SIGUARD: Detecting Signature-Related Vulnerabilities in Smart Contracts

Jiashuo Zhang*†, Yue Li*†, Jianbo Gao*†§, Zhi Guan‡, Zhong Chen*†
*Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, China
†School of Computer Science, Peking University, Beijing, China
‡National Engineering Research Center for Software Engineering, Peking University, Beijing, China
§Corresponding Author
{zhangjiashuo,liyue_cs,gaojianbo,guan,zhongchen}@pku.edu.cn

***Abstract*—Ethereum smart contract enables developers to enforce access control policies of critical functions using built-in signature verification interfaces, *i.e.*, ecrecover. However, due to the lack of best practices for these interfaces, improper verifications commonly exist in deployed smart contracts, leaving potential unauthorized access and financial losses. Even worse, the attack surface is ignored by both developers and existing smart contract security analyzers. In this paper, we take a close look at signature-related vulnerabilities and de-mystify them with clear classification and characterization. We present SIGUARD, the first automatic tool to detect these vulnerabilities in real-world smart contracts. Specifically, SIGUARD explores signature-related paths in the smart contract and extracts data dependencies based on symbolic execution and taint analysis. Then, it conducts vulnerability detection based on a systematic search for violations of standard patterns including EIP-712 and EIP-2621. The preliminary evaluation validated the efficacy of SIGUARD by reporting previously unknown vulnerabilities in deployed smart contracts on Ethereum. A video of SIGUARD is available at https://youtu.be/xXAEhqXWOu0.**

***Index Terms*—smart contract, digital signature, software analysis, vulnerability detection**

I. INTRODUCTION

Digital signatures have been widely adopted in smart contracts to enable on-chain identity authentication and access control. As the most popular on-chain cryptographic practices, they have bought a wide range of innovations to smart contacts and spawned numerous real-world on-chain applications.

However, the deficient understanding of security implications in signatures becomes a large obstacle for smart contract developers, leading to non-standard and insecure real-world practices. Even with provable security guarantees offered by digital signatures and a clean implementation offered by built-in pre-compiled contracts, *i.e.*, *ecrecover*, the ad-hoc designs for signature verification still introduce a large attack surface. For example, a security team reported more than fifty smart contracts that suffer signature replay attacks [1], demonstrating the prevalence and damage of signature-related vulnerabilities.

The new attack surface is far from being well-studied. Most previous studies focus on vulnerabilities introduced by internal mechanisms of smart contracts, such as reentrancy [2]. They rarely discuss non-standard cryptographic practices caused by improper ad-hoc designs and target no vulnerability with signature-specific implications. Consequently, the community still lacks knowledge and tools to mitigate the threat.

Addressing signature-related vulnerabilities introduces two-fold challenges. First, there is no general definition and classification of these vulnerabilities, making it difficult to automatically detect the existence of them in practice. Second, the signature verification commonly involves multiple hash operations and inter-contract invocations, resulting in complex control flows and data flows in its analysis. Such analysis is essentially difficult without well-designed modeling for built-in cryptographic operations and inter-contract invocations.

To address these challenges, we take a close look at signature-related vulnerabilities in smart contracts and de-mystify them with clear classification and well-defined detect patterns. Based on them, we highlight an analysis framework called SIGUARD for automatically detecting these vulnerabilities. We summarize our contributions as follows:

- We characterize two new classes of signature-related vulnerabilities in smart contracts, illustrate their root causes, and introduce feasible detect patterns.
- We propose SIGUARD, the first automatic tool to detect these vulnerabilities. It symbolically explores signature-related branches of given smart contracts, traces the taint from multiple data sources, and detects signature-related vulnerability based on symbolic execution sequences and taint flow information.
- We open-source SIGUARD[1] and preliminarily evaluate it on 1000 signature-related smart contracts collected from Ethereum. It successfully identifies 52 previously unknown vulnerabilities in these contracts, demonstrating its effectiveness in detecting real-world vulnerabilities.

II. SIGNATURE-RELATED VULNERABILITIES

In this section, we start with a brief introduction to ECDSA [3] and its applications in smart contracts. Then, we characterize two classes of signature-related vulnerabilities in smart contracts and illustrate them with motivation examples. Our classification and characterization are well motivated, as they cover the most common attack surface in signature-related applications, *i.e.*, signature replay attacks. They derive from the real-world adversary model, where an adversary has two capabilities: replaying historical signatures of the

[1] https://github.com/Jiashuo-Zhang/Siguard

victim contract and replaying signatures from other deployed contracts. Based on the characterization, we introduce standard solutions and detect patterns for these two vulnerabilities to facilitate the security of real-world practices.

A. ECDSA in smart contracts

Most blockchain platforms use a cryptographic algorithm called Public Key Recovery for signature verification. It takes the signature (r, s) and the signed message m as input, and recovers the public key pk' signing the signature. The signature is valid if and only if the expected signer's public key pk matches the recovered public key pk'.

Specifically, Ethereum officially introduces *ecrecover* [4], a pre-compiled contract deployed at address 0x1, for on-chain public key recovery. It takes the signature (r, s) and an additional recovery identifier v as input and outputs the on-chain address mapped from the recovered public key pk'.

B. Vulnerability 1: Stateless Signature Verification

The first vulnerability is called stateless signature verification. It allows attackers to replay past signatures in a single contract and perform sensitive operations more than once.

Scenarios such as on-chain access control usually require the *one-time* property in signature verification, *i.e.*, a signature should get invalidated once it is verified. Otherwise, attackers can replay the same signature to perform sensitive operations more than once and launch attacks such as double-spending.

However, in practice, many smart contract developers implement signature verification in a stateless way, *i.e.*, they only verify the validity of the signature without checking whether the signature has been used. Consequently, smart contracts will always accept these valid (but being replayed) signatures, leading to unintended contract operations.

```
function permit(address owner, address spender, uint256 value,
uint256 deadline, uint8 v, bytes32 r, bytes32 s) external
{
    bytes32 hash =keccak256(abi.encode(owner, spender, value,
    deadline));
    address signer = ecrecover(hash, v, r, s);
    require(signer != address(0),"Invalid Signature");
    require(owner == signer, "Invalid Signer");
    require(block.timestamp <= deadline, "Permit Expired");

    _approve(owner, spender, value);
}
```

Fig. 1. A Vulnerable Function with Stateless Signature Verification

Fig. 1 shows an example of this vulnerability. The *permit* function allows the token owner to approve his tokens to the spender with a previously signed message. However, due to the stateless signature verification, the attacker is able to replay the signature from the token owner, call the *_approve* function repeatedly, and successfully receive the approved tokens more than once.

In practice, smart contracts usually use standard protective patterns introduced by a set of EIP proposals, *e.g.*, EIP-2612 [5] to fight against this vulnerability. To guarantee the *one-time* property, they require the signed message to contain a storage variable, *e.g.*, a monotonic nonce value, which is updated after each signature verification. Hence, the detect pattern for this vulnerability is to check whether at least one storage variable included in the signed message also appears in the write set of the current transaction.

C. Vulnerability 2: Unseparated Signing Domain

The second vulnerability is called unseparated signing domain. It allows attackers to pass the signature verification in one contract by replaying signatures from other contracts.

A signing domain is a specification for the structure of the signed message with contract-specific semantics. For the example in Fig. 1, the signing domain contains the addresses of the owner and spender, the token value, and the deadline, meaning that the *owner* wants to approve tokens with *value* amount to the *spender* and the approval is valid until *deadline*.

Sharing the same signing domain between different contracts means that a valid signature for contract A will also be valid for contract B. It might directly cause unintended operations in these contracts. This vulnerability could become a large attack surface given that more than 79% of real-world smart contracts are clones [6], thus may share the same signing domain.

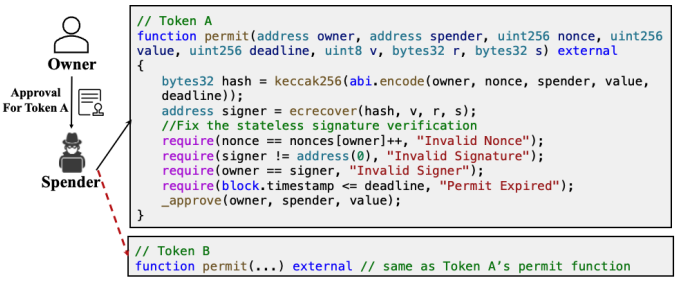

Fig. 2. Vulnerable Smart Contracts with Unseparated Signing Domain

Fig. 2 shows an example of this vulnerability. There are two different token contracts, *i.e.*, token A and token B, implementing the same permit function and thus sharing the same signing domain. The token owner holding both token A and B wants to sign a signature and approve token A to the spender. However, due to the unseparated signing domain between these two contracts, the signature for token A is also valid for token B. Hence, the spender can make another call to the permit function of token B with the same signature and successfully receive token B, even though the token owner never intends to approve token B to the spender.

To explicitly specify which contract the signature is generated for, the address of the verifying contract should always be added into the signed message as a domain separator. Therefore, we use the following detect pattern to detect this vulnerability: mark the contract address (*i.e.*, the stack output of the `ADDRESS` opcode) as the source, and check whether it can propagate to the signing domain during the contract execution.

It is worth noting that several EIP proposals, such as EIP-712 [7], introduce a more complex domain separator with extra fields such as the name of the contract. However, as the contract address is unique enough to capture all cross-contract signature replay attacks, we leave the detection for other fields for future work.

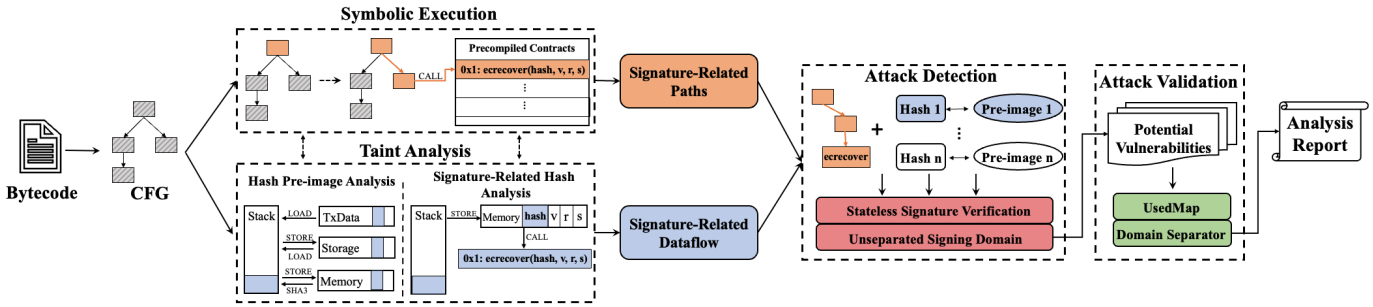

Fig. 3. Workflow of SIGUARD

III. DETECTING SIGNATURE-RELATED VULNERABILITIES

In this section, we introduce our approach based on symbolic execution and taint analysis for detecting signature-related vulnerabilities. Generally speaking, SIGUARD takes the bytecode of the smart contract as input and constructs the control flow graph (CFG). Then, it explores the CFG to extract signature-related paths based on symbolic execution. During the symbolic execution, SIGUARD conducts a dynamic taint analysis to track signature-related dataflow information. After that, the detector takes signature-related path conditions and dataflow information as input and detects two types of potential vulnerabilities. Finally, SIGUARD validates the potential vulnerabilities by identifying possible protective patterns and reports the analysis result. We demonstrate the workflow of SIGUARD in Fig 3.

A. Constructing CFG

SIGUARD builds a context-sensitive and inter-procedural CFG for the creation bytecode of smart contracts. It contains the contract constructor, all functions of the contracts, and inter-procedure calls between them. All public and external functions in the CFG are used as entry points for the following CFG exploration.

B. Exploring Signature-Related Paths

In SIGUARD, symbolic paths in the constructed CFG are explored by transactions with symbolic input data. During the transaction execution, SIGUARD simulates the Ethereum Virtual Machine (EVM) and symbolically executes each opcode in the path and changes the stack, memory, and storage of EVM. If a conditional branch opcode (`JUMPI`) is met, it will query the SMT solver to check whether the branch condition is satisfiable.

SIGUARD extracts all signature-related paths for potential vulnerability detection. Specifically, SIGUARD checks the destination addresses of all external calls (`STATICCALL`, `CALL`, `CALLCODE`, and `DELEGATECALL`) in the path to determine whether it could be a call to *ecrecover*.

Moreover, to trace signature-related dataflow information, SIGUARD introduces a dynamic taint analysis technique that propagates the taints and traces their flows during the symbolic execution. Specifically, it contains two phases, *i.e.*, the hash pre-image analysis and the signature-related hash analysis. In the first phase, SIGUARD traces the pre-images of hash variables. It records the dependency between hash pre-images and external data sources by marking opcodes (`SLOAD`, `CALLDATALOAD`) related to external data as taint sources and the built-in hash opcode (`SHA3`) as the sink. In the second phase, it analyzes which hash variable is related to the signature verification. It marks hash variables as taint sources, traces their propagation in the memory and stack, and extracts hash variables used in the external calls to *ecrecover*.

C. Detecting Potential Vulnerabilities

Based on the path condition and signature-related dataflow information, the detector checks potential vulnerabilities in the contract based on detect patterns proposed in Section II. Specifically, for the stateless signature verification, the detector gets all storage variables in the signature hash and checks whether it is updated after each signature verification based on the storage readset and writeset. For unseparated signing domain, it checks whether the pre-image contains the contract address to prevent conflicted signing domains.

D. Vulnerability Validation

In the last step, the validator identifies two possible protective patterns and attempts to remove false positives in potential vulnerabilities. Moreover, the validator ignores paths causing no state change, *e.g.*, it is a read-only function, as these paths will not cause persistent damage to the contract.

1) UsedMap: Several contracts implement an alternative protective pattern called the UsedMap to prevent signature replay. The difference is that UsedMap does not require the signing domain to include any storage variable. They maintain a mapping that records the hashes of all signed messages they have verified, in order that each signed message can be verified once only. We identify this pattern by heuristically analyzing the storage readset and writeset of the vulnerable path and checking whether a storage slot is updated with a signature-related index. The full-fledged identification of this pattern is left for future work.

2) Domain Seperator in Storage: Several contracts implement the domain separator as a storage variable and initialize it in the contract constructor. To detect such patterns, SIGUARD conducts an inter-transaction analysis to identify which storage slot contains a potential domain separator and whether the storage slot is used in the signature verification.

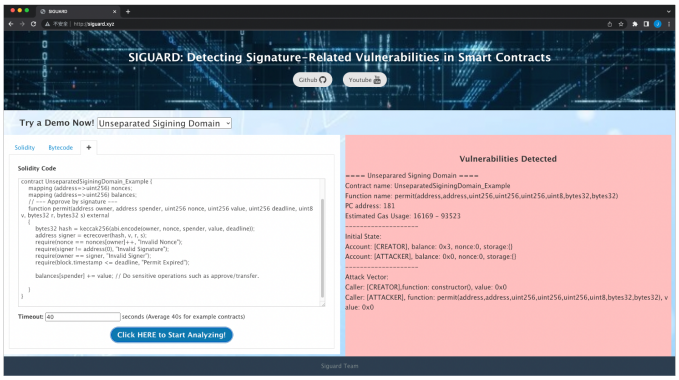

Fig. 4. The Interface of SIGUARD

TABLE I
Evaluation Results of SIGUARD.

Vulnerability Type		Detected	Affected	Severity
Stateless Signature Verification		5	5	High
Unseparated Signing Domain	Lack Separator	23	23	High
	Non-Standard	2	2	Medium
	Uninitialized	22	34	Low
Total		52	64	–

IV. USING SIGUARD

We have implemented and open-sourced SIGUARD to help the community mitigate the threat. We instantiate the symbolic executor with Mythril [8] and use Z3 [9] as the SMT solver, while in principle, any other EVM symbolic executors and SMT solvers would also fit our setting.

Moreover, we have deployed SIGUARD as a public web service[2] to provide vulnerability detection functionalities for smart contract developers before deploying smart contracts. As shown in Fig. 4, it takes the bytecode or source code input by users in the front end and performs the analysis in the back end. After the analysis finishes, SIGUARD will output a detailed analysis report including the types and locations of detected vulnerabilities. It also contains an attack transaction sequence to show how to exploit these vulnerabilities and help developers investigate them.

V. PRELIMINARY EVALUATION

To validate the effectiveness of SIGUARD, we collect 1000 real-world signature-related smart contracts with verified source code and bytecode from Etherscan using the smart contract search APIs[3]. All dataset and results are publicly available[4]. The evaluation results in Table I demonstrate the effectiveness of SIGUARD in finding real-world vulnerabilities. The *Detected* column shows the number of smart contracts reported by SIGUARD while the *Affected* column shows how many contracts on Ethereum have the same bytecode as these detected contracts. We manualy validate all reported vulnerabilities based on their verified source codes. In summary, SIGUARD successfully detects 52 vulnerable smart contracts, which affects 64 deployed contracts on Ethereum, demonstrating a largely underestimated attack surface.

To understand the damage better, we further investigate all these vulnerabilities and assess their severity. First, we assign high severity to vulnerable contracts with stateless signature verification as they may directly suffer signature replay attacks. We further split the unseparate signing domain vulnerabilities into three sub-groups according to whether they contain a domain separator. We assign high severity to vulnerable contracts with no separator, as they provide no protection against cross-contract signature replay attacks. We assign medium severity to vulnerable contracts with non-standard domain separators, *i.e.*, do not contain the contract address, as these ad-hoc designs may lead to conflicted signing domains. We assign low severity to contracts with uninitialized separators. They implement the domain separator as a storage variable that may remain uninitialized when the signature verification is called. It may not cause attacks as long as the initialization is finished during the contract deployment.

VI. RELATED WORK

To the best of our knowledge, SIGUARD is the first automatic tool to detect signature-related vulnerabilities in smart contracts. Previous work [1] manually analyzes the source code of smart contracts and reveals the existence of replay attacks due to improper signature verification. However, in these previous works, there is neither explicit characterization nor automatic detection for these vulnerabilities. Besides signature-related vulnerabilities, there also exists a substantial body of work [10] based on symbolic execution, fuzzing, and formal verification to automatically detect other smart contract vulnerabilities (*e.g.*, reentrancy [2], transaction order dependence [11], and gas-related issues [12]) and enhance the security of smart contracts.

VII. CONCLUSION AND FUTURE WORK

In this paper, we highlight the signature-related attack surface in smart contracts and characterize two classes of vulnerabilities in them. We propose SIGUARD, the first automatic tool based on symbolic execution and taint analysis, to detect these vulnerabilities. In the preliminary evaluation, SIGUARD successfully detects 52 vulnerabilities in real-world smart contracts and validates its effectiveness. In the future, we plan to improve the detection technique to reduce false positives/negatives and conduct a large-scale analysis to facilitate the understanding of signature-related attacks in the decentralized world.

ACKNOWLEDGEMENT

This work is supported by National Key Research and Development Program of China (2020YFB1005404), National Natural Science Foundation of China (62202011, 62172010), Beijing Natural Science Foundation (M21040), and Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing. We thank the anonymous reviewers for their valuable feedback.

[2]http://siguard.xyz
[3]https://etherscan.io/searchcontractlist?a=all&q=ecrecover
[4]https://github.com/Jiashuo-Zhang/Siguard

REFERENCES

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

[2] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 65–68.

[3] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.

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

[6] M. Kondo, G. A. Oliva, Z. M. J. Jiang, A. E. Hassan, and O. Mizuno, "Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, no. 6, pp. 4617–4675, 2020.

[7] Ethereum, "Typed structured data hashing and signing," 2018. [Online]. Available: https://eips.ethereum.org/EIPS/eip-712

[8] Consensys, "Mythril: A security analysis tool for ethereum smart contracts." 2018. [Online]. Available: https://github.com/ConsenSys/mythril

[9] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 337–340.

[10] 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.

[11] 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.

[12] A. Ghaleb, J. Rubin, and K. Pattabiraman, "Etainter: Detecting gas-related vulnerabilities in smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, p. 728–739.