Demystifying and Detecting Cryptographic Defects in Ethereum Smart Contracts

Jiashuo Zhang*, Yiming Shen†, Jiachi Chen†¶, Jianzhong Su†, Yanlin Wang†,
Ting Chen‡, Jianbo Gao§¶, Zhong Chen*¶
*School of Computer Science, Peking University, Beijing, China
†Sun Yat-sen University, Zhuhai, China
‡University of Electronic Science and Technology of China, Chengdu, China
§Beijing Key Laboratory of Security and Privacy in Intelligent Transportation,
Beijing Jiaotong University, Beijing, China
¶Corresponding Authors
zhangjiashuo@pku.edu.cn, seuilping@gmail.com, chenjch86@mail.sysu.edu.cn, sujzh3@mail2.sysu.edu.cn
yanlin-wang@outlook.com, brokendragon@uestc.edu.cn, gao@bjtu.edu.cn, zhongchen@pku.edu.cn

***Abstract*—Ethereum has officially provided a set of system-level cryptographic APIs to enhance smart contracts with cryptographic capabilities. These APIs have been utilized in over 10% of Ethereum transactions, motivating developers to implement various on-chain cryptographic tasks, such as digital signatures. However, since developers may not always be cryptographic experts, their ad-hoc and potentially defective implementations could compromise the theoretical guarantees of cryptography, leading to real-world security issues. To mitigate this threat, we conducted the first study aimed at demystifying and detecting cryptographic defects in smart contracts. Through the analysis of 2,406 real-world security reports, we defined nine types of cryptographic defects in smart contracts with detailed descriptions and practical detection patterns. Based on this categorization, we proposed CRYSOL, a fuzzing-based tool to automate the detection of cryptographic defects in smart contracts. It combines transaction replaying and dynamic taint analysis to extract fine-grained crypto-related semantics and employs crypto-specific strategies to guide the test case generation process. Furthermore, we collected a large-scale dataset containing 25,745 real-world crypto-related smart contracts and evaluated CRYSOL's effectiveness on it. The result demonstrated that CRYSOL achieves an overall precision of 95.4% and a recall of 91.2%. Notably, CRYSOL revealed that 5,847 (22.7%) out of 25,745 smart contracts contain at least one cryptographic defect, highlighting the prevalence of these defects.**

***Index Terms*—Ethereum, smart contracts, defects detection, cryptography**

I. INTRODUCTION

Cryptographic techniques, with their strong capabilities in securing data and communication, have demonstrated significant potential in enhancing the functionality of smart contracts [1]–[3]. To prompt on-chain cryptographic practice, Ethereum has officially introduced a set of system-level cryptographic APIs [4], such as `ECRECOVER`, to enable basic crypto operations within smart contracts. These APIs effectively reduced the gas cost associated with complex cryptographic operations and prompted diverse on-chain cryptographic tasks such as *digital signature* [2] and *Merkle proof* [5]. Currently, more than 10% of Ethereum transactions use these crypto APIs [1], highlighting the significance and prevalence of cryptographic practices in Ethereum smart contracts.

However, since smart contract developers may not necessarily be cryptographic experts, their implementation of cryptographic tasks could be error-prone. Such defective implementations can compromise the theoretical security guarantees of cryptography and lead to real-world security issues in practice [6]–[8]. For example, a security team reported 52 smart contracts that suffered signature replay attacks [6], illustrating the prevalence and damage of on-chain cryptographic defects.

Unfortunately, the community still lacks knowledge and tools to mitigate this threat. A recent empirical study [1] revealed that 56.3% of smart contract developers face obstacles in securing their cryptographic implementations, and 68.1% of developers believe existing security tools need improvement to support their cryptographic practices. Although many studies have focused on defects in smart contracts [9]–[12], they mainly focus on issues arising from general programming tasks, such as *Reentrancy* [10] and *Integer Overflow* [11], while rarely addressing defects specific to cryptographic practices. Consequently, the characterization and mitigation of cryptographic defects remain an open challenge.

To bridge the gap, we conducted the first study focusing on demystifying and detecting cryptographic defects in Ethereum smart contracts. To propose the definition and categorization of common cryptographic defects, we conducted an empirical study on 2,406 smart contract security reports from real-world security teams and investigated crypto-specific security issues they reported. Based on an open-card sorting approach [13], we introduced the first systematic taxonomy of cryptographic defects in smart contracts. It includes nine categories of defects, covering common on-chain cryptographic tasks [1], including *digital signature* [14], *Merkle proof* [5], *message digest* [15], and *random number generation* [16].

Based on our defect definitions, we proposed CRYSOL, a fuzzing-based approach to detect cryptographic defects in real-world smart contracts. To the best of our knowledge, it is the first security technique targeting crypto-specific defects in contracts. It integrates offline analysis with on-chain historical data to address the challenges posed by

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00010

complicated cryptographic operations. Specifically, CrySol employs transaction replay and dynamic taint analysis to initialize the fuzzing context and extract essential crypto-related semantic information, such as data dependencies of cryptographic operations. CrySol utilizes a set of crypto-specific strategies to effectively generate test cases and exploit defects. These strategies guide CrySol's test case generation with fine-grained semantic information and prevent it from getting stuck on trivial test cases, *i.e.*, transactions directly reverted by cryptographic checks. CrySol executes the test cases and detects defects based on a set of crypto-specific oracles. To evaluate CrySol's effectiveness, we collected a dataset containing 25,745 real-world crypto-related contracts and ran CrySol on it. The results indicated that CrySol achieves an overall precision of 95.4% and a recall of 91.2%. Moreover, they demonstrated the prevalence of cryptographic defects in real-world contracts, revealing that 5,847 (22.7%) out of these 25,745 contracts contain at least one defect.

We summarize our main contributions as follows:

- We conducted the first study on cryptographic defects in smart contracts. Through the analysis of 2,406 security reports, we defined and categorized nine types of cryptographic defects, which expands the existing categorization of smart contract defects [9], [17]. We presented these defects with detailed descriptions and practical detection patterns to guide future security solutions.
- We proposed CrySol, the first tool to detect cryptographic defects in smart contracts. It extracts fine-grained cryptographic semantics from on-chain data and employs crypto-specific strategies to guide the fuzzing process. By addressing the functional gap of existing security tools, it has the potential to secure the emerging on-chain cryptographic practice.
- We collected a large-scale dataset containing 25,745 real-world crypto-related smart contracts and evaluated CrySol on it. CrySol revealed that 5,847 (22.7%) of these contracts contain at least one cryptographic defect, with an overall precision of 95.4% and a recall of 91.2%.
- We published the source code of CrySol, all analysis results, and datasets at https://github.com/Jiashuo-Zhang/CrySol, to provide support to further studies.

II. Background

A. Ethereum Virtual Machine (EVM)

Ethereum Virtual Machine (EVM) is the execution environment for Ethereum smart contracts [4]. It manages the on-chain states of smart contracts and transforms these states by iteratively executing instructions known as opcodes [18]. The opcodes include stack/memory/storage operations, arithmetic calculations, and other functionalities required by smart contracts. For example, the SLOAD opcode reads a value from the contract's storage to the stack, and the SSTORE opcode writes a stack element to the storage.

Beyond the opcodes, Ethereum introduced several precompiled contracts as *low-level extensions* of EVM [4]. They are implemented as built-in system-level contracts, to optimize the computation cost of specific functionalities, such as crypto operations. User-defined contracts can use the STATICCALL/CALL/CALLCODE/DELEGATECALL opcode to call precompiled contracts and execute their functionalities.

B. Cryptographic APIs in EVM

To enable cryptographic operations in smart contracts, Ethereum introduced nine cryptographic APIs to EVM [1]. These APIs include one opcode (KECCAK256) and eight precompiled contracts (ECRECOVER, SHA256, RIPEMD160, MODEXP, ECADD, ECMUL, ECPAIRING, BLAKE2F). Specifically, KECCAK256, SHA256, RIPEMD160, and BLAKE2F provide four hash functions in smart contracts [4], [19], *i.e.*, KECCAK256 [20], SHA2-256 [21], RIPEMD-160 [22], and BLAKE2b [23]. We collectively refer to these four APIs as hash operations in the remainder of this paper. ECRECOVER [4] facilitates the on-chain verification of ECDSA signatures on the *secp256k1* elliptic curve [24]. MODEXP [25] enables big integer modular exponentiation. ECADD, ECMUL, and ECPAIRING [26], [27] provide elliptic operations of the *alt_bn_128* curve to enable the verification of paring-based zero-knowledge proofs such as Groth16 [28].

These APIs largely reduce the gas cost of cryptographic operations and have thus attracted widespread application. In a recent empirical study [1], Zhang *et al.* found that 13.8% of Ethereum transactions have utilized these crypto APIs. In particular, KECCAK256, ECRECOVER, SHA256 APIs are the top three commonly used APIs, used by 13.0%, 4.96%, 0.56% of transactions, respectively.

C. Cryptographic Tasks in Smart Contracts

Utilizing these crypto APIs, developers have implemented a variety of cryptographic tasks in smart contracts. Zhang *et al.* [1] analyzed the source codes of crypto-related smart contracts and classified common cryptographic tasks in smart contracts, including *digital signatures* (used in 39.4% of crypto-related contracts), *vector commitments* (24.2%), *message digests* (17.4%), and *random number generators* (14.8%). We briefly introduce these tasks as follows:

- *digital signatures*. Signatures are widely used for on-chain identity authentication [2], [29]. By combining ECRECOVER with hash operations, developers can implement signature verification logic for ECDSA signatures [30].
- *vector commitments*. Vector commitments are widely used to enforce on-chain whitelist and other access control policies. They are typically implemented as Merkle proofs [5].
- *message digest*. Message digest [15] refers to the direct use of hash operations. It is commonly used to compute collision-resistant indexes for dynamic-length contents.
- *random number generator*. Random number generator [16] refers to generating pseudo-random numbers based on crypto operations. It is commonly used for on-chain gaming and gambling.

D. Defects in Smart Contracts

A software defect is an error, flaw, failure, or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways [31]. Several previous studies have documented defects in smart contracts from different aspects [9], [17], [32]. For example, Chen *et al.* [9] defined 20 types of defects in smart contract by analyzing StackExchange posts and real-world contracts. These defects impact the security, availability, performance, maintainability, and reusability of smart contracts. With the ongoing innovation in on-chain applications, such as the integration with cryptographic techniques, and the ever-evolving security issues, the understanding and definition of contract defects are also evolving and expanding [17], [33].

E. Security Reports for Smart Contracts

Due to the prevalence of attacks, integrating security evaluations into smart contract development is essential [34]. Many third-party security teams, such as ConsenSys [35] and Trails of Bits [36], offer security analysis services for smart contract projects. They inspect the codes of smart contracts, search for defects, and produce detailed reports for developers. These security reports, with comprehensive descriptions of real-world defects, are ideal information sources for defining smart contract defects.

III. Cryptographic Defects in Smart Contracts

In this section, we conducted an empirical study on real-world security reports to define and categorize common cryptographic defects in Ethereum smart contracts.

A. Data Collection

During this process, we collected security reports from a wide range of real-world security teams. Specifically, Etherscan [37] provides a list of 75 security teams that specialize in smart contract security. By searching on their official websites and accounts on social media platforms like Medium [38], we identified that 31 out of these security teams have publicly available security reports. We manually collected these security reports and obtained a dataset containing 2,406 reports.

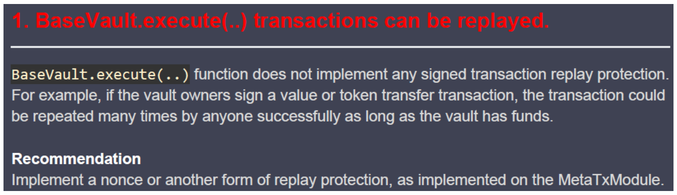
B. Data Pre-processing

To filter out security reports that relate to cryptographic practices, we conducted both keyword-based filtering and manual filtering on the collected reports. During the keyword-based filtering, we utilized terms associated with cryptographic tasks such as "signature" and crypto API names like "ecrecover", as keywords. As a result, we filtered out 893 reports that contained at least one keyword in their content. However, due to the multiple meanings of keywords such as "hash", keyword-based methods are prone to inaccurate identification. For example, the term "hash" in several reports actually refers to the commit hash of the code being audited. We manually checked these reports to remove those with irrelevant keywords. Finally, we collected 211 crypto-related reports, which are available in our online supplementary material [39].

C. Data Analysis

We conducted a manual analysis on the collected 211 reports to investigate the categories of cryptographic defects in Ethereum smart contracts. Due to the exploratory nature of our study, we did not introduce any pre-defined categories of defects. Instead, we employed the open card sorting approach [13], a common approach in software engineering for organizing information into logical groups, to define categories of defects. In line with previous studies [1], [9], [40], we created a card for each report, including detailed descriptions of the defects in the report and the root causes of them. Fig. 1 shows an example of the card for a security report [41]. It describes a defect in the *BaseVault* contract that allows signature replay attacks. The root cause of this defect is the lack of protection against reused signatures, so that signatures in past transactions can be replayed multiple times.

During the card sorting procedure, two authors manually analyzed these cards to define the categories of cryptographic defects. For each card, they first examined its root cause to determine if it could be categorized under an existing category. If not, they evaluated the defect's representability and reproducibility to decide whether to introduce a new category. For example, defects that were highly specific to the business logic of a particular contract were not introduced as new categories. After that, they engaged in discussions to resolve any disagreements and reached a consensus on the results.



1. BaseVault.execute(..) transactions can be replayed.

`BaseVault.execute(..)` function does not implement any signed transaction replay protection. For example, if the vault owners sign a value or token transfer transaction, the transaction could be repeated many times by anyone successfully as long as the vault has funds.

Recommendation

Implement a nonce or another form of replay protection, as implemented on the MetaTxModule.

Fig. 1. An example of the cards of security reports.

D. Defects Definition

Through the analysis of security reports, we identified nine types of cryptographic defects in Ethereum smart contracts, covering all common on-chain cryptographic practices described in Section II-C, *i.e.*, *digital signature*, *vector commitment*, *message digest* and *random number generator*. These defects could compromise theoretical security guarantees offered by cryptography and lead to unintended contract behaviors in practice. Table I enumerates each type of defect along with its definition. In the following, we describe these defects with detailed explanations and illustrative examples.

(1) Single-Contract Signature Replay (SSR). Digital signatures are commonly used in smart contracts for on-chain access control [2], [29]. Transactions with valid signatures can perform sensitive operations in the contracts, such as transferring tokens. In such scenarios, a signature should be invalidated once it is verified, to prevent attackers from replaying the same signature and re-executing sensitive operations. However, with this defect, the contract does not reject these valid but already used signatures. Consequently, they may suffer from signature

TABLE I
DEFINITIONS OF CRYPTOGRAPHIC DEFECTS IN ETHEREUM SMART CONTRACTS

Cryptographic Defect	ID	Definition
Single-Contract Signature Replay	SSR	Do not prevent the same signature from being used multiple times.
Cross-Contract Signature Replay	CSR	Do not distinguish signatures for this contract from those for other contracts.
Signature Front-Running	SF	Allow signatures in pending transactions to be front-run and preemptively used.
Signature Malleability	SM	Lack protection against signature malleability.
Insufficient Signature Verification	ISV	Do not properly check the result of signature verification.
Merkle Proof Replay	MR	Do not prevent the same Merkle proof from being used multiple times.
Merkle Proof Front-Running	MF	Allow Merkle proofs in pending transactions to be front-run and preemptively used.
Hash Collisions With Dynamic-Length Arguments	HC	Do not prevent collisions when hashing concatenated dynamic-length arguments.
Weak Randomness from Hashing Chain Attributes	WR	Use the hash of chain attributes as randomness.

```
1 function permit(address owner, uint256 value,
      uint256 deadline, uint8 v,bytes32 r, bytes32 s)
      external {
2     bytes32 hash = keccak256(abi.encode(owner, value
          , deadline));
3     address signer = ecrecover(hash, v, r, s);
4     require(signer != address(0),"Invalid Signature"
          );
5     require(owner == signer, "Invalid Signer");
6     require(block.timestamp < deadline, "Permit
          Expired");
7     _approve(owner, msg.sender, value);}
```

Fig. 2. An example contract with SSR, CSR, and SF defects

replay attacks: anyone who has observed valid signatures in past on-chain transactions can replay these signatures and pass the signature verification again.

Example: Fig. 2 shows a defective implementation of the ERC-20 permit function [29]. Ideally, this function should allow the *msg.sender* to get approved to spend tokens after submitting a valid signature from the token owner (line 7). However, since this function does not check whether each signature has been used, it always considers signatures used by past transactions as valid signatures. Consequently, it enables the replaying of signatures, allowing *msg.sender* to gain repeated approvals for token spending.

(2) Cross-Contract Signature Replay (CSR). This defect arises when two different contracts have an identical signing domain, *i.e.*, the structures of their signed messages are exactly the same. In such cases, a valid signature for contract *A* will also be valid for contract *B*, enabling cross-contract signature replay. Compared to the SSR defect mentioned above, this defect involves a different attack vector: SSR involves replaying historical signatures that previously used by the victim contract, while this defect involves replaying signatures from other contracts to the victim contract. Both defects could lead to unauthorized access to sensitive operations.

Example: Take the permit function in Fig. 2 as an example. Suppose there are two token contracts, *A* and *B*, each implementing the same permit function. The token owner, holding both tokens *A* and *B*, intends to sign a permit for a spender of token *A*. However, since the signed messages required by token *A* and *B* have exactly the same format (line 2-3), the signature intended solely for token *A* also becomes valid for token *B*. Consequently, a malicious spender can replay the same signature to token *B* and successfully get approved, even though this was never intended.

(3) Signature Front-Running (SF). In Ethereum, pending transactions, *i.e.*, transactions that have been submitted to the network but not yet confirmed in a block, are publicly accessible [42]. Therefore, signatures within pending transactions are susceptible to being captured and preemptively used in a front-running attack [43], [44]. This defect refers to situations where an attack transaction with front-run signatures can successfully pass the verification and lead to unintended contract behaviors.

Example: Consider the permit function in Fig. 2. Normally, a *msg.sender* can obtain the approval by submitting a valid signature. However, in this case, an attacker can intercept the submitted signatures from pending transactions and initiate a front-running transaction to use them preemptively. If the attack succeeds, the new *msg.sender* (*i.e.*, the attacker), instead of the original *msg.sender*, obtains the approval (line 7).

(4) Signature Malleability (SM). The ECDSA signatures supported by `ECRECOVER` precompiled contracts are malleable [45]. Specifically, given a valid signature (v, r, s) for message m, anyone can generate another valid signature(v', r, s') for the same message m [46], [47]. This defect refers to the lack of protection of signature malleability. It is recognized to negatively impact the quality and maintainability of smart contracts [30], [47], [48], and can potentially lead to security issues such as signature replay attacks.

Example: Fig. 3 shows an example in which this defect can cause signature replay attacks. In this case, the hash of the signature is used to prevent signature replay attacks (line 2-3). Normally, after a signature is first verified, it is marked as *used* (line 6), and any further attempts to use a *used* signature are rejected (line 3). However, due to the signature malleability, an attacker knowing a *used* signature can generate a valid but unused signature for the same message. Since the newly generated signature has not been marked as *used* before, it can pass the check at line 3 and make a transfer again (line 7).

(5) Insufficient Signature Verification (ISV). Unlike standard signature verification process, which takes both the public key and the signature as input and indicates the signature's validity with a true/false output, `ECRECOVER` employs the *public key recovery* process [49] for signature verification, which only takes the signature as input and outputs the on-chain address of the "expected" signer. As a result, when

```
1 function transferWithSig(address to, uint256 value,
     uint8 v, bytes32 r, bytes32 s) public {
2    bytes32 sigHash = keccak256(v,r,s);
3    require(!Used[sigHash]);
4    address signer = ecrecover(keccak256(abi.
         encodePacked(to, value, address(this)),v,r,s
         ));
5    require(signer == owner);
6    Used[sigHash] = true;
7    transfer(to, value);
8 }
```

Fig. 3. An example contract with the SM defect

encountering an invalid signature, ECRECOVER still returns an incorrect "expected" signer, instead of reverting the transaction. Additionally, it simply returns zero if the signature is improperly formed [4]. Therefore, when calling ECRECOVER, contracts must check whether the returned "expected" signer is correct according to the business logic, *e.g.*, by checking if it matches the token owner's address. This defect arises when contracts do not properly verify ECRECOVER's return value, leading to unintended contract behaviors.

Example: Fig. 4 illustrates an example of this defect. The intended behavior is to check the managers' signature before permitting an operation. However, attackers can submit a non-existent *opType* and an improperly formed signature to make ECRECOVER return zero. Since *Manager[opType]* also defaults to zero for keys that don't exist, the attacker can successfully pass the signature verification (line 2-3) and gain unauthorized permission (line 4).

```
1 function permitOperation (address opType, uint256
     opID, uint8 v, bytes32 r, bytes32 s) public {
2    address signer = ecrecover(keccak256(opType,
         opID), v, r, s);
3    require(signer == Manager[opType]);
4    permitted[opID]=true;
5 }
```

Fig. 4. An example contract with the ISV defect

(6) Merkle Proof Replay (MR). Merkle proofs are commonly employed to support on-chain whitelists and enable scenarios such as token airdropping [1]. Given a large set of users to be authorized, the contract owner can create a Merkle tree off-chain, distribute its leaves to the users, and upload the Merkle root in the contract [50]. Then, users can submit their leaves and the corresponding Merkle proofs to the contract. The contract will verify the Merkle proof before allowing users to do sensitive operations, such as minting NFTs. Similar to signature replay attacks, lacking protection against Merkle proof replay can cause repeated/unauthorized access to sensitive operations.

Example: As illustrated in Fig. 5, the contract allows whitelisted users to mint tokens (line 2) by submitting a valid Merkle proof (line 3). However, due to this defect, users, even those not in the whitelist, can replay past Merkle proofs submitted by whitelisted users and mint tokens.

(7) Merkle Proof Front-Running (MF). This defect is similar to the *Signature Front-Running* defect. It allows attackers to capture Merkle proofs in the pending transactions and use them preemptively, which could enable unauthorized users to perform sensitive operations in the contract.

```
1 function mint(string memory leaf, bytes32[] calldata
      merkleProof) external {
2    if (MerkleProof.verify(merkleProof, merkleRoot,
         keccak256(abi.encodePacked(leaf)))){
3     _mint(msg.sender, 1); }
4 }
```

Fig. 5. An example contract with MR and MF defects.

Example: The function in Fig. 5 also has this defect. Specifically, anyone observing a pending Merkle proof can launch front-running attacks and preemptively mint tokens to their accounts.

```
1 function addUsers(address[] calldata admins,address
      [] calldata regularUsers, bytes calldata
      signature) external {
2    bytes32 hash = keccak256(abi.encodePacked(admins
         , regularUsers));
3    address signer = hash.recover(signature);
4    require(signer == owner);
5    _addUser(admins,regularUsers)
6 }
```

Fig. 6. An example contract with the HC defect.

(8) Hash Collisions With Dynamic-Length Arguments (HC). Crypto hash operations are expected to be collision-resistant [51], *i.e.*, it is computationally hard to find two input a and b, *s.t.*, $a \neq b \wedge hash(a) = hash(b)$. However, non-standard practice when hashing dynamic-length arguments, *i.e.*, dynamic arrays in Solidity [52], could lead to "collisions".

Example: Fig. 6 demonstrates this defect. Specifically, the built-in function *abi.encodePacked* (line 2) packs all elements in order regardless of whether they're dynamic-length. Therefore, KECCAK256 (*abi.encodePacked* (["0xa", "0xb"], ["0xc"])) is equal to KECCAK256 (*abi.encodePacked* (["0xa"], ["0xb", "0xc"])), leading to a collision. Consequently, attackers can rearrange the addresses in *admins* and *regularUsers* arrays, without changing the hash result (line 2). The signature verification still passes, but the content of these arrays and the contract's behavior (line 5) have been altered.

(9) Weak Randomness from Hashing Chain Attributes (WR). Randomness is commonly used in scenarios such as on-chain gaming and gambling [1]. However, since there is a risk that miners could manipulate chain attributes such as *block.timestamp* to their advantage [53], generating random numbers by hashing chain attributes can compromise the security of these applications.

Example: Fig. 7 provides an example where this defect can be exploited to gain profits. By choosing a *block.timestamp* that meets the condition (line 4), a malicious miner can win the gambling game and receive the rewards (line 5).

Defect vs. Vulnerability vs. Bug. We use the term *defect* to collectively refer to the issues in cryptographic practices. Compared to other terms such as *vulnerability* and *bug*, *defect* has a wider scope [9], [31], thus better representing these issues. Specifically, *vulnerability* refers to defects that can be directly exploited, while excluding other non-standard

```
function gamble() public payable {
    require(msg.value == 1 ether);
    uint8 rand = uint8(keccak256(block.timestamp,
        block.number))
    if (rand == 0) {
        msg.sender.transfer(2 ether);
}}
```

Fig. 7. An example contract with the WR defect.

cryptographic implementations. For example, while *Signature Malleability* negatively impacts the quality and maintainability of the contract, it does not necessarily constitute a *vulnerability*: it can only be directly exploited in certain cases like Fig. 3. Furthermore, *bug* pertains to defects caused by coding errors. However, defects like *Single-Contract Signature Replay* are often a result of design flaws, *i.e.*, the absence of a replay protection scheme, rather than coding errors.

IV. Methodology

Our results in Section III demonstrate nine defects of on-chain cryptographic practices. To provide real-world evidence of these defects in Ethereum smart contracts and assist developers in detecting them in practice, we built CrySol, an automated testing tool for Ethereum smart contracts.

A. Design Decisions

CrySol is built on fuzzing, a plausible technique to detect defects in contracts [53], [54]. Compared to techniques like symbolic execution [55], it can scale better to find defects with deep program paths and complex computations. However, when applying fuzzing to crypto-related contracts, the inherent complexity of cryptography introduces new challenges. In the following, we introduce these challenges and describe the design decisions we made to address these challenges.

Properly Initializing the Fuzzing Context. Crypto-related functions often involve intricate execution contexts. For example, to successfully call the function in Fig. 5, the storage variable *merkleRoot* needs to be properly initialized, and the transaction should include a valid Merkle proof pertaining to that specific *merkleRoot*. Common solutions, such as randomly initiating the contracts' states and transactions, may result in test transactions being trivially reverted by these cryptographic checks. To overcome this, CrySol utilizes real-world contracts' states and transactions to initialize the execution context of the fuzzing engine. By integrating offline analysis with on-chain data, CrySol provides a fuzzing context that is closer to real-world conditions, thereby improving the effectiveness and efficiency of the fuzzing process.

Effectively Generating Test Cases. Generating test cases that can exploit cryptographic defects requires certain guidance. For example, to exploit the SSR defect in Fig. 2, we need to construct two different transactions with the same signature. However, random methods could be highly inefficient to generate such test cases, since they require identifying which transaction parameters are included in the signed message. To address this, CrySol replays historical transactions of the contract and conducts dynamic taint analysis to extract crypto-related semantics, such as data-dependencies of cryptographic operations. Utilizing these semantics, CrySol employs a suite of crypto-specific strategies to effectively generate test cases that trigger potential defects.

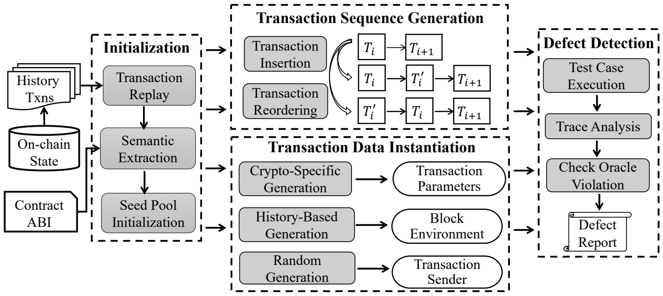


Fig. 8. The workflow of CrySol.

B. Overview

Fig. 8 outlines the overall workflow of CrySol. Given a contract to analyze, CrySol first replays its historical transactions to extract crypto-related semantics and initialize the seed pool (Section IV-C). Then, CrySol starts to generate test cases for the contract to trigger potential defects (Section IV-D). Specifically, the test case generation process involves two steps, *i.e.*, generating the transaction sequence, and instantiating each transaction in the sequence with concrete parameter values. Finally, CrySol executes the test cases and analyzes the execution traces for defect detection (Section IV-E).

C. Initialization

During the initialization, CrySol replays historical on-chain transactions of the contract to collect crypto-related semantic information and initialize the seed pool for fuzzing.

Transaction Replay. CrySol operates an Ethereum archive node [56], which retains all historical state information since the genesis block. For each transaction, CrySol leverages an off-the-chain execution tool [57] designed for transaction replay to extract the contract's pre-state, *i.e.*, the contract state before the transaction execution. Then, CrySol executes the transaction on this pre-state using an instrumented EVM, and gathers execution traces for subsequent analysis.

Semantic Extraction. Based on the execution traces, CrySol collects the following crypto-related semantic information to guide the test case generation processes.

- *Crypto-related functions*. To identify potential execution paths to trigger cryptographic defects, CrySol analyzes the execution traces and filters out functions that used crypto operations. Specifically, for crypto APIs provided as precompiled contracts, CrySol examines the destination address of all contract call opcodes (STATICCALL, CALL, CALLCODE, and DELEGATECALL) to determine whether the transaction calls these crypto APIs. For crypto APIs provided as opcode, *i.e.*, KECCAK256, CrySol analyzes all executed opcodes and checks whether there are crypto calls to KECCAK256. After identifying a function that uses cryptographic operations, CRYSOL records all transactions traces of it for the subsequent data dependency analysis.

- *Crypto-related data dependencies*. CRYSOL employs dynamic taint analysis to extract data dependencies of the cryptographic operations. For example, to determine which transaction parameters may be an ECDSA signature, CRYSOL marks slots of the transaction input data as sources and the arguments of ECRECOVER as sinks. Then, it simulates taint propagation throughout the transaction's execution, checking if the sinks can be reached from the sources. For slots that can reach ECRECOVER, it identifies to which parameters they correspond based on the contract's ABI and then marks these parameters as signature-related. Such information is essential for CRYSOL to generate valid transactions that pass the cryptographic verification.

Seed Pool Initialization. After that, CRYSOL initializes the fuzzing seed pool based on historical on-chain data. Specifically, it includes all historical transactions of the contract as initial seeds. Each seed contains all information required for executing the transaction: (1) the parameters and sender of the transaction; (2) the pre-states of related contracts, including the contract directly called by the external transaction and other contracts called by internal transactions; and (3) the block environment, such as the block number and timestamp.

D. Test Case Generation

With the initialized seed pool and extracted semantic information, CRYSOL begins to iteratively generate test cases. Initially, it selects a seed from the seed pool and sets the contracts' pre-states and block environment recorded in the seed as the starting state for executing the generated test cases. Based on the seed, CRYSOL generates the test transaction sequence and instantiates each transaction with concrete input data.

Transaction Sequence Generation. CRYSOL supports two strategies to generate the transaction sequence, *i.e.*, transaction insertion and transaction reordering. When a seed is chosen, CRYSOL includes the historical transaction from the seed into the initial transaction sequence. Then, by strategically inserting new transactions to the initial sequence and re-ordering them, CRYSOL generates a set of transaction sequences designed to exploit the defects. For example, to exploit the SSR defect, CRYSOL inserts a new attack transaction after the original historical transaction, calling the same function with a replayed signature. To exploit the SF defect, CRYSOL reorders the attack transaction to appear before the original transaction, enabling the front-running use of the signature.

Transaction Data Instantiation. This process is initiated when CRYSOL needs to insert a new transaction into the transaction sequence. Specifically, to instantiate the new transaction, CRYSOL needs to generate three types of concrete data, *i.e.*, the transaction parameters, transaction sender, and the block environment. CRYSOL generates these data based on the following three complementary strategies.

- *Crypto-Specific Generation.* CRYSOL employs a set of crypto-specific strategies to generate crypto-related transaction parameters. Specifically, CRYSOL analyzes crypto-related data dependencies and extracts crypto-related parameters that are used as the input of cryptographic operations. Based on the analysis result, it strategically instantiates these parameters to exploit cryptographic defects. For example, to exploit the SSR or SF defect, CRYSOL needs to construct a new attack transaction containing the same signature as the original seed transaction. To do so, it instantiates signature-related parameters by preserving their values in the original transactions, *i.e.*, simulating the signature replay, while using history-based and random strategies to instantiate non-crypto-related parameters.
- *History-Based Generation.* Given the security implications of cryptographic operations, crypto-related functions might operate within a more subtle context. For example, a randomly selected transaction sender might fail to call the crypto-related functions due to the specific permission structure the contract initialized. To better approximate real-world contexts, CRYSOL offers the ability to instantiate transaction parameters or the transaction sender using values from all historical transactions calling the same function.
- *Random Generation.* In line with previous work [53], [54], [58], CRYSOL infers parameter types based on the contract ABI specification [59] and supports random generation of transaction parameters. Beyond transaction input data, CRYSOL also supports randomly generating the transaction sender and the block environment.

By determining the sequence of transactions and instantiating each transaction with concrete parameter values, sender, and block environment, CRYSOL generates test cases that can be concretely executed to exploit potential defects.

E. Defects Detection

In the last phase, CRYSOL executes the generated test cases and analyzes the execution traces for defect detection. For each test case, CRYSOL instantiates an instrumented EVM with the starting states of the test case, and executes the transaction sequence on it. If the execution violates a pre-defined oracle, CRYSOL reports the identified defect along with the function containing the defect. If the transaction involves multiple contracts, CRYSOL also specifies the contract where the defect occurs by analyzing inter-contract calls during the transaction execution. In the following, we describe the detailed oracles used by CRYSOL to detect each type of defects.

(1) Single-Contract Signature Replay (SSR). CRYSOL examines the transaction sequence and checks whether it contains a successful signature replay attack. Specifically, when a transaction calls ECRECOVER, CRYSOL searches for any subsequent transaction in the sequence that calls ECRECOVER using the same parameters, *i.e.*, replaying the signature. If the transaction with the replayed signature successfully executes and makes changes to the contract storage, CRYSOL reports a SSR defect. Additionally, we found that several token contracts intended to allow token minters to replay signatures and mint tokens until they reach the amount limit per address. To reduce such false positives, CrySol identifies such protective patterns by analyzing transaction execution traces.

(2) Cross-Contract Signature Replay (CSR). CrySol records calls to `ECRECOVER` during the test case execution and checks whether each signed message includes the address of the contract that verifies the signature. Specifically, CrySol taints the return value of opcode `ADDRESS`, which retrieves the contract's address. Then, CrySol monitors if the taint flows into the $hash$ used in `ECRECOVER` $(hash, v, r, s)$. If the signed message does not include the contract's address, *i.e.*, signatures for this contract are not distinguished from those of other contracts, CrySol reports a CSR defect.

(3) Signature Front-Running (SF). CrySol examines the transaction sequence and checks whether it contains signature front-running attacks. Specifically, CrySol identifies cases where, given an original transaction that calls `ECRECOVER`, there exists a preceding attack transaction from a different sender that calls `ECRECOVER` with the same parameters. If so, CrySol conducts a differential analysis on the execution results of the original transaction and attack transaction. It executes them based on the same start state respectively and compares the post-states after execution. If the post-states differ, *i.e.*, the attacker can make unintended changes to the contracts' states, CrySol reports a SF defect.

(4) Signature Malleability (SM). CrySol analyzes whether there is protection against signature malleability. Specifically, when encountering a call to `ECRECOVER` $(hash, v, r, s)$, CrySol analyzes the execution trace and checks whether a branching opcode (`JUMPI`) is executed, conditioning on the comparison between s and the constant elliptic curve order $secp256k1$ [4]. If not, *i.e.*, there is no protection against the signature malleability, CrySol reports a SM defect.

(5) Insufficient Signature Verification (ISV). CrySol checks if there is a transaction that calls `ECRECOVER` with parameters not used in any historical transactions, *i.e.*, the signature is randomly forged by CrySol. If the transaction containing the forged signature successfully executes and makes changes to the storage, CrySol reports an ISV defect.

(6) Merkle Proof Replay (MR). CrySol first identifies the verification process of Merkle proofs based on their operational characteristics. Specifically, CrySol checks whether there is a sequence of hash operations during the transaction execution, where the input of the *i*-th hash is the concatenation of the result of the *(i-1)*-th hash and a proof element provided as the transaction parameters. Then, similar to the detection of SSR, when encountering a transaction that verifies a Merkle proof, CrySol searches for any subsequent transaction that replays that Merkle proof. If both transactions change the contract's storage, CrySol reports a MR defect. To reduce false positives, CrySol identifies the same protective pattern for token minting as in the SSR defect.

(7) Merkle Proof Front-Running (MF). The approach CrySol uses to detect MF defects is analogous to the approach for SF defects. Given an original transaction that verified the Merkle proofs, CrySol searches for any preceding attack transaction that preemptively used the same Merkle proofs. Then, a differential analysis is conducted on the execution results of these two transactions. If the preceding transaction successfully executes and makes different changes to the contract storage, CrySol reports a MF defect.

(8) Hash Collisions With Dynamic-Length Arguments (HC). CrySol conducts dynamic taint analysis on the input of each hash operation to detect HC defects. First, it determines which transaction parameters, if any, serve as input for these hash operations. Then, it checks whether these parameters are dynamic-length based on the contract's ABI. If the hash input contains the concatenation of two dynamic-length parameters, CrySol reports a HC defect.

(9) Weak Randomness from Hashing Chain Attributes (WR). CrySol leverages dynamic taint analysis to check whether the block attributes can affect hash operations. It first taints the returns of opcodes that acquire block attributes (*e.g.*, `NUMBER` and `TIMESTAMP`) and monitors whether the taints flow into hash operations. If there is a hash operation that can be affected by chain attributes and the hash result determines a branch (`JUMPI`) or storage operation (`SSTORE`), CrySol reports a WR defect.

V. Evaluation

The goal of the evaluation is two-fold. Firstly, we utilize a large-scale dataset containing 25,745 crypto-related smart contracts to evaluate the effectiveness of CrySol in defect detection. Secondly, by analyzing the results of this large-scale experiment, we demystify cryptographic defects in the wild and gain insights into their prevalence and distribution.

A. Evaluation Setup

Research Questions. Specifically, we focus on the following three research questions.

- **RQ1.** What is CrySol's performance on our large-scale dataset? Can CrySol find defects with high precision?
- **RQ2.** How effective of CrySol in finding cryptographic defects in terms of recall?
- **RQ3.** What is the prevalence and distribution of cryptographic defects in real-world smart contracts?

Dataset. To answer these research questions, we collected a large-scale dataset containing 25,745 real-world crypto-related smart contracts. Specifically, using the same method as previous studies [1], we first replayed 1,704,224,022 historical Ethereum transactions from block 1 to block 15,500,000 (from 2015.07 to 2022.09) and recorded the contracts that called crypto APIs. In total, we identified 426,296 crypto-related contracts during the execution of historical transactions. After that, we queried Etherscan [60] to collect publicly available source codes and ABI information of these contracts. As a result, we found 25,745 crypto-related smart contracts have available source codes and ABI information. Among these contracts, 83.6% of smart contracts have more than 10 historical transactions, suggesting that the majority of smart contracts in our dataset are engaged in real-world applications, rather than merely being toy contracts.

TABLE II
CRYPTOGRAPHIC DEFECTS DETECTED BY CRYSOL

Defect	# Detected	# Sampled	# TP	# FP	Precision
SSR	151	59	59	0	100.0%
CSR	2,536	93	89	4	95.7%
SF	274	71	69	2	97.2%
SM	1,803	91	89	2	97.8%
ISV	24	20	17	3	85.0%
MR	122	54	48	6	88.9%
MF	33	25	23	2	92.0%
HC	89	46	43	3	93.5%
WR	2,626	93	87	6	93.5%

To retrieve contracts' historical transactions and states, CRYSOL maintained an Ethereum archive node [56] and recorded Ethereum on-chain raw states for subsequent analysis. For each contract in the dataset, CRYSOL fetched its historical transactions and states to initialize the fuzzing seed pool. To ensure efficiency, the maximum seed pool size is set to be 500 transactions, which is considered adequate to cover common usage patterns of the contracts [12], [61]. All experiments were conducted on a machine with two Intel Xeon(R) Platinum 8352V CPUs, 512 GB RAM, and running Ubuntu 22.04.2 LTS.

Our datasets, experiment outputs, and analysis results are all available in the supplementary materials [39].

B. RQ1: Detecting Defects in the Large-Scale Dataset

To answer RQ1, we ran CRYSOL on 25,745 smart contracts and analyzed the results. CRYSOL took 408.0 hours to analyze 25,745 contracts, resulting in an average execution time of 57.1 seconds per contract. In total, CRYSOL reported that 5,847 (22.7%) contracts contain at least one defect. Table II shows a breakdown of CRYSOL's execution results for each defect type.

Precision. To evaluate the precision of CRYSOL in detecting each type of defects, we manually analyzed the defects reported by CRYSOL during the large-scale experiment. In line with previous studies [40], [62], we randomly sampled a number of defects for each defect type to make the manual analysis feasible. The sample size for each defect type was carefully chosen to achieve a confidence level of 95% and a confidence interval of 10. The second and third columns of Table II show the detected and sampled number of contracts with each defect, respectively. Then, two of the authors independently labeled these contracts as true positives (TPs) or false positives (FPs), with the help of the third author to resolve any possible disagreements. The fourth to sixth columns in Table II present the number of true positives, false positives, and the precision rate for each type of defect, respectively. We then computed CRYSOL's overall precision as a weighted average of these precision rates, with the weight being the number of each defect. As a result, the overall precision of CRYSOL is 95.4%.

False Positives. After inspecting the false positives reported by CRYSOL, we found that they are mainly caused by the following two factors. The first is non-standard protective patterns in real-world contracts. For example, for *Signature Malleability*, CRYSOL reported a defect based on whether there is a condition that checks if the input s for ECRECOVER *(hash,v,r,s)* is less than $secp256k1n/2$. However, we found that some contracts used a non-standard protective pattern against signature malleability: they set the first bit of s to 0 before using it as the actual input for ECRECOVER *(hash,v,r,s)*, thereby ensuring that s is less than $secp256k1n/2$. The second is the intended behavior of the contracts. For example, for *Insufficient Signature Verification*, CRYSOL checks whether a transaction with an invalid signature can successfully execute and make changes to the contract's storage. However, we found that some smart contracts do not revert transactions when encountering invalid signatures. Instead, they intendedly record the signature verification results on-chain and continue to execute. In such cases, the transaction with invalid signatures indeed results in the contract's storage changes, letting CRYSOL falsely report an ISV defect.

C. RQ2: Evaluating CRYSOL *on the Annotated Dataset*

To answer RQ2, we built an annotated dataset and evaluated the recall of CRYSOL on it. We have published the annotated dataset and analysis results in our online supplement materials [39].

Recall. The evaluation of the recall requires a dataset with annotations of true positives and false negatives. To establish the ground truth, we first randomly sampled a number of smart contracts from the large-scale dataset and manually annotated them. Specifically, in line with previous studies [40], we randomly sampled 96 out of 25,745 contracts to achieve a confidence interval of 10 and a confidence level of 95%. Then, we followed the same labeling process as Section V-B to manually analyze these sampled contracts. In total, we found 34 defects in these 96 contracts. After comparing these manual labels and the results given by CRYSOL, we found CRYSOL reports 31 true positives, 1 false positive, and 3 false negatives for these contracts, which yields a recall of 91.2%.

False Negatives. In detail, CRYSOL failed to detect one SSR, one CSR, and one WR defect in 96 contracts. After inspecting these false negatives, we found that they are mainly due to the lack of information to properly initialize the fuzzing context. For example, while some smart contracts contain signature verification functionalities, such functions are rarely actually called. Consequently, CRYSOL observed limited semantic information, hindering its ability to generate valid test cases for meaningful exploration. However, automatically generating valid crypto-related transactions with solely off-line analysis is challenging. In particular, cryptographic operations could render common techniques such as concolic testing [63] ineffective, since analyzing them results in complex symbolic expressions that cannot be handled by the SMT solver [64], [65]. Addressing these challenges is beyond the scope of this paper and is left as potential future work.

D. RQ3: Characterizing Cryptographic Defects in the Wild

While demonstrating the effectiveness of CRYSOL, our large-scale experiment also provided a first close look at cryptographic defects in real-world smart contracts.

TABLE III
THE STATISTIC METRICS OF DEFECTIVE CONTRACTS

Type	Prop.(%)	LOC(avg)	#Func(avg)	#ETH(avg)	#Txn(avg)
SSR	0.59%	1369.5	33.8	6.0	8,163
CSR	9.85%	1487.1	30.3	9.3	37,958
SF	1.06%	1466.2	28.5	12.8	76,188
SM	7.00%	1238.4	27.6	5.4	64,906
ISV	0.09%	783.9	26.6	0.1	30,669
MR	0.47%	1747.5	41.9	4.2	1,299
MF	0.13%	1585.1	38.9	5.1	1,544
HC	0.35%	1551.5	29.7	0.8	6,868
WR	10.20%	1352.9	30.7	6.3	3,469
Total	22.71%	1396.8	30.3	7.1	22,930

Prevalence and Distribution of Cryptographic Defects. The first column of Table III presents the proportion of defective contracts regarding each defect type. Among nine types of defects, WR, CSR, and SM are the most common, occurring in 2,626 (10.20%), 2,536 (9.85%), and 1,803 (7.00%) of the analyzed smart contracts, respectively. While the remaining six defect types are less common (appearing in about or less than 1% of contracts), the total number of contracts affected by them is still considerable. Such results provide real-world evidence for the findings of Zhang *et al.* [1], which suggest a lack of understanding of crypto-specific secure practices among smart contract developers. Note that a contract with cryptographic defects indicates deviations from best practices in cryptographic implementations. While defects may not directly lead to security issues, they can undermine the contract's maintainability and increase the risk of future security vulnerabilities. For instance, the CSR defect might not initially cause security problems when only one contract verifies the authorizer's signatures. However, if the system evolves and multiple contracts start using the same authorizer's signatures for managing sensitive operations, this defect can directly enable cross-contract signature replay attacks. A more detailed analysis of these cases is provided in our online supplementary materials [39].

Contracts with Cryptographic Defects. To better understand cryptographic defects in the wild, we analyzed the average lines of code, number of external/public functions, Ether balances, and transaction counts of defective contracts, and presented them in columns three to six of Table III. The result shows that contracts with MR and MF defects are generally more complex than others, likely due to the inherent complexity of Merkle proofs and their applications, such as reward distribution. Furthermore, contracts with SSR, CSR, SF, SM, and ISV defects, are more frequently called by real-world transactions, indicating a broader influence associated with signature-related defects.

VI. DISCUSSION

A. Mitigations for Cryptographic Defects

During the evaluation, we found that cryptographic defects are commonly caused by the direct use of low-level crypto APIs without necessary protection. Therefore, in addition to introducing CRYSOL, we provided possible solutions for each type of defect in Table IV. These solutions are summarized from the standard practices outlined in official Ethereum improvement proposals (EIPs) [2], [29] and defect remediation recommendations in security reports [8], [41], [48], [66].

TABLE IV
POSSIBLE SOLUTIONS FOR CRYPTOGRAPHIC DEFECTS

Type	Possible Solution
SSR	Include a monotonic increasing nonce into the signed message
CSR	Include the contract address into the signed message
SF	Prevent front-run signatures from causing unintended behaviors
SM	Add protection against ECDSA signature malleability
ISV	Check the return value of ECRECOVER before sensitive operations
MR	Check if the Merkle proof has been used before accepting it
MF	Prevent front-run Merkle proofs from causing unintended behaviors
HC	Use collision-resistant encoding to hash dynamic-length variables
WR	Use verifiable random function (VRF) for randomness

For example, Fig. 9 shows a fixed version of the defective contract in Fig. 2. It comes from a standard template [67] provided by OpenZeppelin [68], which employs the above solutions to prevent SSR, CSR, and SF defects. It integrates a nonce in the signed message to prevent SSR defects (line 4). It also includes a domain separator containing the contract address into the signed message to prevent the CSR defects (line 5). Furthermore, to address SF defects, it replaces the address to be approved (line 9 in Fig. 9 and line 7 in Fig. 2) from *msg.sender* to the *spender* specified by the signature (line 4). It ensures that even if an attacker front-runs the signature, he cannot change the intended contract behavior, *i.e.*, *owner* approving *spender* for a certain *value* of tokens.

In the supplementary material [39], we provide more real-world examples to demonstrate how these solutions are applied to prevent cryptographic defects in practice.

```
1 function permit(address owner,address spender,
      uint256 value,uint256 deadline,uint8 v,bytes32 r
      ,bytes32 s) public virtual {
2     if (block.timestamp > deadline) {
3         revert ERC2612ExpiredSignature(deadline);}
4     bytes32 structHash = keccak256(abi.encode(
          PERMIT_TYPEHASH, owner, spender, value,
          _useNonce(owner), deadline));
5     bytes32 hash = _hashTypedDataV4(structHash);
6     address signer = ECDSA.recover(hash, v, r, s);
7     if (signer != owner) {
8         revert ERC2612InvalidSigner(signer, owner);}
9     _approve(owner, spender, value);
10 }
```

Fig. 9. Fixing defects in Fig. 2

B. Threats to Validity and Limitations

Threats to Validity. In the experiment, we employed random sampling to evaluate the effectiveness of CRYSOL, which might introduce potential sampling bias. To reduce the impact, we carefully selected the sampling ratio and size to achieve a confidence level of 95% and a confidence interval of 10, which is considered sufficient in previous studies [1], [9], [12], [40]. Additionally, we manually labeled true/false positives and negatives of the sampled contracts, which could potentially lead to labeling mistakes. To mitigate this threat, we employed a double-check process, conducted by authors with more than three years of research experience in smart contract security.

Limitations. Despite CRYSOL's strengths, it might have the following potential limitations. First, CRYSOL relies on pre-defined oracles to detect defects, which might not cover newly emerging defects beyond the existing nine categories of cryptographic defects. However, given that these defined defects are derived from up to security reports from 31 security teams and involve all common on-chain cryptographic tasks, we believe CRYSOL effectively captures *common* cryptographic defects in existing smart contracts. Its framework also allows future studies to easily incorporate new defects. Second, CRYSOL relies on on-chain information to guide the fuzzing process. In scenarios such as analyzing undeployed smart contracts, such information might not be directly accessible. However, internal testing conducted before contract deployment, such as acceptance testing on local testnets, typically covers the main usage patterns of the contracts. Utilizing these test transactions, CRYSOL can extract necessary information and detect defects before deployment.

VII. Related Work

A. Defining and Detecting Defects in Smart Contracts

Due to the recurring security incidents, a substantial body of research has been dedicated to defining and detecting defects in smart contracts [69]–[71]. Luu *et al.* [55] took the first close look at smart contract security and proposed Oyente to detect four types of defects in smart contracts. Chen *et al.* [9] defined 20 types of contract defects through the analysis of Stack Exchange posts and real-world smart contracts and proposed a tool to detect them [72]. Liu *et al.* [12] studied access control bugs in smart contracts and detected them by dynamically role mining and conformance testing. However, they mainly studied general programming defects such as *Reentrancy* [10], rather than crypto-specific defects we focused on. For example, Liu *et al.* [12] focused on defective access control policies, rather than cryptographic defects that compromise the access control. Ye *et al.* [61] introduced a fuzzing tool to detect state inconsistency bugs, which utilizes contextual information collected from on-chain transactions to guide the fuzzing process.

While there is a lack of academic research on cryptographic defects, several defects we defined have attracted attention from the industry [6], [17]. To our knowledge, the Smart Contract Weakness Classification (SWC) list [17] has the most overlap with our categorization, which includes only four of nine defects we defined. Specifically, SWC-121 [73] documents weaknesses caused by single-contract/cross-contract signature replays, involving SSR and CSR defects. SWC-133 [74] and SWC-117 [47] are analogous to HC and SM defects, respectively. While the SWC list documents these defects, it does not provide practical detect patterns or tools for their detection.

B. Cryptographic Defects in Traditional Software

Cryptographic defects have become a common cause of security issues in software [75]–[78]. Lazar *et al.* [79] analyzed 269 crypto-related security incidents in the CVE database and found 83% of them were caused by cryptographic defects introduced by developers' non-standard practices. Egele *et al.* [76] summarized six common cryptographic defects in Android applications and proposed a tool to detect them. They found that 88% of 11,748 Android applications that use cryptographic functionalities contain at least one defect. Hazhirpasand *et al.* [80] found that 99.8% of 489 Github projects using Java Cryptography Architecture (JCA) APIs contain at least one defect.

However, these studies mainly focus on cryptographic defects in traditional software. Our results reveal differences between cryptographic defects in smart contracts and those in other well-studied software (*i.e.*, Java applications), in terms of both definition and detection. Firstly, due to the differences in common cryptographic tasks, the definition and categorization of defects in smart contracts differ inherently. For example, encryption-related defects are the most common in Java, but smart contracts rarely implement encryption, hence do not have these defects. Secondly, the detection methods also differ. In Java, defects often arise from direct API misuses, such as passing incorrect parameters to JCA APIs [80], and can be efficiently detected by static analyzers [78]. However, detecting smart contract defects like SSR involves analyzing multiple transactions interacting with a stateful contract, making existing detection techniques difficult to apply.

VIII. Conclusion

In this paper, we conducted the first study aimed at understanding and uncovering cryptographic defects in smart contracts. Through the analysis of 2,406 security reports, we proposed the first classification of cryptographic defects in smart contracts. It encompasses nine distinct defect types and covers a wide range of cryptographic tasks in smart contracts. To demonstrate these defects in real-world applications, we presented CRYSOL, a fuzzing-based tool for cryptographic defect detection. It collects fine-grained crypto-related semantics based on transaction replaying and dynamic taint analysis and incorporates crypto-specific fuzzing strategies for test case generation. The evaluation results indicated that CRYSOL can effectively detect real-world cryptographic defects, with an overall precision of 95.4% and a recall of 91.2%. Furthermore, CRYSOL revealed that 5,847 (22.7%) out of 25,745 crypto-related smart contracts contain at least one cryptographic defect, demonstrating their prevalence in real-world cryptographic practices.

Acknowledgement

This work is supported by National Key Research and Development Program of China (2023YFB2704800) and National Natural Science Foundation of China (62202011, 62172010, 62332004). We also thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] J. Zhang, J. Chen, Z. Wan, T. Chen, J. Gao, and Z. Chen, "When contracts meets crypto: Exploring developers' struggles with ethereum cryptographic apis," in *46th International Conference on Software Engineering (ICSE 24)*, 2024.

[2] R. Bloemen, L. Logvinov, and J. Evans, "Eip-712: Typed structured data hashing and signing," 2017. [Online]. Available: https://eips.ethereum.org/EIPS/eip-712

[3] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[4] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[5] Ethereum, "Merkle proofs for offline data integrity," 2023. [Online]. Available: https://ethereum.org/vi/developers/tutorials/merkle-proofs-for-offline-data-integrity

[6] Z. Bai, "You may pay more than you can imagine," 2018. [Online]. Available: https://github.com/nkbai/defcon26/tree/master/docs

[7] Immunefi, "Hack analysis: Nomad bridge, august 2022," 2022. [Online]. Available: https://medium.com/immunefi/hack-analysis-nomad-bridge-august-2022-5aa63d53814a

[8] ——, "Polygon double-spend bugfix review," 2021. [Online]. Available: https://medium.com/immunefi/polygon-double-spend-bug-fix-postmortem-2m-bounty-5a1db09db7f1

[9] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 327–345, 2020.

[10] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 65–68.

[11] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 664–676.

[12] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 716–727.

[13] J. R. Wood and L. E. Wood, "Card sorting: current practices and beyond," *Journal of Usability Studies*, vol. 4, no. 1, pp. 1–6, 2008.

[14] C. Nist, "The digital signature standard," *Communications of the ACM*, vol. 35, no. 7, pp. 36–40, 1992.

[15] B. Preneel, "Cryptographic hash functions," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 431–448, 1994.

[16] Wikipedia, "List of random number generators," 2023. [Online]. Available: https://en.wikipedia.org/wiki/List_of_random_number_generators

[17] S. Registry, "Smart contract weakness classification and test cases," 2023. [Online]. Available: https://swcregistry.io/

[18] Ethereum, "Opcodes for the evm," 2023. [Online]. Available: https://ethereum.org/en/developers/docs/evm/opcodes

[19] H. Tjaden, L. Matt, D. Piotr, and H. James, "Eip-152: Add blake2 compression function 'f' precompile," 2016. [Online]. Available: https://eips.ethereum.org/EIPS/eip-152

[20] B. Guido, D. Joan, P. Michal, and G. V. Assche, "The keccak sha-3 submission," 2011. [Online]. Available: https://keccak.team/files/Keccak-submission-3.pdf

[21] W. Penard and T. van Werkhoven, "On the secure hash algorithm family," *Cryptography in context*, pp. 1–18, 2008.

[22] H. Dobbertin, A. Bosselaers, and B. Preneel, "Ripemd-160: A strengthened version of ripemd," in *Fast Software Encryption: Third International Workshop Cambridge, UK, February 21–23 1996 Proceedings 3*. Springer, 1996, pp. 71–82.

[23] J.-P. Aumasson, W. Meier, R. C.-W. Phan, L. Henzen, J.-P. Aumasson, W. Meier, R. C.-W. Phan, and L. Henzen, "Blake2," *The Hash Function BLAKE*, pp. 165–183, 2014.

[24] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, pp. 36–63, 2001.

[25] V. Buterin, "Eip-198: Big integer modular exponentiation," 2017. [Online]. Available: https://eips.ethereum.org/EIPS/eip-198

[26] C. Reitwiessner, "Eip-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128," 2017. [Online]. Available: https://eips.ethereum.org/EIPS/eip-196

[27] V. Buterin and C. Reitwiessner, "Eip-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128," 2017. [Online]. Available: https://eips.ethereum.org/EIPS/eip-197

[28] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 2016, pp. 305–326.

[29] M. Lundfal, "Erc-2612: Permit extension for eip-20 signed approvals," 2020. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2612

[30] Openzepplin, "Checking signatures on-chain," 2023. [Online]. Available: https://docs.openzeppelin.com/contracts/2.x/utilities

[31] W. A. Florac *et al.*, *Software quality measurement: A framework for counting problems and defects*. Carnegie Mellon University, Software Engineering Institute, 1992.

[32] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[33] J. Chen, M. Huang, Z. Lin, P. Zheng, and Z. Zheng, "To healthier ethereum: A comprehensive and iterative smart contract weakness enumeration," *arXiv preprint arXiv:2308.10227*, 2023.

[34] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo, and X. Yang, "Smart contract security: a practitioners' perspective," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1410–1422.

[35] Consensys, "A complete suite of products to create and participate in web3," 2023. [Online]. Available: https://consensys.io/

[36] T. of Bits, "Trails of bits," 2023. [Online]. Available: https://www.trailofbits.com/

[37] Etherscan, "Smart contracts audit and security," 2023. [Online]. Available: https://etherscan.io/directory/Smart_Contracts/Smart_Contracts_Audit_And_Security

[38] Medium, "Medium," 2023. [Online]. Available: https://medium.com/

[39] CrySol, "Online supplement material," 2023. [Online]. Available: https://github.com/Jiashuo-Zhang/CrySol

[40] S. Yang, J. Chen, and Z. Zheng, "Definition and detection of defects in nft smart contracts," in *32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[41] Solidified, "Audit report for loopring on may 21st, 2020." 2020. [Online]. Available: https://github.com/solidified-platform/audits/blob/master/AuditReport-LoopringHebaoWallet[21.05.2020].pdf

[42] Etherscan, "Ethereum pending transactions," 2023. [Online]. Available: https://etherscan.io/txsPending

[43] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 910–927.

[44] C. Baum, J. Hsin-yu Chiang, B. David, T. K. Frederiksen, and L. Gentile, "Sok: Mitigation of front-running in decentralized finance," in *International Conference on Financial Cryptography and Data Security*. Springer, 2022, pp. 250–271.

[45] J. Groth and V. Shoup, "On the security of ecdsa with additive key derivation and presignatures," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2022, pp. 365–396.

[46] C. Decker and R. Wattenhofer, "Bitcoin transaction malleability and mtgox," in *Computer Security-ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II 19*. Springer, 2014, pp. 313–326.

[47] S. Registry, "Signature malleability," 2023. [Online]. Available: https://swcregistry.io/docs/SWC-117

[48] Verichains, "Verichains public audit report - thetanarena," 2021. [Online]. Available: https://github.com/verichains/public-audit-reports/blob/main/VerichainsPublicAuditReport-ThetanArena-v1.2.pdf

[49] E. C. D. S. Algorithm, "Public key recovery," 2023. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm#Public_key_recovery

[50] E. O. Documentation, "Merkle proofs for offline data integrity," 2023. [Online]. Available: https://ethereum.org/es/developers/tutorials/merkle-proofs-for-offline-data-integrity

[51] M. Bellare and P. Rogaway, “Collision-resistant hashing: Towards making uowhfs practical,” in *Annual International Cryptology Conference*. Springer, 1997, pp. 470–484.

[52] Ethereuk, “Non-standard packed mode,” 2023. [Online]. Available: https://docs.soliditylang.org/en/v0.8.23/abi-spec.html#non-standard-packed-mode

[53] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, “Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 227–239.

[54] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, “sfuzz: An efficient adaptive fuzzer for solidity smart contracts,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[55] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, “Making smart contracts smarter,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[56] Ethereum, “Ethereum archive node,” 2023. [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/archive-nodes

[57] Y. Kim, S. Jeong, K. Jezek, B. Burgstaller, and B. Scholz, “An off-the-chain execution environment for scalable testing and profiling of smart contracts,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 565–579.

[58] C. Shou, S. Tan, and K. Sen, “Ityfuzz: Snapshot-based fuzzer for smart contract,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 322–333.

[59] S. Documentation, “Contract abi specification,” 2023. [Online]. Available: https://docs.soliditylang.org/en/latest/abi-spec.html

[60] Etherscan, “The ethereum blockchain explorer,” 2023. [Online]. Available: https://etherscan.io/

[61] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, “Detecting state inconsistency bugs in dapps via on-chain transaction replay and fuzzing,” in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 298–309.

[62] L. Liu, L. Wei, W. Zhang, M. Wen, Y. Liu, and S.-C. Cheung, “Characterizing transaction-reverting statements in ethereum smart contracts,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 630–641.

[63] K. Sen, “Concolic testing,” in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, 2007, pp. 571–572.

[64] R. Corin and F. A. Manzano, “Efficient symbolic execution for analysing cryptographic protocol implementations,” in *International Symposium on Engineering Secure Software and Systems*. Springer, 2011, pp. 58–72.

[65] M. Vanhoef and F. Piessens, “Symbolic execution of security protocol implementations: Handling cryptographic primitives,” in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[66] Quantstamp, “Pine audit report,” 2022. [Online]. Available: https://certificate.quantstamp.com/full/pine.pdf

[67] Openzepplin, “Implementation of the erc-20 permit extension,” 2023. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC20Permit.sol

[68] ——, “The standard for secure blockchain applications,” 2023. [Online]. Available: https://www.openzeppelin.com/

[69] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, “Smart contract vulnerability analysis and security audit,” *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.

[70] Z. Wang, H. Jin, W. Dai, K.-K. R. Choo, and D. Zou, “Ethereum smart contract security research: survey and future research opportunities,” *Frontiers of Computer Science*, vol. 15, pp. 1–18, 2021.

[71] N. Ivanov, C. Li, Q. Yan, Z. Sun, Z. Cao, and X. Luo, “Security threat mitigation for smart contracts: A comprehensive survey,” *ACM Computing Surveys*, 2023.

[72] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, “Defectchecker: Automated smart contract defect detection by analyzing evm bytecode,” *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2189–2207, 2021.

[73] S. Registry, “Missing protection against signature replay attacks,” 2023. [Online]. Available: https://swcregistry.io/docs/SWC-121

[74] SWC, “Hash collisions with multiple variable length arguments,” 2023. [Online]. Available: https://swcregistry.io/docs/SWC-133

[75] A. S. Ami, N. Cooper, K. Kafle, K. Moran, D. Poshyvanyk, and A. Nadkarni, “Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 614–631.

[76] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, “An empirical study of cryptographic misuse in android applications,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013, pp. 73–84.

[77] A.-K. Wickert, L. Baumgärtner, F. Breitfelder, and M. Mezini, “Python crypto misuses in the wild,” in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–6.

[78] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. Yao, and N. Meng, “Automatic detection of java cryptographic api misuses: Are we there yet?” *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 288–303, 2022.

[79] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, “Why does cryptographic software fail? a case study and open problems,” in *Proceedings of 5th Asia-Pacific Workshop on Systems*, 2014, pp. 1–7.

[80] M. Hazhirpasand, M. Ghafari, and O. Nierstrasz, “Java cryptography uses in the wild,” in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–6.