compatibility across wallets, DeFi, and infrastructure.

2.3 EIP-7702 Smart Account (SA)

The EIP-7702 proposal [10] introduces SAs by enabling an authorization transaction type that permanently sets the code to an EOA, thereby upgrading it into a programmable account that supports delegated contract execution.

EIP-7702 authorization transactions are identified by a new transaction type `SET_CODE_TX_TYPE(0x04)`, which includes an additional payload field named `authorization_list`, which contains one or more authorization tuples. Each authorization tuple consists of `(chain_id, address, nonce, y_parity, r, s)`. Specifically, `chain_id` denotes the blockchain platform to prevent cross-chain replay attacks, while `address` denotes the target address of the smart contract code. The fields `r`, `s`, and `y_parity` jointly encode the ECDSA signature that proves EOA's authorization intent.

Once authorized, the code of the EOA is updated to a special byte sequence: `(0xef,0100, address)`. This sequence encodes a system-level delegate proxy, where `0xef` is a prefix indicating a delegated execution marker, `0100` specifies the version number, and `address` denotes the address of the target contract. Instead of embedding code logic directly into the EOA, this approach allows the account to delegate all incoming calls to the specified contract within the original EOA context, such as its address and balance. Notably, the SA can revert to the EOA by setting the address to the zero address[✉], restoring to a code-less state.

2.4 Threat Model

In this paper, we consider a threat scenario in which an attacker leverages EIP-7702 to exploit the upgrade mechanism of SAs for malicious purposes. Our threat model specifies the attacker's objectives, capabilities, and key assumptions regarding the blockchain platform, covering EOA-targeted, contract-targeted, and composite attack situations.

Attack Scenario: In this scenario, the adversary is a malicious blockchain user who exploits the EIP-7702 upgrade mechanism of SAs. The adversary can either upgrade their own EOAs or craft malicious EIP-7702 authorization requests targeting benign EOAs, which helps the adversary indirectly gain control over victim EOAs and smart contracts and execute unauthorized behaviors.

Adversary's Objectives: The adversary's goals can be summarized as follows: *(i) Account Hijacking*. The adversary aims to upgrade their malicious EOAs or other benign EOAs into SAs with malicious logic, thereby gaining control over the target EOAs or CAs. *(ii) Asset extraction*. With hijacked access, the adversary aims to construct and execute malicious transactions to transfer native cryptocurrency or token assets, thereby achieving financial gain.

[✉]The zero address, denoted by `address(0)`, consists entirely of zeros. It lacks an associated private key and code field, commonly used as a placeholder to indicate an invalid address.

Adversary's Capability & Assumption: Adversaries can create an arbitrary number of EOA, SAs, and smart contracts. Adversaries can initiate all types of transactions supported by the blockchain platform, including token transfer, contract creation, contract invocation, and EIP-7702 authorization. In contrast, adversaries cannot directly control any benign EOAs by stealing their private keys. Adversaries can also construct EIP-7702 authorization requests to any other EOA, which attempt to attach an arbitrary contract address to the benign account. We assume that some fraction of these malicious requests will be approved by users (e.g., via phishing, misleading UIs, or social engineering), causing the EOA to upgrade to an SA with attacker-chosen code. Adversaries can deploy smart contracts that interact with DeFi protocols to request flashloans. A flash loan allows the adversary to borrow large amounts of assets with no collateral, because the loan must be repaid within the same transaction. If the repayment fails, the entire transaction will be reversed. By invoking a flashloan request through smart contracts, the adversary can temporarily hold a substantial balance, which may be used for subsequent attack steps before the contract execution completes.

3 Security Risks of Smart Accounts

3.1 Overview

EIP-7702 redefines who can execute with whose authority by allowing an EOA to delegate its execution to contract code while preserving its address. Combining the EIP-7702 documentation [10] with our threat model and documented real-world attack incidents [7, 8, 29], we observe that risks arise along three victim-centered interaction paths: (*i*) harm to the attached victim account (inward), (*ii*) harm to external contracts with which it interacts (outward), and (*iii*) chained combinations of the two. We therefore adopt the following three-way categories of SA-based attacks: EOA-targeted attacks, CA-targeted attacks, and composite attacks.

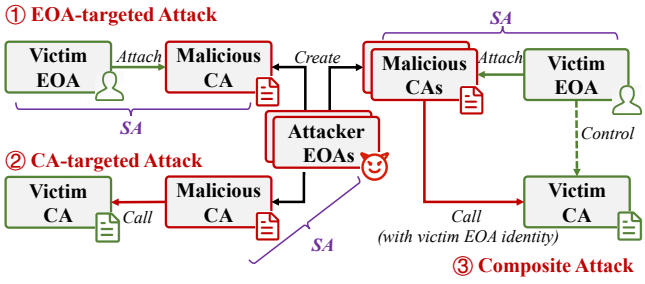

Figure 2: An overview of three SA-based attacks.

As illustrated in Figure 2, each attack category is defined by its unique victim account. EOA-targeted attacks compromise user EOAs by binding them to malicious contract logic, enabling asset theft or unauthorized actions directly

```solidity
contract Wallet {
  //  ... (other logic)
  // execute a batch of calls
  function executeBatch(Call[] calldata calls)
      external onlyOwner {
    for (uint256 i = 0; i < calls.length; i++){
      (bool success, ) = calls[i].to.call{value
          : calls[i].value}(calls[i].data);
      require(success, "Call failed");
  }}
  // accept ETH
  receive() external payable {}
}
```

Figure 3: An example of a benign wallet contract.

from the victim's account. CA-targeted attacks exploit legacy CAs by leveraging SAs with preserved EOA identities to bypass existing security assumptions and invoke restricted functions. Composite attacks represent a more sophisticated threat, chaining both vectors: attackers hijack privileged EOAs and, through them, gain indirect control over sensitive CAs, amplifying the potential impact. In the following sections, we detail the mechanisms behind each attack strategy and provide real-world examples to illustrate their practical implications.

3.2 EOA-Targeted Attack

As discussed in Section 2, EOAs can be upgraded to SAs through EIP-7702 authorization transactions, which link original EOAs to an external contract address. During this process, the blockchain validates only the cryptographic correctness of the signature fields: (`r`, `s`, and `y_parity`), thereby confirming that the transaction was signed by the EOA. However, EIP-7702 imposes no constraints on the `address` field, which designates the contract to be linked. As a result, users must manually verify the validity and trustworthiness of the contract address and its code before proceeding with the upgrade, since authorizing malicious code within the SA context can lead to unauthorized asset transfers.

Importantly, EIP-7702 authorization requests are not limited to transactions that users construct field-by-field in their wallets. Both benign DApps and attackers can prepare partially filled transactions off-chain: they explicitly predefine the authorization_list fields, including the target contract address and initialization parameters, and then present the transaction to the user to sign. In common wallet UIs, users may only see a high-level summary ("upgrade account" or "authorize smart account logic") rather than the raw contract address or its code, which makes it difficult to distinguish a legitimate SA upgrade from a malicious one. In the following, we present two contrasting scenarios of benign and malicious EIP-7702 authorizations, highlighting security threats during SA upgrade.

Benign Scenario: Figure 3 shows a benign contract used for

wallet upgrade. In this case, a wallet software can construct an EIP-7702 upgrade request that binds the user's EOA to a trusted `Wallet` contract. Once authorized, the EOA will be upgraded to an SA with advanced asset management features. As shown in Figure 3, the `Wallet` contract includes a `executeBatch()` function at line 4, which can help SAs execute a batch of external calls in one transaction.

```
1  contract Wallet {
2    address public destination;
3    //  ... (other logic)
4    // execute a batch of calls
5    function executeBatch(Call[] calldata calls)
         external onlyOwner {
6      for (uint256 i=0; i<calls.length; i++){
7        (bool success,) = calls[i].to.call{value
             :calls[i].value}(calls[i].data);
8        require(success, "Call failed");
9    }}
10   // set destination to the thief's address
11   function set_destination(address _thief)
         public {
12     destination = _thief;
13   }
14   // accept ETH and transfer to the
         destination
15   receive() external payable {
16     payable(destination).transfer(msg.value);
17 }}
```

Figure 4: An example of a malicious wallet contract.

Malicious Scenario: In contrast, a malicious contract may closely resemble a benign wallet in structure, but embed hidden logic designed to steal assets. As shown in Figure 4, the contract retains the same `executeBatch()` function, while introducing a `set_destination()` function at line 11 that allows an attacker to set an arbitrary recipient address. The contract then overrides the `receive()` function at line 15, which can automatically forward any received ETH to the recipient address. If an EOA mistakenly authorizes such a contract via an EIP-7702 upgrade, any ETH sent to the account will be immediately transferred to the `destination` address. In the incidents we observed, this authorization typically occurred after users interacted with phishing websites or copied transaction data from untrusted channels (e.g., "airdrop" or "bonus" campaigns). The malicious upgrade transaction appeared superficially similar to a benign wallet upgrade, but the underlying code contained the transfer-on-receipt hook shown above.

This malicious `receive()` function has been observed in real-world attack contracts [24], which have been deployed and subsequently authorized through malicious EIP-7702 upgrade requests affecting over 100,000 victim SAs. This example illustrates how subtle code modifications can transform an otherwise benign wallet into a malicious contract, and the full attack explanation is defined in Section 4.1. In practice, some attackers choose function names and interfaces that mimic popular wallet contracts to blend into normal upgrade flows, while others rely purely on opaque bytecode and social engineering, without attempting to appear as a specific "wallet" implementation in the source view. These mimicked malicious contracts often opt not to publish their source code, which further hinders users from recognizing their malicious logic, thereby increasing the overall difficulty of threat detection in practice.

From External Exploits to In-Context Attacks: Notably, the attack strategy of these malicious contracts diverges from conventional patterns. Traditional malicious contracts are external adversaries: they lure victims into interacting with untrusted code that then extracts value from the victim or from vulnerable third-party contracts (e.g., reentrancy, price oracle manipulation [54]). The attacker's code runs in its own contract context, and theft typically occurs via exploiting call chains or protocol logic outside the victim's account.

By contrast, EOA-targeted attacks under EIP-7702 embed malicious logic directly into the victim SA context. After the victim authorizes an upgrade, the attacker's code executes with the account's own privileges, inheriting its storage, balance, and signing authority. The contract's objective is no longer to exploit external protocols but to exfiltrate the victim's assets by initiating transfers or withdrawals from the upgraded account to attacker-controlled addresses. This "inside-the-account" execution model collapses the boundary between user and contract, reduces observable call indirections, and makes abuses harder to detect. The shift in attack surface highlights the increased severity and subtlety of these risks enabled by new account abstraction.

3.3 CA-Targeted Attack

As a new type of account abstraction, SAs are also capable of directly invoking various smart contracts on the blockchain, just like traditional EOAs. However, as described in the official EIP-7702 specification [10], SAs can fundamentally break long-standing security assumptions when interacting with contracts. In smart contract code, `tx.origin` indicates the original transaction creator, i.e., the EOA that initiates the transaction, while `msg.sender` denotes the immediate caller of the current function. For example, when an EOA creates a transaction to a smart contract A, and contract A subsequently calls contract B, the value of `msg.sender` in the contract B refers to the contract A, while `tx.origin` is the initial account that originally sent the transaction.

Building on this behavior, many smart contracts adopt an assumption `require(msg.sender == tx.origin)` to restrict that a function can only be invoked directly by an EOA, thereby blocking contract-to-contract calls.

Historically, this method was widely used in DeFi protocols to prevent complex interaction chains and reduce the risk of reentrancy and composability-based attacks [56]. With the Pectra upgrade introducing SAs via EIP-7702, the underlying

```
1  contract QuickConverter {
2    //  ... (other logic)
3    // The onlyEOA modifier prevents this from
         being done with a flash loan.
4    function convert(address token0, address
         token1) external onlyEOA() {
5      _convert(token0, token1);
6    }
7    // Try to make flash-loan exploit harder to
          do by only allowing EOAs.
8    modifier onlyEOA() {
9      require(msg.sender==tx.origin,  "
           QuickConverter: must use EOA");
10     _;
11 }}
```

Figure 5: An example of a victim contract with vulnerable `onlyEOA` modifier.

security assumption of this pattern no longer holds: attackers can upgrade their EOAs into SAs and attach malicious contract logic while preserving the original `tx.origin`. The attacker's code executes within the account's context, and interactions that were previously disallowed as contract calls can now occur within a single transaction under the original `tx.origin`. As a result, it is now strongly discouraged in best-practice guidelines. However, we show in Section 5.2 that about a thousand active contracts still rely on `msg.sender == tx.origin` for access control, and that a non-trivial fraction of these contracts remain active and hold significant balances.

Malicious Scenario: As shown in Figure 5, the contract implements a sensitive function `convert(token0, token1)` at line 4, restricted by `require(msg.sender == tx.origin)` at line 9. This check aims to prevent contract-based calls, such as those from flashloan contracts that can manipulate token prices. However, the attacker deployed a contract to initiate a flashloan and redirected the funds to their SA, which bypassed the `onlyEOA()` restriction and successfully executed the `convert()` function. This real-world attack on the Polygon blockchain resulted in a loss of 13,789 WMATIC tokens [8]. Beyond this example, a full explanation of the CA-targeted attack is given in Section 4.3.

3.4 Composite Attack

We have discussed two distinct attack vectors that target EOAs and CAs independently. These two attack vectors can also be combined to form more sophisticated and dangerous exploits.

For EOAs, once a victim EOA signs an EIP-7702 authorization transaction delegating control to a malicious contract, the malicious code obtains full execution privileges within the context of the victim. It can transfer assets directly from the SA and invoke arbitrary external calls, all while preserving the SA's identity as the caller (i.e., msg.sender).

This capability becomes particularly dangerous when the

```
1  contract OwnerControlledContract{
2    //  ... (other logic)
3    address public owner;
4    constructor() {
5      owner = msg.sender;
6    }
7    // Sensitive operation
8    function botsSellCSM(uint256 amount)
         external{
9      require(msg.sender == owner);
10     _sellCSM(amount);
11 }}
```

Figure 6: A contract example with vulnerable ownership.

compromised SA holds privileged roles in other contracts. Many contracts implement role-based access control, commonly assigning the contract creator a high-privilege role that can perform sensitive operations, such as contract upgrades and asset transfers. Attackers can construct EIP-7702 authorization transactions targeting EOAs that possess such roles, thereby executing malicious logic under the victim SA's identity. In effect, the attacker bypasses role checks and executes privileged functions in victim-owned contracts.

Collapse of the EOA-Contract Trust Boundary: This composite attack strategy highlights a critical threat introduced by SAs: adversaries can now compromise and effectively cross the long-standing trust boundary between EOAs and contracts. By hijacking a victim EOA through a malicious EIP-7702 authorization, attackers not only gain complete control over the victim EOAs and corresponding account assets, but also inherit the privileged rights that the victim EOA holds in other smart contracts. This enables adversaries to bypass role-based access controls, and ultimately exploit both the victim EOAs and their dependent contracts in a single unified attack.

Malicious Scenario: As shown in Figure 6, the contract assigns the `owner` variable to `msg.sender` during deployment (line 5). It restricts access to the `botsSellCSM()` function using an access control check at line 9, allowing only the `owner` to invoke this sensitive operation. However, the designated owner later authorized a malicious smart contract address through an EIP-7702 upgrade. Since the SA inherits the original EOA's address, any function call from this upgraded account still satisfies the condition `msg.sender == owner`. As a result, the embedded malicious logic within the SA can bypass the access control and successfully invoke `botsSellCSM()` with a large amount of CSM tokens. Due to the automated market maker mechanism [2], this invocation causes a large tokens to be sold with significant slippage, resulting in the tokens being sold at a substantially reduced price. The attacker can then repurchase the tokens at the discounted rate, effectively profiting from the price differential through price arbitrage. We identified this real-world instance on the Binance Smart Chain [7]. In this case, 11 similar victim contracts were manipulated to sell CSM tokens at low prices,

demonstrating the widespread impact of the attack across multiple contracts. Beyond this example, a full explanation of the attack is given in Section 4.3.

4 Real-world Threat Detection

To enable effective detection of SA-based attacks and potential victims, we define blockchain transactions, which lay the foundation for systematic identification of abnormal behaviors across different attack scenarios.

Transaction Model: We formally define existing transactions, including `Transfer`, `Create`, `Invoke`, and `Attach` transactions. `Transfer` denotes native token transfers between accounts. `Create` refers to the deployment of new smart contracts. `Invoke` represents contract function calls with specific arguments. `Attach` corresponds to EIP-7702 authorization, upgrading an EOA into an SA.

Both EOAs and SAs are capable of initiating all types of transactions. In contrast, smart contracts can only be invoked and execute predefined code logic, which may include external calls that indirectly generate internal transactions to other addresses. We denote such an internal transaction as `call` $\leftarrow$ `invoke`, where `call` represents the external call triggered by the `invoke` transaction. In the following sections, we introduce our detection methods for the three types of attacks.

4.1 Detection of EOA-Targeted Attacks

This section presents our approach to detecting malicious contracts in EOA-targeted attacks. We begin by analyzing their transaction flow and subsequently introduce a detection pipeline to identify corresponding malicious contracts.

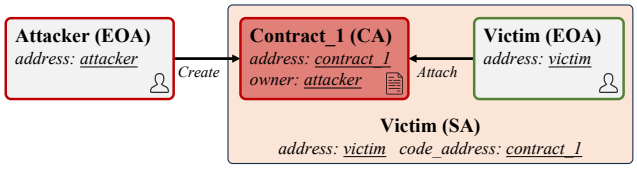

Figure 7: An overview of the EOA-targeted attack process.

Attack Analysis: As shown in Figure 7, we analyze the transaction flow involved in EOA-targeted attacks. Firstly, the attacker EOA initiates a `create(Attacker, Contract_1, malicious_code)` transaction to deploy a malicious smart contract `Contract_1`. Then, the victim EOA approves the malicious EIP-7702 authorization request `attach(Victim, Contract_1)`, and upgrades to an SA with the target code `malicious_code`. The malicious smart contract `Contract_1` typically contains specific hook functions that are automatically triggered upon sensitive events (e.g., a `receive` function). These hooks immediately initiate unauthorized external calls, such as transferring the received assets, thereby enabling automated asset extraction.

Attack Detection: We introduce the EOA-targeted attacker detection tool to identify corresponding malicious contracts in real-world blockchains. This tool consists of three main components: *Transaction analyzer*, *Decompiler*, and *Contract analyzer*. As input, we collect all historical transactions on 7 blockchains (see details in Section 5.1).

The *Transaction analyzer* is responsible for reducing the search space of potentially malicious transactions and contracts, given the enormous volume of historical blockchain data, which exceeds several billion transactions. For EOA-targeted attacks, the analyzer filters the EIP-7702 transactions `attach(from, to)` according to the `authorization_list` field in row transactions. Subsequently, we extract suspicious contract addresses `from` and collect the contract bytecode through the web3 *getCode* API [12].

Then, the *Decompiler* takes the bytecode as input and translates it into an intermediate representation (IR). Similar to previous studies [39, 45], we adopt Gigahorse [31] as the decompiler. It not only performs the translation but also reconstructs rich semantics from the bytecode, including function definitions, internal call relations, and external call statements.

$$
\begin{aligned}
&\textsc{AttackContract}(\textit{pubfunc}, \textit{call}) :- \\
&\quad \textsc{FunctionSelector}(\textit{pubfunc}), \\
&\quad \textsc{FunctionCallEdge}(\textit{pubfunc}, \textit{tracedFunc}), \\
&\quad \textsc{FunctionExternalCall}(\textit{tracedFunc}, \textit{call}), \\
&\quad \textsc{ExternalCallInfo}(\textit{call}, \_, \textit{args}), \\
&\quad \textsc{MatchMalFuncs}(\textit{func}, \textit{funcSig}), \\
&\quad \textsc{MatchMalCalls}(\textit{call}, \textit{args}).
\end{aligned} \tag{1}
$$

Finally, the *Contract Analyzer* takes the decompiled semantic facts as input and applies rule-based pattern matching to identify malicious contract semantics. The rules are derived from the generic execution pattern in Figure 7 and Attack Analysis: once a victim EOA has been upgraded into an SA, the attacker's key objective is to automatically extract any assets that enter the SA, without requiring further user interaction. To succeed at scale, such exfiltration combines two fundamental ingredients: (*i*) a standardized inbound hook that is invoked whenever the SA receives native currency or tokens (e.g., `receive`, `fallback`, or ERC reception callbacks), and (*ii*) a value-moving external call that forwards the received assets to an attacker-controlled address (e.g., via `transfer`, `transferFrom`, or low-level call).

Without a hook that fires at the moment of reception, the attacker would have to rely on additional, manually crafted transactions to receive money from each victim account, which is slower, noisier, and easier to interrupt or detect. Likewise, without an external call that moves value out of the SA, the attacker cannot achieve direct financial gain from the upgrade. Hence, our rules focus on the combination of standardized reception functions and reachable external transfers from those hooks. This combination is not an incidental property of a few examples, but the general pattern makes

EOA-targeted attacks profitable, automatic, and scalable.

As shown in Rules 1, the detection begins by enumerating all public functions of the suspected contract using FUNCTIONSELECTOR, and then constructs the complete intra-program call graph for each public function. The predicate FUNCTIONCALLEDGE captures all reachable functions and statements that may be executed in public function invocations, enabling a precise understanding of its potential runtime behavior. Based on this representation, FUNCTIONEXTERNALCALL identifies all reachable external call statements, while EXTERNALCALLINFO reconstructs the callee and argument information from the corresponding IR. Next, MATCHMALFUNCS compares the function name with predefined hook function names, and MATCHMALCALLS compares the corresponding malicious external call patterns. If any contract matches both criteria, it will be identified as EOA-targeted attack semantics.

As shown in Table 1, we define six malicious hook functions based on popular EIP standards (e.g., EIP-721) as the content matched by MATCHMALFUNCS, and 6 external call patterns commonly used for unauthorized asset extraction as the matching content for MATCHMALCALLS. Our selection of hook functions focuses on those that are either native Ethereum functions [51] or specified by widely adopted EIP standards [16, 17, 18]. These functions cover most standardized asset callback situations, ensuring comprehensive detection coverage and strong generalizability across native cryptocurrency and diverse token standards.

Table 1: Predefined hook functions and external calls.

Hook Function Name	**Usage**
`receive()`	Native transfer reception
`fallback()`	Unmatched call reception
`onERC721Received()`	ERC-721 reception
`onERC1155Received()`	ERC-1155 reception
`onERC1155BatchReceived()`	ERC-1155 batch reception
`tokensReceived()`	ERC-777 reception
External Call Name	**Usage**
`send()`	Native transfer
`transfer()`	Token transfer
`transferFrom()`	Delegated transfer
`safeTransferFrom()`	Wrapped token transfer
`safeTransfer()`	Wrapped delegated transfer
`call()`	Low-level transfer

4.2 Detection of CA-Targeted Attacks

This section presents our approach to detecting malicious contracts in CA-targeted attacks, including transaction flow analysis and a corresponding detection pipeline.

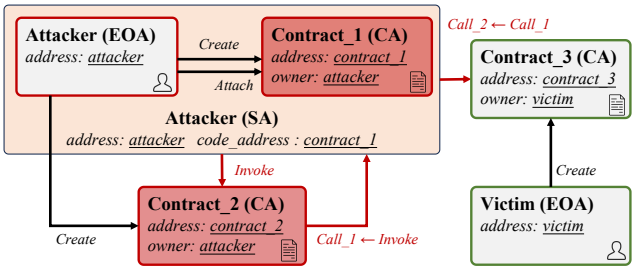

Figure 8: An overview of CA-targeted attack process.

Attack Analysis: As shown in Figure 8, we analyze the transaction flow involved in CA-targeted attacks. Firstly, the attacker `Attacker` initiates two malicious contracts, including `Contract_1` and `Contract_2`. Then, the attacker EOA executes EIP-7702 authorization transaction `attach(Victim, Contract_1)`, and upgrades to an SA. Finally, the malicious SA invokes `Contract_2` for executing the entire attack. In this transaction, `Contract_2` initiates a flashloan and subsequently sends the funds back to the SA (`Call_1` ← `Invoke`). The SA then invokes the victim-owned contract `Contract_3`, thereby bypassing its security checks.

Attack Detection: Similar to EOA-targeted attack detection discussed in Section 4.1, we introduce the CA-targeted attacker detection tool, which also consists of three components: *Transaction analyzer*, *Decompiler*, and *Contract analyzer*.

For CA-targeted attacks, as described in the above Attack Analysis, they have a characteristic three-step structure: (i) the attacker's EOA deploys the contracts that will act as the SA controller (`ca_1`) and the attack orchestrator (`ca_2`), (ii) the EOA upgrades itself into an SA by attaching `ca_1`, and (iii) within a single transaction, `ca_2` obtains short-lived liquidity (typically via a flashloan), routes control back into the SA, and the SA then calls a victim contract that incorrectly assumes it is talking to a plain EOA (`msg.sender == tx.origin`). For such attacks to be practical and capital-efficient, two behaviors are fundamental: (*i*) leveraged liquidity within one transaction, which is overwhelmingly realized today via well-known flashloan protocols (ERC-3156, Aave, etc.), and (*ii*) a concrete callback chain from the flashloan hook in `ca_2` into a public function of `ca_1` (the SA code), and from there into the victim contract. This call chain must exist to make the attack atomic and to preserve `tx.origin` while allowing complex logic to run under the SA's authority.

Our rules therefore search for: (*i*) an EOA that creates both `ca_1` and `ca_2` and then attaches `ca_1` via EIP-7702; (*ii*) a public function in `ca_2` that matches one of several standardized flashloan hook names and issues at least one external call; and (*iii*) a public function in `ca_1` whose signature matches the callback target from `ca_2` and itself performs an external call that can reach a victim contract. These properties derive directly from the economic and technical requirements of the attacks we observed, and they are difficult to bypass

without losing atomicity or leverage. For instance, omitting the flashloan step would remove condition (*i*) but would also require the attacker to provide all capital upfront, making the strategy substantially less attractive in practice.

Hence, as shown in Rules 2, the *Transaction analyzer* executes a SQL-like filter rule for CA-targeted attacks. This rule searches the transaction patterns from the historical transactions, in which a single externally owned account (`eoa`) creates two separate contracts, denoted as `ca_1` and `ca_2`. The contract `ca_1` is subsequently attached to the `eoa` through an EIP-7702 authorization transaction, thereby upgrading the `eoa` into an SA governed by `ca_1`. Meanwhile, the `eoa` also invokes `ca_2`, which typically implements the core attack logic. This transaction sequence reflects a suspected attack strategy, where `ca_1` serves as the control layer for the SA, and `ca_2` acts as the execution layer for carrying out malicious operations. Finally, we label `eoa`, `ca_1`, `ca_2` as suspected addresses.

```
SELECT
  eoa AS suspected_attacker
  ca_1, ca_2 AS suspected_contract
FROM tx_dataset
WHERE Create(eoa, ca_1)
  AND Create(eoa, ca_2)
  AND Attach(eoa, ca_1)
  AND Invoke(eoa, ca_2)
```
(2)

Then, we reuse the same *Decompiler* to process contract bytecode and construct a new cross-contract *Contract Analyzer* for static analysis. As illustrated in Rules 3, the analyzer performs two rule-based matching tasks that analyze the bytecode of `ca_1` and `ca_2`, both created by the same EOA. FLASHLOANCONTRACT rule identifies whether `ca_2` matches the behavioral pattern of the attack contract `Contract_2`. It extracts all public functions using FUNCTIONSELECTOR, and uses MATCHFLASHLOAN to check whether the public function signatures match any predefined flashloan-related hook functions. FUNCTIONEXTERNALCALL collects all external calls in the flashloan function, and *count(call)* != 0 ensures that the matched function performs at least one external call, which may enable a callback to the upgraded SA. The SMARTACCOUNT rule checks whether `ca_1` matches the behavioral pattern of another contract `Contract_1`. It uses MATCHCALLSIG to compare the public function signatures of `ca_1` with the external call targets invoked by `ca_2`, identifying potential callback relationships between the two contracts. In addition, FUNCTIONEXTERNALCALL and *count(call)* != 0 ensure that the matched function also contains an external call, potentially targeting a victim contract.

As shown in Table 2, we collect six representative hook functions used by widely adopted flashloan protocols (e.g., Uniswap V2) and defined in the ERC-3156 flashloan standard. These predefined hook names are used as matching content in MATCHFLASHLOAN, which covers most situations of existing standard flashloan requests.

$$
\begin{aligned}
&\textsc{FlashLoanContract}(\textit{pubfunc}, \textit{call}, \texttt{ca\_2}) :- \\
&\quad \textsc{FunctionSelector}(\textit{pubfunc}), \\
&\quad \textsc{FunctionCallEdge}(\textit{pubfunc}, \textit{tracedFunc}), \\
&\quad \textsc{FunctionExternalCall}(\textit{tracedFunc}, \textit{call}), \\
&\quad \textsc{MatchFlashLoan}(\textit{pubfunc}), \\
&\quad \textit{count}(\textit{call}) \mathrel{!}= 0 . \\
&\textsc{SmartAccount}(\textit{pubfunc}, \textit{call}, \texttt{ca\_1}) :- \\
&\quad \textsc{FunctionSelector}(\textit{pubfunc}), \\
&\quad \textsc{FunctionCallEdge}(\textit{pubfunc}, \textit{tracedFunc}), \\
&\quad \textsc{FunctionExternalCall}(\textit{tracedFunc}, \textit{call}), \\
&\quad \textsc{FlashLoanContract}(\_, \textit{callback}, \texttt{ca\_2}), \\
&\quad \textsc{MatchCallSig}(\textit{callback}, \textit{pubfunc}), \\
&\quad \textit{count}(\textit{call}) \mathrel{!}= 0 .
\end{aligned}
\tag{3}
$$

Table 2: Predefined flashloan hook functions.

Hook Function Name	Protocol
`onFlashLoan()`	ERC-3156 protocol
`callFunction()`	DyDx
`executeOperation()`	Aave
`uniswapV2Call()`	Uniswap v2
`uniswapV3FlashCallback()`	Uniswap v3
`receiveFlashLoan()`	Balancer

4.3 Detection of Composite Attacks

This section presents our approach to detecting malicious contracts in Composite attacks. We also conduct transaction flow analysis and then present a corresponding detection pipeline.

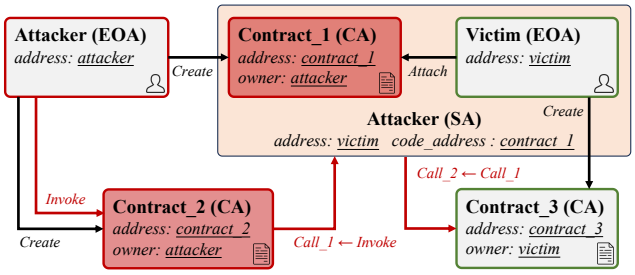

Figure 9: An overview of the composite attack process.

Attack Analysis: As shown in Figure 9, we analyze the transaction flow involved in composite attacks. Firstly, the attacker `Attacker` initiates two malicious contracts, including `Contract_1` and `Contract_2`. Then, the victim EOA approves the malicious EIP-7702 authorization request `attach(Victim, Contract_1)`, and upgrades to an SA with the target code malicious_code. Then, the malicious

SA invokes `Contract_2` for executing the entire attack. In this transaction, `Contract_2` initiates a flashloan and subsequently calls the SA (`Call_1` ← `Invoke`). The SA then invokes the victim-owned contract `Contract_3`, thereby bypassing the ownership check, since the message sender in `Call_2` ← `Invoke` is the victim EOA.

Attack Detection: Similar to previous detection methods mentioned in Section 4.1 and Section 4.2, the composite attacker detection tool also consists of three main components: *Transaction analyzer*, *Decompiler*, and *Contract analyzer*.

Composite attacks chain together the mechanisms of EOA-targeted and CA-targeted attacks. Their defining feature is that the attacker not only drains the compromised SA itself, but also abuses owner privileges that the victim EOA holds in other contracts. To achieve this, three structural conditions must hold: (*i*) a benign EOA (`eoa_v`) has created at least one privileged contract (`ca_v`), (*ii*) the same EOA has later been upgraded into an SA by attaching attacker-controlled logic (`ca_1`), and (*iii*) an attacker EOA (`eoa_a`) has deployed both `ca_1` and an attack contract (`ca_2`) that obtains liquidity via a flashloan, calls back into the SA (`ca_1`), and then causes the SA to invoke sensitive functions in `ca_v` with `msg.sender` still equal to the victim address.

```
SELECT
  eoa_v AS suspected_victim
  eoa_a AS suspected_attacker
  ca_1, ca_2 AS suspected_contract
FROM tx_dataset
WHERE Create(eoa_v, ca_v)
  AND Attach(eoa_v, ca_1)
  AND Create(eoa_a, ca_1)
  AND Create(eoa_a, ca_2)
  AND Invoke(eoa_a, ca_2)
```
(4)

Our composite attack rules therefore enforce: (*i*) that `ca_v` was created by the suspected victim EOA; (*ii*) that `ca_1` and `ca_2` were created by the same suspected attacker EOA; (*iii*) that `ca_1` is attached to the victim's EOA via EIP-7702; (*iv*) that `ca_2` exposes a standardized flashloan hook that performs at least one external call; (*v*) and that there is a chain of matching function signatures from the flashloan hook in `ca_2` to a public function in `ca_1` and from there to a public function in `ca_v`. This structure captures what is fundamental to the success of the real-world composite attacks we observed: hijacking an EOA that owns other contracts, amplifying impact with flashloan liquidity, and then reusing the EOA's original address (now as an SA) to pass admin checks in those contracts. An attacker who avoids this structure, for example, by forgoing flashloans, or by not reusing the victim's admin EOA address, would no longer realize the same "drain both wallet and downstream contracts in one shot" behavior that motivates composite attacks in practice.

Therefore, as shown in Rules 4, the *Transaction Analyzer* executes a SQL-like filter rule for composite attacks. This rule searches the transaction patterns from the historical transactions, in which a suspected victim EOA `eoa_v` creates a smart contract `ca_v` and another suspected attacker EOA `eoa_a` creates at least two smart contracts `ca_1, ca_2`. The contract `ca_1` is subsequently attached to the victim eoa, and the contract `ca_2` is invoked by its creator `eoa_a`.

Then, we reuse the same *Decompiler* and construct a new cross-contract *Contract Analyzer* for static analysis similar to Section 4.2. As illustrated in Rules 5, the analyzer performs three rule-based matching tasks that analyze the bytecode of `ca_1`, `ca_2`, and `ca_v`. The FLASHLOANCONTRACT rule and the SMARTACCOUNT rule are similar to those in CA-targeted attack detection, which identify the attack contract `ca_2` and `ca_2` respectively, and the new rule VICTIMCONTRACT identifies a potential victim contract `ca_v`. It uses MATCHCALLSIG to compare the public function signatures of `ca_1` with the external call targets invoked by `ca_2`, identifying potential invoke relationships between the SA and the user-owned contract.

```
FLASHLOANCONTRACT(pubfunc, call, ca_2) :−
  FUNCTIONSELECTOR(pubfunc),
  FUNCTIONCALLEDGE(pubfunc, tracedFunc),
  FUNCTIONEXTERNALCALL(tracedFunc, call),
  MATCHFLASHLOAN(pubfunc),
  count(call) != 0 .
SMARTACCOUNT(pubfunc, call, ca_1) :−
  FUNCTIONSELECTOR(pubfunc),
  FUNCTIONCALLEDGE(pubfunc, tracedFunc),
  FUNCTIONEXTERNALCALL(tracedFunc, call),
  FLASHLOANCONTRACT(_, callsa, ca_2),
  MATCHCALLSIG(callsa, pubfunc),
  count(call) != 0 .
VICTIMCONTRACT(pubfunc, ca_v) :−
  FUNCTIONSELECTOR(pubfunc),
  SMARTACCOUNT(_, callvictim, ca_2),
  MATCHCALLSIG(callvictim, pubfunc).
```
(5)

4.4 Hypothetical Limitations

While our rules are designed around behaviors that are fundamental to the profit-driven attacks we observe, they are not perfect. We note two hypothetical limitations:

- ***Pre-execution contracts:*** Malicious code without preparation transactions (though impractical for profit-driven attacks). Our method relies on observable transaction evidence of attack preparation, so it may miss malicious contracts before those transactions occur.
- ***Novel mechanisms:*** Unsupported interfaces (we cover the primary standard interfaces in use, i.e., EIP-721, 1155, 777 interfaces [16, 17, 18]). Our method relies on certain interfaces to understand certain code semantics, so it may miss unusual function names.

5 Evaluation and Data Analysis

In this section, we implement our detection tools for EOA-targeted, CA-targeted, and composite attacks across seven blockchains, uncovering real-world attacks. Finally, we conduct an in-depth data analysis and derive security insights.

5.1 Evaluation of Detection Tools

Dataset: We construct three datasets in the evaluation and data analysis process:
① ***Transaction dataset:*** We collect historical transactions from seven blockchain platforms that have supported EIP-7702 SAs, including: Ethereum [23], Binance [9], Polygon [44], Optimism [43], Arbitrum [3], Base [5], and Gnosis [28]. We obtain this dataset from full nodes [36] and online data providers, including Dune [15] and Chainbase [11]. To the best of our knowledge, these seven platforms are the earliest mainstream blockchains that have adopted SAs. The dataset covers all historical blocks before July 15th, 2025, containing 3,664,116 EIP-7702 authorization transactions, each of which corresponds to a single SA upgrade.
② ***Suspected bytecode dataset:*** After the transaction analyzer filters the historical transactions, we obtain a set of suspected contract addresses associated with different types of attacks. For each suspected contract, we obtain the corresponding contract bytecode using the Web3 *getCode* API [12].
③ ***Ethereum source code dataset:*** In order to analyze existing vulnerable contracts, we also constructed an Ethereum source code dataset. Specifically, we scan the Ethereum blockchain transactions since the genesis block in 2016. For each identified contract creation, we try to request the corresponding source code with the Etherscan Explorer *getsourcecode* API. This special dataset enables a systematic evaluation of how adversarials interact with legacy contracts. Since contracts with publicly verified source code are typically deployed by benign developers and contain clearer program semantics, this source code dataset is particularly suitable for large-scale vulnerability analysis.
Experiment Enviroment: We deploy and execute our detection tools for EOA-targeted, CA-targeted, and composite attacks on a machine running Ubuntu 22.04 (ARM64) operating system with 8 CPU cores and 16 GB of RAM. Each detection tool includes three main components: a *Transaction Analyzer*, a *Decompiler*, and a *Contract Analyzer*. We also use this environment in the following data analysis process.
Detection Results: We employ our detection tools across seven blockchain platforms. As shown in Table 3, we begin with the ① ***Transaction dataset*** comprising over 22.8 billion real-world transactions, including 3,664,166 EIP-7702 authorization transactions. This dataset is processed by specialized *transaction analyzers*, each designed to detect a specific type of attack. Through rule-based filtering, the analyzers identify 2,685 EOA-targeted attack suspected contracts, 27,633 CA-targeted attack suspected contracts, and 5,641 composite attacks suspected contracts, which significantly reduces the workload for subsequent code-level analysis.

Based on the filtered transactions, we apply the specialized *contract analyzers* to identify malicious smart contracts associated with each type of attack. As summarized in Table 4, we identify a total of 793 malicious contracts involved in EOA-targeted attacks, 124 in CA-targeted attacks, and 7 in composite attacks, leading to a total of 2,362,848.76 USD asset loss. Among the three categories, EOA-targeted attacks account for the largest number of suspected malicious contracts, with cases identified on all seven blockchain platforms. Notably, Ethereum contains the highest number of such contracts, due to its large user base, deep liquidity, and extensive DeFi ecosystem, which collectively increase the attack surface and incentives for adversaries. In contrast, CA-targeted attacks are less prevalent and are observed exclusively on Polygon, Binance Smart Chain, and Base, because these chains combine lower fees and mature account abstraction tooling that make such exploits more feasible for attackers. Composite attacks, which represent the most sophisticated attack pattern by combining elements of both EOA-targeted and CA-targeted attack behaviors, are relatively rare up to now, and only seven malicious contracts have been identified on Binance.
Effectiveness: To evaluate the effectiveness of our approach, we manually verified all 924 detected malicious contracts: 793 EOA-targeted, 124 CA-targeted, and 7 composite cases. Each contract was independently reviewed by two authors, with any disagreements resolved by a third. The contracts in this set were consistently identified as malicious. To assess potential false negatives, we randomly sampled 500 candidates from the suspected sets produced by each transaction analyzer. These were manually decompiled and analyzed, and the confirmed malicious cases were already included in the final 924 contracts.

This outcome is supported by our multi-stage methodology, which combines transaction-level filtering with code-level static analysis to focus on contracts exhibiting clear suspicious characteristics. The transaction analyzers first narrow the search space by identifying suspicious transactions, while the contract analyzers then apply rule-based pattern matching over decompiled bytecode, targeting well-defined hooks and external call patterns. Furthermore, most attackers do not employ code obfuscation, and many malicious contracts openly reveal their intent in function names, making decompilation and analysis more straightforward (detailed in Section 5.2). Overall, our approach, integrating comprehensive transaction analysis, decompilation, targeted matching, and manual review, supports the reliability of the detection results.
Performance: Since the transaction analyzers process the entire dataset within a few seconds, we focus our detailed reporting on the contract analyzers. During the experiment, we recorded their analysis time across categories. Specifically, the contract analyzer took 8,023.35 seconds to analyze the

Table 3: Cross-chain dataset statistics and suspected transactions identified by the transaction analyzer.

Category	Ethereum	Binance	Polygon	Arbitrum	Optimism	Base	Gnosis	Total
Dataset	2,926M	8,400M	5,479M	1,752M	660M	3,326M	297M	22,841M
EIP-7702	262,986	823,177	1,123,030	208,482	210,187	514,270	522,034	3,664,166
EOA-targeted attack	741	677	116	184	262	649	56	2,685
CA-targeted attack	1,073	1,287	8,309	3,460	3,185	10,266	53	27,633
Composite attack	1,095	922	530	993	814	1,177	110	5,641

Table 4: Cross-chain detected malicious contracts identified by the contract analyzer.

Category	Ethereum	Binance	Polygon	Arbitrum	Optimism	Base	Gnosis	Total
EOA-targeted attack	240	177	36	59	108	159	14	793
CA-targeted attack	0	18	14	0	0	92	0	124
Composite attack	0	7	0	0	0	0	0	7

suspected contracts across three attacks. That said, it requires 0.22 seconds on average per contract for code-level analysis. In comparison, without the transaction analyzers, applying contract analyzers directly to all potential 1,383 million candidates of 22,841 million transactions in the raw dataset would require approximately 304,421,834 seconds (3523 days) with the current experiment setting, which is impractical. This highlights the importance of transaction analyzers in pruning the search space and ensuring that the overall runtime remains manageable. Overall, our detection methodology, integrating transaction and code level analyses to detect real-world risks introduced by EIP-7702 SAs, is highly effective and efficient.

Zero-day Attack: Based on our detection results, we identify multiple previously unreported suspicious EOAs and malicious SAs [33, 35]. To promote early threat awareness, we have reported the top 10 most widely used malicious SAs anonymously. Several of these contracts [25, 26] were double-checked by other security companies after 15 days of our initial alarm, demonstrating that our detection significantly outpaced public responses.

5.2 Data Analysis of Real-world Risks

This section delves into the detection results, highlighting transaction behaviors and code patterns observed in real-world attacks. Our findings reveal several key insights into adversarial behaviors and vulnerable contracts.

Abuse of Smart Accounts: According to the detection results presented in Table 4, we identify 793 malicious authorized contracts involved in EOA-targeted attacks, out of a total of 2,685 unique SAs. This indicates that about 30% of authorized contracts are controlled by adversaries. Furthermore, we observe that these malicious contracts are associated with 2,322,548 EIP-7702 authorization transactions, accounting for 63% of all 3,664,166 authorization transactions in our dataset. This suggests that not only are malicious SAs prevalent, but they are also disproportionately reused in real-world attacks. To further understand this trend, we analyze the top 30 most frequently authorized smart contracts on each chain. As shown in Figure 10, each chain has malicious contracts ranked among its top 30 most authorized contracts. On average, over 12 out of the top 30 contracts per chain are identified as malicious, demonstrating their widespread distribution across ecosystems. Notably, the top-ranked contracts used for EIP-7702 upgrades on Binance, Polygon, and Base are all malicious contracts, underscoring the severity of the threat. These findings indicate that SAs are being widely abused for malicious purposes, rather than serving legitimate use cases.

> ***Finding 1:*** *Over 30% of authorized smart contracts and 63% of EIP-7702 authorization transactions are associated with malicious EOA-targeted attack activity, indicating widespread abuse of the SA mechanism.*

In these malicious EIP-7702 transactions, we identify a type of special authorization transaction in which adversary-controlled EOAs authorize their malicious smart contracts used in EOA-targeted attacks. These transactions likely function as proof-of-concept attempts to validate the efficacy of the malicious attack. Since EIP-7702 SAs were only introduced to major blockchains in 2025, it is plausible that adversaries are also in an early, exploratory stage. The observed attack practice activities suggest that current exploitation remains exploratory and relatively limited in scope.

> ***Finding 2:*** *We find several EOA-targeted attack practice transactions, indicating that current EIP-7702 exploitation remains in an early, exploratory stage.*

Evasion of Attack Detection: During the evaluation, we examine whether adversaries employed evasion methods to bypass potential detection. Specifically, we decompile and ana-

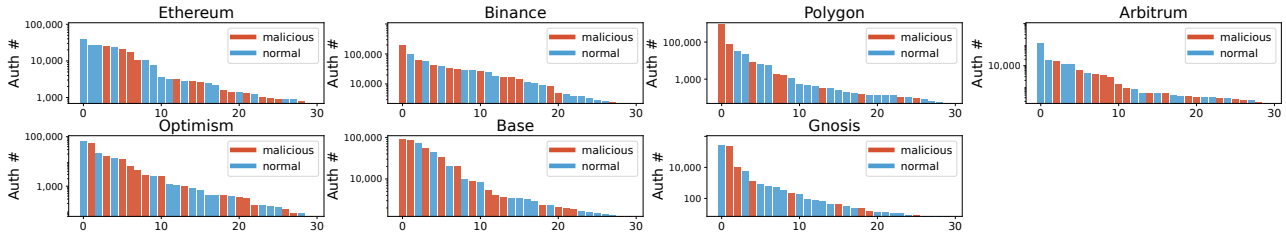

Figure 10: Top 30 authorized contracts by EIP-7702 transaction number, with malicious contracts highlighted.

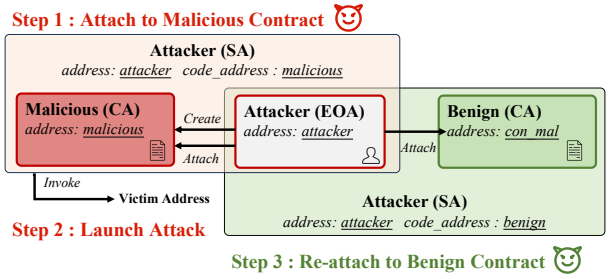

Figure 11: Evade detection by reassigning to benign contracts.

lyze all 35,959 contracts in ② ***Suspected bytecode dataset***. We successfully decompile the bytecode of all contracts, recovering intermediate representations (IRs) from 100% of contracts and public function selector information from 92.7%. The remaining 7.3% mostly consist of short bytecodes, such as proxy contracts [53], that lack sufficient structure to extract function information. This indicates that adversaries generally did not employ code obfuscation to conceal their malicious intent, even for EOA-targeted contracts, which require the victim to approve the malicious EIP-7702 authorization.

We further investigate whether adversaries employed specific types of statements to conceal malicious behaviors and evade code-level detection. In this analysis, we focus on two commonly used mechanisms that could obscure the presence of attack logic: contract creation via the `CREATE` opcode, which allows adversaries to offload malicious code to newly deployed child contracts, and `DELEGATECALL`, which enables execution of external contract code within the context of the caller. Among the SAs in our dataset, we identify 30 instances involving `CREATE` and 148 instances involving `DELEGATECALL`. However, our manual inspection reveals that these cases were associated with benign operations, such as creating sub-wallets, rather than carrying out malicious behavior. Therefore, we find no evidence that existing adversaries systematically relied on such statements to evade detection.

In addition, although bytecode inherently loses certain high-level information such as variable names, public function names can typically be recovered through decompilation and inferred with a publicly available function signature hash database [1]. During our analysis, we successfully extracted the public function names from all decompiled contracts. We then matched the names against a list of seven common attack-related keywords, including `attack`, `steal`, `hack`, `sweep`, `drain`, `exploit`, `pwn`. Among the identified malicious SAs, we found 69 instances containing function names that explicitly suggest malicious intent. This observation indicates that some adversaries did not conceal the purpose during code construction, even though such information can be easily inferred through decompilation techniques and public knowledge.

> ***Finding 3:*** *Many adversaries do not attempt to conceal their malicious intent. Notably, some even include explicit malicious keywords at the source code level.*

However, we also observe that certain adversaries exploit the flexibility of the EIP-7702 mechanism to evade post-incident detection. In CA-targeted attacks, adversarial EOAs attach malicious contracts to their EOAs during attack preparation. However, as shown in Figure 11, we find that certain adversaries reassign their SAs to benign contracts once the attack concludes. This tactic allows the SAs to appear legitimate after attacks, thereby evading detection methods that rely solely on the current authorization relationship of SAs. To overcome this challenge, our detection tools analyze the complete transaction history of each SA. By examining all historical authorization transactions, we can still reconstruct previous bindings between EOAs and malicious contracts.

> ***Finding 4:*** *For CA-targeted attacks, adversaries can rebind SAs to benign contracts after execution, making detection based on current authorization states unreliable.*

Discussion on Potential Evasion via Contract Obfuscation: Although we observe little obfuscation in current malicious contracts, it may become a practical evasion strategy as defenses mature. We therefore discuss the potential evasion via contract obfuscation here.

Impact on Code Obfuscation in Malicious Smart Contracts. From the attacker's perspective, contract obfuscation does not change the underlying EVM execution model, but it can substantially hinder decompilers' ability to recover code semantics, thereby concealing their intentions. Typical obfuscation techniques aim to conceal data flows, control-flow transitions, and code layouts, thereby obscuring key execution semantics [55]. In our setting, this may primarily impact CA-targeted attacks, where our rules rely on specific external-

call code constructs. Heavy control-flow flattening or opaque predicates could degrade the recovered intermediate representation, making it harder to match such patterns reliably.

Limitations on Code Obfuscation in Malicious Smart Contracts. There are also inherent limitations to what obfuscation can achieve in our context. First, the Ethereum ABI exposes a stable interface surface: each public function is identified by a standardized function signature (e.g., `transfer(address,uint256)`), which deterministically maps to a 4-byte selector. While attackers can obscure the internal logic that is executed after the selector is dispatched, they still need to expose compatible selectors and call sequences to interact with existing DeFi protocols and victim contracts, especially for CA-targeted attacks that depend on flashloan callbacks or specific admin functions. Likewise, for EOA-targeted attacks, the malicious SA contracts must still implement recognizable entry points (e.g., `fallback` or `receive`) and perform observable state changes such as large token transfers. These externally visible behaviors are harder to hide without sacrificing attack effectiveness.

Second, deploying high-quality obfuscated EVM bytecode is non-trivial in practice. Existing general-purpose obfuscation tools for high-level languages do not directly target Solidity-to-EVM pipelines, and hand-crafted obfuscation increases development cost and the risk of introducing bugs that could lock attacker funds. Moreover, aggressive obfuscation often incurs additional gas overhead, which can be significant for complex multi-step attacks such as flashloan-based composites. As a result, although obfuscation is feasible, it does not come without cost for attackers.

Future Directions in the Future Arms Race. On the defensive side, one direction is to treat decompilers and IRs as upgradable components: our Datalog rules can be applied over alternative compiler backends [32] with minimal changes, allowing us to benefit from continuous improvements in decompilation quality. Another complementary direction is to further reduce dependence on internal contract semantics and place greater emphasis on stable external signals such as function selectors, inter-contract relationships, and transaction-level behaviors. These signals remain observable even under substantial internal obfuscation and can be naturally incorporated into our current framework as additional predicates. Overall, while sophisticated obfuscation could reduce the recall of some of our current rules, it does not fundamentally prevent us from extending the approach, and we explicitly leave this as a promising avenue for future work.

Potencial Victim Smart Contracts: While we have evaluated real-world attacks, further analysis is necessary to better understand potential victim profiles. Notably, both EOA-targeted and composite attacks require user authorization. Thus, users who do not approve SA permissions are not considered potential victims of these two attack types. In contrast, CA-targeted attacks exploit legacy smart contracts relying on outdated checks such as `msg.sender == tx.origin`. These contracts are inherently vulnerable and can be exploited without any additional user interaction or authorization. Hence, we analyze the ③ *Ethereum source code dataset* for vulnerable contracts. We parse each contract source code into Abstract Syntax Trees (ASTs) and match for the outdated check statements. To this end, we identify 16,993 instances of insecure `tx.origin` checks across 15,020 unique contracts. To provide a clearer justification, we further analyzed `msg.sender == tx.origin` directly used for defense against flashloan attacks. Concretely, we examined all functions in which the `msg.sender == tx.origin` predicate guards the function body and labels a contract as flashloan-defense style if (*i*) the guarded function is externally callable, and (*ii*) its interface suggests security-critical or asset-moving semantics. Results show 8,254 of 15,020 contracts use this ineffective defense pattern. We further analyzed the activity status of these contracts. In the original dataset, 1,385 vulnerable contracts (9.2%) remain active, and 967 contracts (11.7%) remain active in the subset. Notably, these 967 active contracts currently pose 10,140,000 USD in assets to potential high risk. Due to the immutability of blockchain, these vulnerable contracts cannot be further patched after deployment.

> ***Finding 5:*** *For CA-targeted attacks, we find 8,254 potentially vulnerable real-world smart contracts, leaving digital assets worth over 10 million dollars under severe threats.*

Delegated Addresses without Contract Code: In analyzing delegated addresses in EIP-7702 authorization transactions, we expected them to be either deployed contract addresses or the standard zero address, enabling account upgrades or reverting to a conventional EOA. However, we find 500 special delegated addresses without deployed code, which means these addresses can be other EOAs or uninitialized contracts. Attaching to other EOAs effectively achieves the same result as attaching to the zero address, which can revert SAs to a conventional EOA. Another case is when the attached addresses are precomputed `CREATE2` contract address. `CREATE2` [42] allows users to deploy contracts on the precomputed addresses deterministically. An attacker can first convince a victim to authorize a yet-unused CREATE2 address as the SA delegate, and only afterwards deploy arbitrary contract code to that address. From the chain's perspective, the delegated address remains constant across all transactions, but the underlying code can appear or be replaced at any time, meaning the SA's effective behavior can be radically altered without any visible change in the recorded delegated address. Given these risks, attaching to any such non-zero addresses is not recommended.

> ***Finding 6:*** *We identify non-zero delegated addresses without deployed code in EIP-7702 authorization transactions, posing risks as these addresses may be precomputed by `CREATE2` that allows future code deployment.*

Delegated Addresses with Improper Contracts: In analyzing delegated addresses in EIP-7702 authorization trans-

```
1  contract InsecureWallet {
2    address public owner;
3    //  ... (other logic)
4    // constructor intended to set owner
5    constructor() {
6      owner = msg.sender;
7    }
8    // function to set owner
9    function setOwner(address _owner) public {
10     require(owner == address(0));
11     owner = _owner;
12   }
13   // modifier to restrict owner
14   modifier onlyOwner() {
15     require(msg.sender == owner, "...");
16     _;
17 }}
```

Figure 12: An example of an improper wallet contract.

actions, we identify several improper contracts that are not designed for SA upgrades. As shown in Figure 12, the `InsecureWallet` contract initializes its ownership exclusively within the constructor at line 6, assigning `owner = msg.sender` during deployment. This contract should be directly deployed by the user, and the deployer can be automatically assigned as the owner. However, when the contract is attached to an EIP-7702 SA, the constructor will not be executed again. Consequently, the `owner` variable remains unset (`address(0)`), allowing any external caller to invoke the public `setOwner()` function at line 9 to set ownership. Hence, this improper initialization not only leads to a useless upgrade with gas waste but also introduces security risks.

Finding 7: *We observe EIP-7702-upgraded EOAs delegating to legacy wallet contracts that rely on constructor-time initialization. This leaves ownership uninitialized, turning the upgrade into wasted gas and introducing new security risks.*

6 Related Works

Empirical Studies on Blockchain Ecosystems: Several empirical studies have explored security challenges within Ethereum and related blockchain ecosystems. Zhang *et al.* [54] introduced TXSPECTOR, a framework that conducts transaction-level analysis to detect attack patterns in Ethereum systematically, such as reentrancy and integer overflow attacks. Su *et al.* [48] conducted a large-scale study of attacks on DApps, uncovering 476,342 exploit transactions affecting 85 DApps. Yaish *et al.* [52] demonstrated that Ethereum smart contracts allow adversaries to mount denial-of-service (DoS) attacks by exploiting the gas mechanism to impose disproportionate computation costs. Sendner *et al.* [46] systematically evaluated existing tools on four datasets, revealing that even the most accurate tools struggle on real-world benchmarks, highlighting the limitations of existing detection tools. These empirical studies provide valuable insights into the security challenges of blockchain ecosystems. However, existing studies have not systematically evaluated the security risk of SAs.

Smart Contracts Vulnerability Detection Tools: Several tools have been proposed to detect vulnerabilities in smart contracts, including symbolic approaches and static analysis. Oyente [40] is a symbolic execution tool for Ethereum bytecode, detecting reentrancy and ordering issues. Securify [49] extracts a contract dependency graph to extract precise semantic information, demonstrating high correctness in practice. VeriSmart [47] enhances symbolic execution with transaction-invariant inference, achieving highly precise arithmetic verification with negligible false positives across real-world contracts. ContractFuzzer [37] is a fuzzing tool that leverages smart contract specifications to define test oracles, identifying over 459 vulnerabilities among 6,991 tested smart contracts. Slither [27] adopts static analysis and rule-based checking through an intermediate representation, enabling coverage in vulnerability detection. DefectChecker [13] takes a bytecode-level static analysis approach, enabling vulnerability detection directly on EVM code without the need for source code, enhancing applicability to deployed bytecode contracts. Gigahorse [31] enables high-fidelity decompilation for advanced analysis, MadMax [30] builds on Gigahorse to detect gas-related vulnerabilities. Clockwork Finance [4] applies formal verification to detect economic security flaws. SODA [14] supports online monitoring for real-time vulnerability detection, enabling real-time security enforcement.

7 Conclusion and Future Work

In this paper, we conduct the first comprehensive empirical analysis of incurred security risks introduced by EIP-7702-enabled SAs, revealing their programmability as a novel attack surface. We systematically define three real-world attack categories: EOA-targeted, CA-targeted, and composite attacks. Through large-scale cross-chain detection, we identify 924 malicious smart contracts, including several previously unknown zero-day attacks. We observe that most attackers do not employ sophisticated obfuscation techniques, although some attempt to evade detection by rebinding SAs to benign contracts after attacks. The introduction of SAs has already resulted in more than 2 million USD in losses and placed over 10 million USD worth of assets at risk, revealing the urgent need to address these emerging risks.

Future research can investigate emerging SA-based attack strategies, including integrating static analysis with machine learning to enhance detection capabilities. Another promising direction is to translate detection techniques into practical tools, such as explorer integrated plugins. These efforts may enhance the resilience of SAs in the blockchain ecosystem.

Acknowledgments

We thank all reviewers for their constructive comments. The HKUST authors were supported in part by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China HKUST R6005-25, WEB26EG01, and research fund provided by HSBC. Daoyuan Wu's role in this research was partially supported by Lingnan University's Direct Grant (ref. no.: DR26F6) and Faculty Research Grant (ref. no.: SDS25A13).

Ethical Considerations

In this section, we clarify the potential ethical considerations associated with our research. The study has potential implications for several stakeholders, including Web3 project developers, ordinary blockchain users, and the wider Web3 ecosystem. We therefore carefully considered the ethical impact of data collection, disclosure, and long-term community safety.

Stakeholders and Potential Benefits. The primary beneficiaries of our work are web3 project developers and ordinary blockchain users, who can use our findings to identify and mitigate emerging smart account attacks. By analyzing real malicious behaviors, our study enhances collective understanding of evolving security patterns in decentralized finance, aligning with public interest in a safer blockchain ecosystem.

Potential Risks and Mitigations. We acknowledge that revealing malicious contract behaviors could theoretically inform attackers or unintentionally expose benign Web3 project developers. To minimize these risks, we (1) only released raw datasets with all victim information removed, ensuring that no sensitive data is disclosed, (2) sanitized all code snippets by removing addresses and identifiers, and (3) only disclosed hacker contracts whose publication benefits public safety. These steps ensure that the research cannot be exploited to replicate attacks or harm existing projects.

Attack Analysis and Detection. As described in Section 4, our detection process relies solely on publicly available transaction data and smart contract bytecode to identify potential attacker addresses, including malicious EOAs and malicious contracts. We do not disclose any information related to associated victims or that can be used to detect potential victims. In Section 3, we illustrate the smart account threats with code examples and all associated incidents that have already been publicly reported. All code snippets are manually trimmed and sanitized to prevent their use in direct exploitation or the identification of potential victims.

Vulnerability Analysis and Detection. In the findings section, we analyze vulnerable contracts that could be targeted by smart account attacks. To prevent misuse, the original dataset containing smart contract source code is not publicly released. In the detection results we provide, all victim addresses are hashed to obscure their identities, and only non-identifying vulnerable code lines are selectively presented. Although the dataset can theoretically be reconstructed from public sources, its practical feasibility is extremely low. Blockchain explorers (e.g., Etherscan) do not support batch downloads of source code or code-snippet searches, and only the latest 20 contracts can be viewed per query. Accessing a specific contract requires manual browsing, which means the community is already concerned about data misuse. Our source code dataset was collected by continuously running scripts (over more than one year) to monitor verified contracts. Therefore, while the dataset is theoretically reproducible, in practice, it cannot be reconstructed or matched at scale by potential attackers based on the sanitized code snippets. These measures ensure that attackers cannot leverage our results to locate vulnerable smart contracts.

Reflection and Future Guidance. Our analysis serves as a warning to the community about emerging security risks. During the course of the project, we were mindful of the sensitivity of real-world vulnerable contract data and actively took steps to protect potential victims from an ethical standpoint. In future work, we will further engage with security disclosure communities or blockchain foundations at an early stage to help the ecosystem establish proper data sharing and vulnerability mitigation practices. We hope this reflection can inspire future researchers to better balance transparency and responsibility in blockchain security research.

Open Science

Our dataset and corresponding detection code are available at https://zenodo.org/records/17895541[34]. A detailed README file is inside this repository to reproduce our experiments.

References

[1] 4byte.directory contributors. 4byte.directory: Ethereum function signature database. https://www.4byte.directory, 2025.

[2] Hayden Adams. Uniswap: A Fully Decentralized Protocol for Automated Liquidity Provision on Ethereum. https://uniswap.org/whitepaper.pdf, 2018.

[3] Arbiscan. Arbiscan Arbitrum Explorer. https://arbiscan.io, 2025.

[4] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2499–2516, 2023. doi: 10.1109/SP46215.2023.10179346.

[5] BaseScan. BaseScan Explorer. https://basescan.org, 2025.

[6] Binance Attack Reporter. Recent Contract Attacks Exploit EIP-7702 Features, Causing Significant Losses. https://www.binance.com/en/square/post/07-09-2025-recent-contract-attacks-exploit-eip-7702-features-causing-significant-losses-26704391788145, 2025.

[7] BlockSec Explorer. CSM Token Dump Attack. https://app.blocksec.com/explorer/tx/bsc/0x580f504af22ba7626fbf773bec7ef9d3b6732d307800abbd7fa793f6f5c36e94, 2025.

[8] BlockSec Explorer. Flashloan attack on polygon. https://app.blocksec.com/explorer/tx/polygon/0x60d5292aa5dfcc074e0b97d31c919f09bd282c24e138b7079484cf87d5ab11cc, 2025.

[9] BscScan. BscScan Binance Smart Chain Explorer. https://bscscan.com/txnAuthList, 2025.

[10] Vitalik Buterin, Sam Wilson, Ansgar Dietrichs, Matt Garnett, and Nicolas Liochon. EIP-7702: Set code for EOAs. https://eips.ethereum.org/EIPS/eip-7702, 2025.

[11] Chainbase. Chainbase Data Cloud. https://console.chainbase.com/dataCloud/v2, 2025.

[12] ChainSafe. Web3 API Library. https://web3js.readthedocs.io/en/v1.2.11/web3-eth.html#getcode, 2025.

[13] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering*, 48(7): 2189–2207, 2022. doi: 10.1109/TSE.2021.3054928.

[14] 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.

[15] Dune. Dune Analytics. https://dune.com, 2025.

[16] Ethereum. ERC-777: Token Standard. https://eips.ethereum.org/EIPS/eip-777, 2017.

[17] Ethereum. ERC-1155: Multi Token Standard. https://eips.ethereum.org/EIPS/eip-1155, 2018.

[18] Ethereum. ERC-721: Non-Fungible Token Standard. https://eips.ethereum.org/EIPS/eip-721, 2018.

[19] Ethereum. Ethereum Developer Documentation: Ethereum Accounts. https://ethereum.org/en/developers/docs/accounts, 2025.

[20] Ethereum. Introduction to smart contracts. https://ethereum.org/en/developers/docs/smart-contracts, 2025.

[21] Ethereum. Pectra Upgrade Mainnet Announcement. https://blog.ethereum.org/2025/04/23/pectra-mainnet, 2025.

[22] Ethereum. Ethereum Improvement Proposals. https://eips.ethereum.org/, 2025.

[23] EtherScan. EtherScan Explorer. https://etherscan.io, 2025.

[24] Etherscan. Etherscan Address: Crime Enjoyor. https://etherscan.io/address/0x710fad1041f0ee79916bb1a6adef662303bb8b6e, 2025.

[25] Etherscan. AdvancedCrimeEnjoyor. https://etherscan.io/address/0xb6785b782571980b3ddb5d40659f4861ff15aa02, 2025.

[26] Etherscan. AdvancedCrimeEnjoyor2. https://etherscan.io/address/0x89046d34e70a65acab2152c26a0c8e493b5ba629, 2025.

[27] 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.

[28] GnosisScan. GnosisScan Explorer. https://gnosisscan.io/txnAuthList, 2025.

[29] GoPlus Security. Understanding eip-7702 attacks: A comprehensive guide to protection strategies for wallets. https://goplussecurity.medium.com/understanding-eip-7702-phishing-attacks-a-comprehensive-guide-to-protection-strategies-for-wallets-8e8372e3d5ea, 2025.

[30] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. 2(OOPSLA), 2018. doi: 10.1145/3276486. URL https://doi.org/10.1145/3276486.

[31] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1176–1186. IEEE, 2019.

[32] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. Elipmoc: advanced decompilation of ethereum smart contracts. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. doi: 10.1145/3527321. URL https://doi.org/10.1145/3527321.

[33] Mingyuan Huang. Case Explanation of the Exploit. https://x.com/deseclab/status/1930518478450233774, 2025.

[34] Mingyuan Huang. Smart account attack detection tool and dataset. https://zenodo.org/records/17895541, 2025.

[35] Mingyuan Huang. Zero-Day Vulnerability Report. https://x.com/deseclab/status/1930550219957317978, 2025.

[36] Infura. Infura Blockchain Infrastructure. https://www.infura.io, 2025.

[37] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269, 2018.

[38] Kong. In-depth discussion on eip-7702 and best practices. https://slowmist.medium.com/in-depth-discussion-on-eip-7702-and-best-practices-968b6f57c0d5, 2025.

[39] Wenkai Li, Xiaoqi Li, Zongwei Li, and Yuqing Zhang. Cobra: interaction-aware bytecode-level vulnerability detector for smart contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1358–1369, 2024.

[40] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 254–269, 2016.

[41] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[42] OpenZeppelin. Deploying with create2. https://docs.openzeppelin.com/cli/2.8/deploying-with-create2, 2025.

[43] Optimism Blockscout. Optimism Blockscout Explorer. https://optimism.blockscout.com/stats/newEip7702Auths, 2025.

[44] PolygonScan. PolygonScan Explorer. https://polygonscan.com/txnAuthList, 2025.

[45] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. Not your type! detecting storage collision vulnerabilities in ethereum smart contracts. In *Proc. Netw. Distrib. Syst. Secur. Symp*, pages 1–17, 2024.

[46] Christoph Sendner, Lukas Petzi, Jasper Stang, and Alexandra Dmitrienko. Large-scale study of vulnerability scanners for ethereum smart contracts. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 2273–2290. IEEE, 2024.

[47] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1678–1694. IEEE, 2020.

[48] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. Evil under the sun: Understanding and discovering attacks on ethereum decentralized applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1307–1324. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/su.

[49] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 67–82, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356930. doi: 10.1145/3243734.3243780. URL https://doi.org/10.1145/3243734.3243780.

[50] Fabian Vogelsteller and Vitalik Buterin. EIP-20: Token Standard. https://eips.ethereum.org/EIPS/eip-20, 2015.

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

[52] Aviv Yaish, Kaihua Qin, Liyi Zhou, Aviv Zohar, and Arthur Gervais. Speculative Denial-of-Service attacks in ethereum. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3531–3548, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/yaish.

[53] ZeroAge. The more-minimal proxy. https://medium.com/@0age/the-more-minimal-proxy-5756ae08ee48, 2019.

[54] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. TXSPECTOR: Uncovering attacks in ethereum from transactions. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2775–2792. USENIX Association, August 2020. ISBN 978-1-939133-17-5. URL https://www.usenix.org/con

ference/usenixsecurity20/presentation/zhang-mengya.

[55] Pengcheng Zhang, Qifan Yu, Yan Xiao, Hai Dong, Xiapu Luo, Xiao Wang, and Meng Zhang. Bian: Smart contract source code obfuscation. *IEEE Transactions on Software Engineering*, 49(9):4456–4476, 2023. doi: 10.1109/TSE.2023.3298609.

[56] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2444–2461, 2023. doi: 10.1109/SP46215.2023.10179435.