Securing Deployed Smart Contracts and DeFi With Distributed TEE Cluster

Zecheng Li, Bin Xiao, *Senior Member, IEEE*, Songtao Guo, *Senior Member, IEEE*, and Yuanyuan Yang, *Fellow, IEEE*

Abstract—Smart contract technologies can be used to implement almost arbitrary business logic. They can revolutionize many businesses such as payments, insurance, and crowdfunding. The resulting birth of decentralized finance (DeFi) has gained significant momentum. Smart contracts and DeFi are now attractive targets for attacks. An important research question is how to protect deployed smart contracts and DeFi. Smart contracts cannot be modified once deployed, namely vulnerabilities cannot be fixed by patching. In this case, vulnerabilities in deployed contracts and DeFi might cause devastating consequences. In this paper, we put forward SolSaviour, a framework for protecting deployed smart contracts and DeFi. The core of SolSaviour is to build a smart contract protection mechanism based on democratic voting using a distributed trusted execution environment (TEE) cluster. Once a vulnerability in deployed contracts or DeFi is found, SolSaviour can destroy the defective contract and redeploy a patched contract via the distributed TEE cluster. Moreover, SolSaviour can migrate funds and state variables from the destroyed contract to the patched one. Compared with previous work, our approach can protect smart contracts and DeFi in a distributed manner, avoiding reliance on privileged users or trusted third parties. Our experiment results show that SolSaviour can protect smart contracts and complex DeFi protocols with feasible overhead.

Index Terms—Blockchain, DeFi security, decentralized finance (DeFi), smart contract, trusted execution environment (TEE)

1 Introduction

As the global market size of smart contract grows year by year, the security of smart contracts has raised huge concerns. Smart contracts and decentralized finance (DeFi) operate on a distributed blockchain environment. Their deployment and execution rely on the distributed consensus of the blockchain. Due to the tamper-proof feature of the blockchain, contracts and DeFi cannot be modified once deployed. Therefore, we cannot fix a deployed smart contract in the same way as patching a traditional application when the deployed contract has a vulnerability.

Vulnerabilities in deployed smart contracts and DeFi have caused significant losses in recent years due to the lack of proper protection methods. One of the most notorious incidents was the DAO hack [1], in which attackers exploited the reentrancy vulnerability in the DAO contract to steal ethers wantonly. During this attack, honest contract users can do nothing but to withdraw ethers to secure accounts as fast as possible. The DAO contract lost around 3.6 million ethers in this attack. In addition, with the increased popularity of DeFi applications, new types of attacks are appearing. In April 2022, the Fei protocol suffered from a reentrancy attack. The attacker discovered a reentrancy bug in Fei's collateral mechanism, meaning that it was possible to use the `fallback` function to release assets locked in the Fei contract when the loan arrived, and thus steal assets inside Fei. This attack caused around 28,380 ETH loss [2].

Spurred by these attacks, the community started to conduct research on detecting vulnerabilities before deployment. Many software analysis techniques are explored, including but not limited to symbolic execution [3], formal verification [4], static analysis [5], [6], dynamic analysis [7], and fuzzing testing [8]. However, these detection methods still have certain limitations. They cover limited types of vulnerabilities, have restricted detection efficiency (i.e., sensitive to some vulnerabilities, but insensitive to others), and suffer from possible false negative cases. We point out that for high-net-worth smart contracts, pre-deployment detection methods are not fully effective. It is possible that vulnerabilities may be discovered after deployment. Therefore, how to protect deployed smart contracts and DeFi remains a crucial problem.

Proxy pattern is a promising method to safeguard deployed smart contracts and DeFi. Using proxy pattern, a smart contract is separated into a proxy contract and an implementation contract. The proxy contract stores the address of implementation contract. Function calls to the proxy contract are forwarded to the implementation using `delegatecall` opcode, which executes the code of implementation contract with the data inside the proxy contract.

- *Zecheng Li and Bin Xiao are with the Department of Computing, The Hong Kong Polytechnic University, Kowloon, Hong Kong. E-mail: {zechli, b.xiao}@polyu.edu.hk.*
- *Songtao Guo is with the College of Computer Science, Chongqing University, Chongqing 400044, China. E-mail: guosongtao@cqu.edu.cn.*
- *Yuanyuan Yang is with the Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, NY 11794 USA. E-mail: yuanyuan.yang@stonybrook.edu.*

Manuscript received 26 June 2022; revised 11 November 2022; accepted 11 December 2022. Date of publication 27 December 2022; date of current version 13 January 2023.
This work was supported in part by Key-Area Research Development Program of Guangdong Provice under Grant 2020B0101090003, in part by HK RGC GRF under Grants PolyU 15209822 and PolyU 15217321.
(Corresponding author: Bin Xiao.)
Recommended for acceptance by Y. Yang.
Digital Object Identifier no. 10.1109/TPDS.2022.3232548

If a vulnerability is discovered in the implementation contract, the proxy pattern allows users to deploy a new patched implementation contract to replace the defective one. However, the proxy pattern still has some limitations, such as the proxy selector clashing problem, which can be exploited by attackers to steal assets inside contracts. In August 2022, a DeFi protocol Nomad was exploited and $190M assets were stolen [9]. Nomad bridge was deployed in proxy pattern and the vulnerability was introduced in an upgrade. In this case, though proxy pattern provides a possible solution to fix vulnerabilities in deployed smart contracts, it is still lack of the protection of inside assets. However, the asset in smart contracts and DeFi is, to some extent, the most important thing to protect.

In this paper, we focus on protecting deployed smart contracts and DeFi from external attacks or internal bugs, with a particular emphasis on securing inside assets. Considering that most high-net-worth contracts are multi-user scenarios, we implement the management of contracts in a democratic voting way. We propose the voteDestruct mechanism to allow contract participants (i.e., stakeholders) to vote on the contract's future. They can vote to lock, unlock, and destroy a potentially defective contract. The number of ethers (i.e., stake) they have deposited in the contract determines the weight of their votes. The more stake a stakeholder controls, the more weight its vote has. We also propose a distributed trusted execution environment (TEE) cluster to enable the management of contracts, such as update, destruction, and redeployment. The TEE cluster also takes charge of temporary asset escrow after destroying a defective smart contract and stake migration. By transferring the assets and state variables from defective contract to patched contract, TEE cluster protects the assets and guarantees the consistency of the contract state. For newly-deployed smart contracts, TEE cluster ensures that the data and internal stake distribution remains unchanged through state migration. For patched smart contracts, the TEE cluster deploys it and conducts the state migration to transfer all internal assets to the newly-deployed contract.

We achieve decentralized control of smart contracts and DeFi by multiple parties through a secure and principled combination of blockchain and trusted hardware. Assuming the integrity of the blockchain, users do not need to trust the validity, persistence, confidentiality, or correctness of smart contract creators, miners, or TEE nodes. SolSaviour thus can provide self-sustaining service even when some miners, contract creators, contract participants, or TEE nodes are unavailable.

In our preliminary conference version of this paper [10], we first present SolSaviour for repairing and recovering a defective smart contract. However, SolSaviour still has some drawbacks, such as external calls not being authenticated by the TEE cluster and no valid verification for generated patches. In this paper, we overcome these shortcomings and extend our work to the protection of DeFi. Some modules are added including a policy-based exception detector, an identity authentication module, and a patch tester. Complete API via which clients can invoke SolSaviour is provided. We also explore a method for redeploying a patched contract without TEE asset escrow. The main contributions of this paper are summarized as follow:

- We propose a voteDestruct mechanism to enable democratic voting on deployed smart contracts and DeFi, allowing stakeholders to make fair future decisions.
- We propose a distributed TEE cluster that allows contract stakeholders to replace the defective contract with a patched contract in a trusted manner. Our proposed TEE cluster can take charge of asset escrow and contract state migration. It also can verify the identity of message calls, preserve trusted execution of contract invocation, patching, and deployment. We also provide complete API of the TEE cluster.
- We give a thorough security analysis of SolSaviour in three aspects: balance security, correctness, and fairness.
- We collect smart contracts and DeFi protocols that were attacked in the past and use them to evaluate the effectiveness and performance of SolSaviour. Experiment results show that SolSaviour can effectively mitigate the loss caused by smart contract vulnerabilities with little overhead.

The remainder of this paper is organized as follows. Section 2 gives some background knowledge of this paper. In Section 3, we introduce the overview, workflow, building blocks, and API of SolSaviour. The detailed implementation is presented in Section 4. We analyze the security of SolSaviour in Section 5. We discuss potential improvement of SolSaviour in 6. The effectiveness and performance of SolSaviour are evaluated in Section 7. We present related work in Section 8 and conclude our work in Section 9.

2 Background

2.1 Smart Contract and DeFi

Smart Contract. A smart contract is a distributed computer program that runs in a blockchain environment. Supported by the underlying blockchain, smart contracts can store arbitrary state and perform arbitrary computation. The deployment and invocation of a smart contract is achieved by sending transactions to the blockchain. Message call transactions that conform to the internal logic of the contract are executed by miners and later included in the new blocks.

DeFi. DeFi applications are essentially smart contracts running on the blockchain. They can provide financial instruments that do not rely on third-party intermediaries, such as exchanges and banks. DeFi allows users to carry out financial services such as lending, investing in derivatives, and insuring. Users can store their money in a secure digital wallet and interact with DeFi through message call transactions. In DeFi, transactions are not executed through traditional centralized exchanges, but between participants and mediated via smart contracts. Since a DeFi application typically consists of multiple contracts, the risk of smart contract errors increases.

Internal State. The internal state indicates the values of contract variables and the stake distribution inside the contract. While migrating and upgrading contracts, the consistency of contract internal state should be maintained. People could recover the value of variables inside a smart

contract. The `getter` function can be used to acquire values of contract variables. The stake distribution can be recovered in a similar way as long as the contract explicitly define variables to store stake distribution. However, it is non-trivial to migrate the stake distribution from defective contracts to patched contracts, which requires actual transactions of funds. In this case, assets are transferred from defective contract account to patched contract account, in accordance with the stake distribution inside defective contract.

2.2 Defining Defects

Smart contracts and DeFi are subject to a wide variety of defects. Defective smart contracts can be divided into two categories: exploitable smart contracts and unexpected smart contacts (i.e., may have unexpected internal states). For the first type, either there are some problems within the contract implementation that create bugs (e.g., reentrancy vulnerability), or an attacker can exploit the contract internal logic to launch attacks (e.g., front running attack). Attackers can gain benefits that do not belong to themselves by exploiting these defects. For the second type, these defects may cause a smart contract to an unexpected state (e.g., a locked state). For example, a jackpot may never succeed because of a strictly equal operation [11]. Regardless of the defect, effective protection measures are needed to prevent potential asset loss.

2.3 Contract Protection

Generally, we define the defending methods of smart contracts and DeFi as repairing and recovering techniques. Repairing technique can alleviate the bugs in a smart contract, and recovering technique can save a contract from serious states.

Currently, almost all contract remediation techniques focus on repairing smart contracts before deployment. Some efforts identify vulnerabilities by statically analyzing the contract code and generating the appropriate patches. Other efforts, such as runtime validation, determine if a deployed contract is vulnerable and generating appropriate patches. The two efforts are similar in their inability to protect deployed smart contracts and DeFi. In addition, they cannot fix vulnerabilities that have not been detected. Nor can they protect the assets in deployed smart contracts.

One possible solution for protecting deployed contracts and DeFi is the proxy pattern, where the smart contract is separated into a proxy contract and a implementation contract. Once an error is exposed, a new implementation contract is deployed to replace the defective one. However, this approach is subject to the requirements of a trusted contract owner. That is, the contract developer sets its own address as a super user for future maintenance, which is a strong trust assumption.

2.4 Distributed TEE Cluster

TEE is a hardware-level trusted computing technology in which the CPU divides a portion of the memory area to ensure that internally loaded code and data are protected in terms of integrity and confidentiality. Integrity ensures that software outside the TEE cannot tamper with the inside code and data without authorization. Confidentiality implies that entities outside the TEE cannot acquire the information inside the TEE without permission. In Intel Software Protection Extensions (Intel SGX), these two attributes are achieved by hardware-level memory encryption, which isolates application-specific code and data in memory. TEE execution results can be verified in the form of remote attestation.

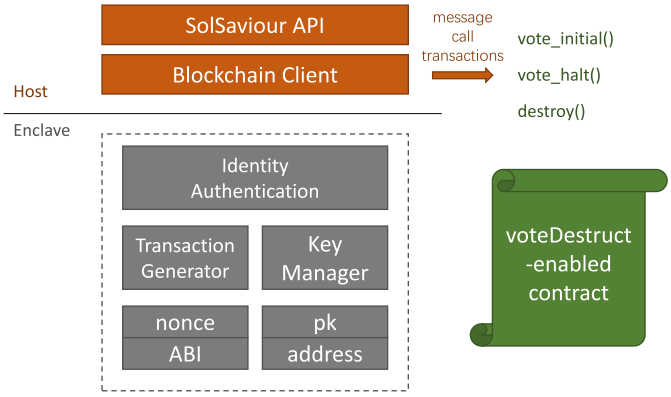


Fig. 1. The architecture of SolSaviour. Contract stakeholders can invoke the TEE cluster via SolSaviour API to generate message call transactions to instruct the voteDestruct-enabled contract.

Considering the availability and confidentiality problem of TEE, we propose distributed TEE cluster. Multiple TEE nodes work together in a distributed environment can avoid the service termination problem. We also propose a distributed signature scheme to avoid storing private keys in a single TEE node.

3 SolSaviour

3.1 What is SolSaviour

The architecture of SolSaviour is depicted in Fig. 1. SolSaviour consists of two core parts: a voteDestruct mechanism and a distributed TEE cluster. The voteDestruct mechanism is embedded in smart contracts. It allows smart contracts to be destroyed in the voting manner. TEE cluster can deploy a patched contract onto the blockchain and migrate all assets and stake distribution. Once a patched contract is deployed, stakeholders can continue to execute the contract without the vulnerability.

3.2 Workflow

First of all, stakeholders should prepare a TEE cluster for protecting smart contracts and DeFi. Contract stakeholders collect a cluster of SGX-capable computers and launch enclaves into them. Then, these TEE nodes corporate to establish a distributed TEE cluster following the bootstrapping process. After constructing the TEE cluster, users can invoke it to deploy a voteDestruct-enabled smart contract.

During the contract execution, an unknown bug may be disclosed. Then, stakeholders can check whether this exposed bug is a false positive. If not, they can invoke SolSaviour API to protect the deployed contract as shown in Fig. 2. The detailed workflow is summarized below:

Phase 1: Destroying the Defective Contract.

① Once identified a bug, stakeholders invoke the TEE cluster to lock the defective contract to prevent further attacks.

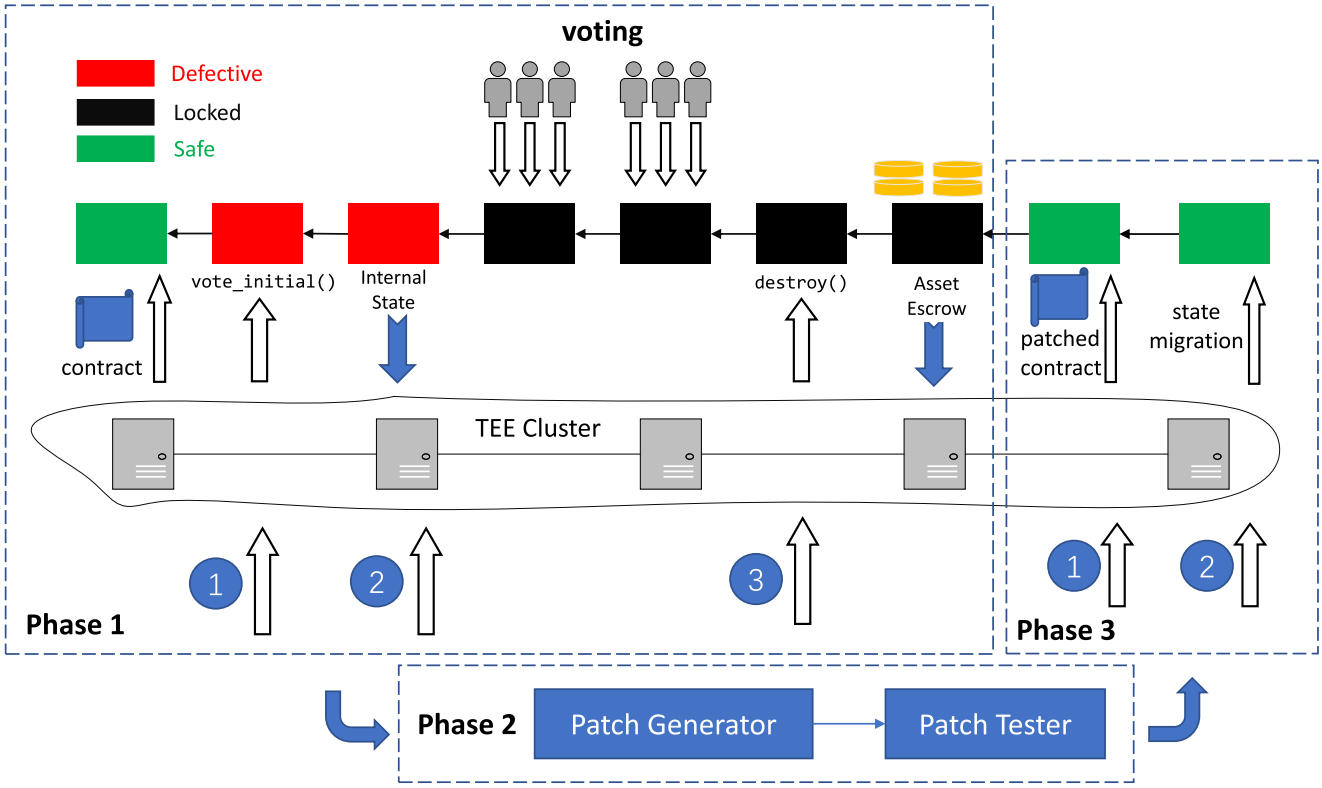


Fig. 2. The workflow of SolSaviour. In Phase 1, contract stakeholders can destroy a defective contract via democratic voting. In Phase 2, a patched contract is generated and tested. In Phase 3, patched contract is deployed to replace the defective one.

② Stakeholders invoke the TEE cluster to extract the internal state of the defective smart contract for future state migration. Specifically, it stores the values of state variables and the stake distribution at the time that the contract is locked.

③ Stakeholders can vote to destroy the defective smart contract via a cumulative voting way. They simply invoke the `vote()` function provided by the voteDestruct mechanism. After voting, stakeholders can invoke the TEE cluster to destroy the defective contract.

Phase 2: Preparing a Patched Contract Offline.

In this phase, stakeholders generate and test patches for the defective contract. Stakeholders can prepare a patch for the located vulnerability and integrate it with the original contract. After that, stakeholders leverage the patch tester to validate whether the vulnerability is resolved and whether the functionalities of the patched contract remain the same. Then, contract stakeholders can upload a patched contract into the TEE cluster.

Phase 3: Redeploying the Patched Contract.

① Stakeholders invoke the TEE cluster to deploy the patched contract prepared in Phase 2. The TEE cluster generates a contract creation transaction to deploy the patched contract.

② Stakeholders migrate the previously-extracted state as well as assets into the patched contract via TEE cluster. After destroying the defective smart contract, all inside assets are temporarily held by the TEE cluster. The temporarily-held assets are transferred into the deployed patched contract according to the stake distribution and previously-extracted values are written into the state variables of the newly-deployed patched contract.

3.3 Building Blocks

3.3.1 *Exception Detector*

For the detection of potential attacks in SolSaviour-protected smart contracts and DeFi, we propose a policy-based exception detector. Two exception detection strategies have been designed for SolSaviour. The first one is to alert when a contract withdrawal exceeds a certain percentage of the contract's total assets within a certain period of time. The exact parameters can be fine-tuned according to the security requirements. The second one is to alert when a contract withdrawal is made to an address that is not a contract stakeholder. Warning messages are published via event messages on the blockchain. These two strategies ensure that stakeholders are able to quickly perceive hazards and react when contracts are at risk of potential asset loss.

The implementation of exception detector is on the contract level. The warning messages exist on the blockchain network and are not actively communicated to the contract stakeholders, who therefore need to implement an active crawler to automatically and continuously monitor event messages. This way, when an exception occurs in a monitored contract, stakeholders can be aware of it and take actions as soon as possible.

3.3.2 *voteDestruct Mechanism*

Currently, a typical method to destroy a contract is to have a privileged destruction function that only specific users can invoke. This approach is not feasible to multi-user smart contract scenarios. For example, assets in a DeFi contract typically belong to different users. Stakeholders of a DeFi have the right to decide whether to destroy the smart

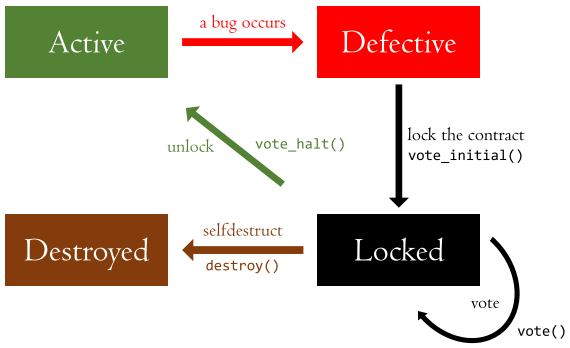


Fig. 3. The state diagram of a voteDestruct-enabled smart contract.

contract. In this case, we propose the voteDestruct mechanism to enable decentralized control of smart contracts and DeFi.

In voteDestruct-enabled contracts, stakeholders can vote on whether to destroy the smart contract and withdraw all internal funds. The voteDestruct mechanism is based on the contract stake distribution, which is recorded by state variables. The more stake a stakeholder controls, the greater its vote weight. Specifically, The voteDestruct is processed in three steps. The contract first forms the stake distribution during its execution. Then, once a vulnerability is exposed, stakeholders can invoke the contract via TEE cluster to vote. After that, the contract stakeholders can invoke the TEE cluster to destroy the defective contract if the support rate exceeds a predefined threshold.

Stake Distribution: Each depositing transaction will be recorded. Specifically, the address of stakeholder and amount of deposited assets are stored in a tuple.

Voting: During the voting process, the smart contract is locked so that no external users can deposit or withdraw assets. This ensures that the stake distribution remains constant throughout the voting. Once completed, the contract stores the percentage of stake on supporting and opposing, respectively. Then, stakeholders can determine whether this contract can be destroyed.

State Transition: The state transition of a voteDestruct-enabled smart contract is depicted in Fig. 3. A smart contract is initially in an active state. Then, an unknown bug is discovered and the contract enters a defective state. Stakeholders can now lock the defective contract using the `vote_initial()` function. For false positive cases, they could unlock the contract via `vote_halt()` function. For true bugs, stakeholders could vote on whether to destroy the defective contract. When the support rate exceeds the threshold, the user can call the `destroy()` function to destroy it.

Destroying: A contract can only be destroyed if the number of votes in favour of destruction exceeds the threshold. Stakeholders can instruct the TEE cluster to invoke the `destroy` function once the voting process completes.

3.3.3 TEE Cluster

Participants can monitor the blockchain for protocol deviations and respond appropriately. In SolSaviour, TEE act as the root of trust. The TEE cluster is independent of the blockchain and can assure faithful implementation. SolSaviour can provide safe destruction, redeployment, and state migration of defective smart contracts by employing the TEE cluster as an independent root of trust. Furthermore, the cluster architecture improves the system's overall failure tolerance. A single point of failure will not influence overall availability.

1: **procedure** BOOTSTRAPPING($\sigma_i, i \in [0, n-1]$)
2: load σ_i
3: broadcast $\{\text{sgx\_quote}_i, K_i\}$
4: verify sgx_quote_j
5: generate $\{K$,addr$\}$
6: **end procedure**
7: **procedure** IDENTITY AUTHENTICATION
8: on receiving a message call tx
9: require(st.map(msg.sender()).st_amount)
10: revert()
11: **end procedure**
12: **procedure** LOCKING DEFECTIVE CONTRACT
13: goto line 8, on receiving $addr$
14: $\text{Sign}_K(\text{tx.payload}(\text{addr}, \text{vote\_initial}))$
15: upload tx, broadcast(++nonce)
16: **end procedure**
17: **procedure** EXTRACTING INTERNAL STATE
18: goto line 8, verify_tx(tx,pub_1,pub_2)
19: d $\leftarrow$ create(tx)
20: mapping stake_dist(addr$\rightarrow$stake_amount)
21: stake_dist $\leftarrow$ addr.st_map_getter()
22: **end procedure**
23: **procedure** DESTROYING DEFECTIVE CONTRACT
24: goto line 8, on receiving $addr$
25: $\text{Sign}_K(\text{tx.payload}(\text{addr}, \text{destroy}))$
26: upload tx, broadcast(++nonce)
27: **end procedure**
28: **procedure** REDEPLOYING PATCHED CONTRACT
29: goto line 8, on receiving C_p
30: $\text{Sign}_K(\text{tx.payload}(C_p))$
31: upload tx, broadcast(++nonce)
32: **end procedure**
33: **procedure** MIGRATING TO PATCHED CONTRACT
34: goto line 8
35: tx.payload(addr_C_p, stake_dist), go to line 15
36: values[]$\leftarrow$stake_dist[]
37: balance[c_id_n]$\leftarrow$values[c_id_n]
38: **end procedure**

Fig. 4. The working logic of TEE cluster. The TEE cluster receives instructions from contract stakeholders and generates transactions to call smart contract functions.

The working logic of distributed TEE cluster is depicted in Fig. 4. Each enclave is denoted as σ_i. We use c to represent a potentially defective smart contract and C_p to denote a patched contract. To demonstrate the difference between intra-TEE cluster communication and TEE cluster-blockchain communication, we use "broadcast" to indicate broadcasting messages inside TEE cluster and "upload" to represent uploading transactions onto the blockchain.

Bootstrapping: Let $\mathbb{G}$ be an Elliptic curve group of order q with generator (base point) G. Each TEE node P_i has the information of $(\mathbb{G}, \text{G}, q)$. A TEE node P_i first chooses a random x_i from $\mathbb{Z}_q^*$ and computing $Q_i = x_i \cdot G$. Each TEE node stores $(\mathbb{G}, \text{G}, q, x)$. Then, each TEE node P_i broadcasts its

TABLE 1
SolSaviour API

SolSaviour API	Inputs	Outputs	API Description
Bootstrap			
new_address	N/A	$\top\|\bot$	Generate a new key pair as well as the blockchain address for the TEE cluster
Deployment			
new_contract	$\mathcal{C}_b$	tx_id	Deploys a compiled smart contract and returns a transaction id
patched_contract	$\mathcal{C}_b$	tx_id	Receives the bytecode-version smart contract, deploys it and returns a transaction id
Message Call			
lock	$\mathcal{C}_{addr}$	tx_id	Receives the address of contract and returns the id of transaction that invokes `vote_initial()`
unlock	$\mathcal{C}_{addr}$	tx_id	Receives the address of contract and returns the id of transaction that invokes `vote_halt()`
destroy	$\mathcal{C}_{addr}$	tx_id	Receives the address of contract and returns the id of transaction that invokes `destroy()`

generated x_i to P_j for every $j \in [n]\backslash\{j\}$. After receiving x_j ($j \in [n]\backslash\{j\}$) from other parties, P_i locally computes $x = \sum_{l=1}^{n} x_l$ and $Q = x \cdot G$. Each party P_i locally stores Q as the ECDSA public key.

Signing: A TEE node P_i locally generates k_i and ρ_i randomly. Then, P_i broadcasts k_i and ρ_i to P_j for every $j \in [n]\backslash\{j\}$. After receiving k_j and ρ_j ($j \in [n]\backslash\{j\}$). Each party can locally compute $k = \sum_{l=1}^{n} k_l$ and $\rho = \sum_{l=1}^{n} \rho_l$. Then, TEE node P_i computes $\tau = k \cdot \rho$ and $R = k \cdot G$. The TEE node P_i can now compute $R = (r_x, r_y)$ and $r = r_x \bmod q$. For a raw transaction m, node P_i generates $\beta = \rho \cdot (m + x \cdot r) \bmod q$. Then, P_i computes $s' = \tau^{-1} \cdot \beta \bmod q$ and $s = min(s, q - s)$. Finally, TEE node P_i outputs (r, s) as the ECDSA signature of the transaction.

Identity Authentication: Identity authentication is a crucial component in TEE cluster, which happens before each external call. Identity authentication restricts that only eligible stakeholders can invoke the TEE cluster. SolSaviour needs to verify that the identity of the object initiating the call is reliable, which reduces the likelihood of an attacker invoking the TEE cluster for dangerous operations, such as DoS attacks. TEE Cluster uses membership proof for identity authentication. It first extracts the address of the stakeholder that initiated the call, which in turn polls the membership proof integrated with the voteDestruct mechanism. Only if the membership proof is true, the TEE cluster determines that the invoking stakeholder is valid and proceeds.

Destroying: TEE cluster allows stakeholders to lock the defective contracts as shown in *line 12-16*. Once an error is detected, the stakeholder can call the TEE cluster to lock the contract. The TEE cluster generates a message call transaction to invoke the `vote_initial()` function and signs it. The signed transaction is uploaded to the blockchain and an incremented nonce is broadcasted inside the TEE cluster.

Before destroying the defective smart contract, the TEE cluster extracts the internal states from it as shown in *line 17-22*. Internal states include state variable values and the stake distribution of stakeholders at the time of locking. The TEE cluster can obtain internal states via `getter` functions and save them locally.

After completing the voting process, stakeholders can instruct the TEE cluster to invoke the `destroy()` function as shown in *line 23-27*. The TEE cluster generates a signed message call transaction destined to the address of the defective smart contract. Provided the amount of stake in favour of destruction exceeds a specified threshold, the contract is allowed to be destroyed.

Redeploying: Contract stakeholders first generate the patch offline. Then, they could invoke the TEE cluster to redeploy a patched smart contract as shown in *line 28-32*. After receiving the patched contract, the TEE cluster generates a contract creation transaction with compiled contract as payload, uploads the signed transaction onto the blockchain, and broadcasts an incremental nonce.

Based on the previously-extracted internal states, stakeholders can call the TEE cluster to migrate them to the patched contract as shown in *line 33-38*. For stake distribution and assets, TEE cluster generates signed message call transactions to invoke the patched contract. In this process, the TEE cluster ensures that states are consistent. The TEE cluster also guarantees the atomicity of execution, i.e. any intermediate state resulting from the call, and the need to provide an effective rollback mechanism in the event of a failed call, allowing the system to revert to the state before the call, eliminating the impact of intermediate state resulting from the call.

3.4 SolSaviour API

We summarize the API of SolSaviour in Table. 1. Contract stakeholders can invoke the functionalities provided by SolSaviour via these interfaces.

SolSaviour API includes 3 types: bootstrap, deployment, and message call. In the bootstrap type, stakeholders can invoke the `new_address` function to generate a new key pair for the agreed account address utilized in the TEE cluster. In the deployment type, there are two functions: `new_contract` and `patched_contract`. The former one is used to deploy a compiled new smart contract, and the later one is used to deploy a patched contract. Both functions return the id of the contract generation transaction that utilized to deploy the contract.

For message call type, stakeholders can invoke them to generate message call transactions to instruct the voteDestruct mechanism. Stakeholders should provide the address of the defective contract as inputs so that TEE cluster know which contract should call. SolSaviour provides 3 message

```
contract voteDestruct_sample {
    struct st_holder
    { uint key_index; uint st_amount; bool voted;}
    mapping(address => st_holder) public st_map;
    address public TEE_addr;
    uint public contract_stake; uint public support_stake;
    enum State {Active, Locked} State public state;

    modifier inState(State _state)
    { require(state == _state); _;}

    constructor() { TEE_addr = msg.sender;}

    function any_payable_function() inState(State.Active)
    public payable {
        st_map[msg.sender].st_amount += msg.value;
        contract_stake += msg.value;}

    function vote_initial() inState(State.Active) public {
        require (msg.sender == TEE_addr);
        state = State.Locked;}

    function vote_halt() inState(State.Locked) public {
        require (msg.sender == TEE_addr);
        state = State.Active;}

    function vote(bool choice) inState(State.Locked) public {
        require(!st_map[msg.sender].voted);
        st_map[msg.sender].voted = true;
        if (choice)
        { support_stake += st_map[msg.sender].st_amount;}}

    function destroy() public {
        require (msg.sender == TEE_addr);
        require (support_stake > (contract_stake * 2 / 3));
        selfdestruct(payable(TEE_addr));}}
```

Fig. 5. A voteDestruct-enabled contract sample.

calls: `lock`, `unlock`, and `destroy`. The `lock` call can invoke the `vote_initial` function in voteDestruct mechanism to lock a defective smart contract, `unlock` can invoke `vote_halt` function to unlock a falsely locked smart contract, and `destroy` function can invoke the `destroy` function in voteDestruct mechanism to destroy a defective smart contract and transfer all assets to TEE cluster. After receiving the invocation from stakeholders, the TEE cluster passes parameters to the transaction generator to generate a signed message call transaction, which is later uploaded onto the blockchain.

4 Implementation

4.1 Destroying

Currently, we can use `selfdestruct` to destroy deployed smart contracts and refund all inside assets. In SolSaviour, we introduce the voteDestruct mechanism to better leverage the `selfdestruct` opcode. A voteDestruct-enabled contract can only be destroyed if and only if most of the stakeholders vote to destroy it. Fig. 5 shows a sample of the voteDestruct mechanism. We emphasize that its implementation does not require new EVM opcodes. It is constructed based on pure Solidity language. Moreover, the voteDestruct mechanism can be implemented in different versions of Solidity with minor modifications.

In the life cycle of a smart contract, contract participants may deposit ethers before a bug is exposed. The voteDestruct mechanism records these participants as stakeholders `st_holder` and the amount of their deposited ethers as `st_amount`. Once the exception detector issues a warning message or a stakeholder notices that the contract has a potential vulnerability, all contract stakeholders can vote to lock the contract. If the support rate exceeds 1/3 (i.e., lock threshold), the contract enters a locked state and no external calls can be executed, except for calls that unlock or destroy the contract. During the locking phase, the contract stakeholders can analyze the exposed vulnerability and develop corresponding patches.

If most stakeholders think that this vulnerability is a false positive case, they can vote to unlock the smart contract. The voting threshold for unlocking a locked contract is the same as the threshold for locking it. If stakeholders think the vulnerability may lead to serious consequences, they could vote to destroy the defective contract. When the support rate exceeds 2/3 (i.e., destroy threshold), the vote is passed and the contract can be destroyed. All internal assets are transferred to the distributed TEE cluster for temporary escrow.

4.2 Patching

In SolSaviour, patches for defective smart contracts are provided by the contract stakeholders. This is because the main purpose of SolSaviour is to provide a framework for securing defective deployed smart contracts and DeFi, rather than providing a system that can automatically generate patches. Smart contract patches can be generated manually using existing tools such as sGuard [12] and SCRepair [13]. Once a patched contract is prepared, the patched contract should be tested thoroughly before deployment by replaying previous related transactions. This can test whether the patched contract functions well and has fixed all related bugs. We propose a patch tester to re-execute non-malicious history transactions on the patched contract and verifies whether the execution results of the old contract and the patched contract are consistent. Any execution discrepancies are scrutinised to determine whether the patch has caused the patched smart contract to function inconsistently with the defective contract. Detailed implementation is provided in Fig. 6.

4.3 Redeploying

4.3.1 *Redeploy a Patched Contract*

In this step, stakeholders can redeploy a patched contract and migrate the internal state from the defective contract to the patched one.

During redeployment, the TEE cluster takes charge of injecting the initial state of the patched contract. The TEE cluster injects a list of stakeholder addresses and the amount of their stakes to the patched contract, which indicates the amount of assets they deposited before contract destruction. Then, the TEE cluster generates a contract creation transaction for the patched contract and broadcast it onto the blockchain. For contract stakeholders, the internal state of the redeployed contract remains the same as the previous defective contract, but SolSaviour has already fixed the vulnerabilities.

```
static bytes C_patched[];
void patchGeneration(bytes C_defective, bytes& C_patched){
    C_patched[0] << sGuard(C_defective);
    C_patched[1] << SCRepair(C_defective);
    C_patched[3] << C_patched;}
bytes patchTest(bytes& C_patched[]){
    string Tx[];
    while(web3.eth.addr){
        Tx += web3.eth.addr.transaction;}
    uint length = Tx.length; uint index;
    uint success[] = 0;
    for(int i =0; i<3; i++){
        for(index=0, index<length; index++){
            if(C_patched[i].exe(Tx[index])){
                success[i]++;}}}
    if(success[0]>success[1]){
        if(success[0]>success[2]) return C_patched[0];
        else return(C_patched[2]);}
    else{
        if(success[1]>success[2]) return(C_patched[1]);
        else return(C_patched[2]);}}
```

Fig. 6. The implementation of patching a defective contract.

4.3.2 State Migration

For migrating states to the newly-patched smart contract, the TEE cluster first extracts required variables from the blockchain. Then, the TEE cluster modifies the patched smart contracts provided by the contract stakeholders. The purpose of this modification is to migrate the internal state from the old, defective contract to the new, patched smart contract. TEE cluster ensures that variable values in the patched contract are the same with before by initializing them. TEE cluster directly transfers all the escrow assets to the newly-deployed contract. Since the stake distribution has been injected by TEE cluster, the ownership of these assets is certain and consistent, as well as their corresponding voting rights. Detailed implementation is provided in Fig. 7.

```
void stateVariableGetter(string addr){
    struct stateVariable{
        string name;
        bytes value;};
    stateVariable states[]; uint index;
    uint length = addr.ABI.length;
    for (index=0, index<length; index++){
        states[index].name = addr.ABI[index].name;
        states[index].value = addr.ABI[index].value;}}
void stakeDistribution(string addr){
    struct stakeDist{
        string addr;
        uint amount;};
    stakeDist stakes[]; uint index;
    uint length = addr.st_map.length;
    for (index=0; index<length; index++){
        stakes[index].addr = addr.st_map[index].key;
        stakes[index].amount = addr.st_map[index].st_amount;}}
void stateMigration(string addrFrom[], string addrTo[],
↪  uint256 values[]){
    require(addrFrom.length == values.length);
    uint256 length = values.length;
    uint i;
    for (i=0; i<length; i++){
        balances[addrTo[i]] = values[i];
        emit Transfer(0x0, addrTo[i], values[i]);}}
```

Fig. 7. The implementation of recovering a patched contract.

5 Security Analysis

5.1 Threat Model

Our threat model considers the security of SolSaviour from following three perspectives.

Host. We assume hosts are potentially malicious. They may delay messages between the blockchain and its hosted enclaves for a period of time. We also assume an adversary $\mathcal{A}$ that can corrupt up to t of n hosts in the TEE cluster.

Enclave. We assume the attested execution result of enclave is trusted, which means the malicious host cannot tamper with the code and data inside enclaves. However, we assume the enclave may suffer from some confidentiality problem.

Stakeholder. We assume the stakeholders are rational and potentially malicious. If there are benefits, they may deviate from the protocol execution and try to steal funds that belong to others. Stakeholders are greedy, if a smart contract is under attack, they will steal assets that belong to others.

5.2 Threat Analysis

In this section, we analyze the security of SolSaviour in three aspects: balance security, correctness, and fairness.

5.2.1 Balance Security

The balance security of SolSaviour is twofold. First, honest stakeholders won't lose assets except necessary transaction fees as long as stakeholders behave honestly. Second, the stake distribution remains the same in the old defective contract and new patched contract.

First, we consider the case that a smart contract is in the locked state. If contract stakeholders cannot reach an agreement on a patched contract, the defective smart contract remains locked and reject all external calls except ones from the TEE cluster. In this case, no assets can be stolen by attackers. If stakeholders agree on a patched contract, the TEE cluster proceeds to conduct the state migration.

We then prove that the assets security is preserved during the state migration process. Before migrating assets into new patched contract, assets are held in a blockchain account controlled by the TEE cluster. As assumed before, attackers cannot control more than t of n TEE nodes. In this case, attackers cannot withdraw assets from the new patched contract with legitimate signature so that cannot steal assets. Moreover, there is no way for attackers to tamper with the code inside TEE cluster since the execution logic of the enclave is fixed once it has been encapsulated. As a result, attackers are unable to alter the TEE cluster's state migration logic and transfer assets to their accounts.

Finally, we discuss the consistency of stake distribution between the defective contract and patched contract. The history of defective smart contracts is publicly available on the blockchain. The stake distribution a defective contract can be determined by looking up its transaction history. Additionally, the TEE cluster can crawl the contract history to ascertain the internal variable values for the defective contract. In this way, TEE cluster can ensure a safe migration of contract internal state from the old vulnerable smart contract to the new patched one. After state migration, due

to the tamper-evident nature of on-chain data, assets cannot be stolen once they are transferred into a new contract.

5.2.2 Correctness

Intuitively, correctness states that all honest parties can output correct results, namely successfully patching and transmitting the assets from defective smart contracts to the patched contracts. For the proof of correctness, we divide it into two parts: bootstrapping of the TEE cluster and destruction of a defective contract.

First, we prove that the bootstrapping process of TEE cluster satisfies correctness with threshold t, which indicates that TEE nodes can reach an agreement on a key when there are at most t corrupted parties among n TEE nodes. During the bootstrapping process, once a node U_i receives other node's K_j, it can verify it. If the check fails for index j, U_i can broadcast a complaint against node U_j. If more than t nodes complain about a node U_j, that node is recognized as disqualified. Each node stores a node set `QUAL` for all qualified nodes. In this case, they generate the key based on nodes inside `QUAL`. As all honest nodes construct identical `QUAL`, they can generate the same key and derive the same blockchain address. Thus, we can show that TEE cluster can correctly complete the bootstrapping process for a given attacker threshold.

Second, we prove that the destruction of defective smart contracts satisfies correctness. We consider rational stakeholders, that means they behave maliciously if they think their behaviour is more profitable. In this case, we analyze the choice of rational stakeholders in two scenarios. When a smart contract is exposed to a vulnerability or under attack, stakeholders have two choices, attacking or locking. As stakeholders have some assets inside defective smart contracts, the best way is to lock contract as soon as possible. In the scenario that a defective contract is successfully locked, the best way is also to prepare a patched contract for deployment as soon as possible. This is because stakeholders have assets inside locked smart contracts, which do no output interest. Only when smart contracts are active, stakeholders can use their assets for arbitrage or other financial services.

5.2.3 Fairness

Fairness indicates that our proposed voteDestruct mechanism is fair. All stakeholders cannot have more power than the amount of their controlled stake. For the proof of fairness, even if an attacker holds a certain amount of stake in the defective contract and has the right to vote, it cannot maliciously manipulate the voting result or prevent the destruction of the defective smart contract.

We first analyze the case where there are some malicious stakeholders. As malicious stakeholders are profit oriented, what they want is to steal the assets from honest stakeholders. However, honest stakeholders can always safely exit a smart contract as long as their cumulative stake amount exceeds the specified destroy threshold. In SolSaviour, a smart contract has three states: active (but potentially defective), locked, and destroyed.

In locked state, a contract is protected by blockchain miners that reject all function calls except those initiated from the TEE cluster. In this case, malicious stakeholders cannot steal assets. In destroyed state, assets in a contract are held by the TEE cluster, so that malicious stakeholders cannot profit either. The only chance for malicious stakeholders to profit is during the active (but potentially defective) state. In SolSaviour, the threshold of required stake amount to lock a contract is $1/3$. Only when the amount of stake held by malicious stakeholders exceeds $2/3$, they can prevent the contract from entering the locked state.

During the active (but potentially defective) state, when a hidden vulnerability is exposed, malicious stakeholders can try to prevent the contract from entering the locked state and exploit the vulnerability. However, due to the unknown feature of the vulnerability, it may cause the contract to a deadlock state, which is also unprofitable for malicious stakeholders. Therefore, it is also unprofitable for malicious stakeholders to prevent locking defective contract.

6 Discussion

6.1 Limitations and Security Risks

In this section, we discuss the limitations and security risks of SolSaviour. One of the main limitations of SolSaviour is that it can only protect contracts that have integrated the voteDestruct mechanism. Due to the tamper-proof feature of the blockchain, SolSaviour cannot provide the defence mechanism for active smart contracts that have already been deployed. As TEE is a technology still under development, there may be unknown vulnerabilities. TEE is therefore at risk and newly discovered TEE vulnerabilities could compromise the security of the entire system and the in-contract assets.

6.2 Recovering Patched Contract without TEE Escrow

6.2.1 Implementation

Considering the risks to the security of assets temporarily held in the TEE cluster, we introduce a new way to recover a defective smart contract to a patched contract directly.

Setup Phase. In this way, SolSaviour first locks a potentially defective contract to prevent it from further attacks. Then, stakeholders develop a patched smart contract and provide it to the TEE cluster. The TEE cluster first deploys the patched contract onto the blockchain. After deploying the patched smart contract, the voteDestruct mechanism could set the parameter of the `destroy` function to the address of the patched contract. In this way, the assets are directly transferred from the defective smart contract to the patched contract without interaction of the TEE cluster. In this way, even the serious problem such as key leakage of the account controlled by the TEE cluster, the assets are still under protection.

Recovering Phase. Unlike reverting to a TEE cluster, this approach eschews the use of TEE and therefore its state transfer behaviour cannot be implemented through TEE. We then need to implement complex state migration logic in the smart contract. Specifically, we need to store the complete state variables of the smart contract and the take distribution of the assets stored in the take holders, and we need to have transfer logic that acts on top of these state variables. We consider using a bridging contract to accomplish this

step, i.e. deploying a smart contract dedicated to state migration, in which the state variables and take distribution associated with the defective smart contract are stored, and in the constructor of the patched contract, allowing it to read and write the state variables in the contract. In this way, we can achieve state transfer for smart contracts without the need for TEE intervention. In contrast, this approach requires significant gas consumption to maintain the various operations and data stores. The cost required to implement contract recovery on the public chain is significantly higher due to the addition of bridging contracts and the corresponding storage of state variables. However, the advantages of this approach are clear: by avoiding temporary asset hosting of TEE clusters, this approach significantly reduces the attack surface of the system and eliminates the risk of security issues that may arise from the TEE itself.

6.2.2 Comparison

In summary, there are advantages and disadvantages to both recovering to the TEE cluster and recovering to the patched contract, the former being more gas efficient but less secure than the latter, and the latter being secure, but the cost of protecting the smart contract is greatly increased by the high gas consumption it entails. The latter is secure, but the high gas consumption associated with it can add significantly to the cost of protecting smart contracts. Therefore, when faced with a specific smart contract, the contract developer needs to make a trade-off between security and cost and choose the best method to protect the smart contract.

7 Experiment

In our prototype of SolSaviour, the voteDestruct mechanism is implemented in Solidity and the TEE cluster is implemented based on Intel SGX with around 2000 LOC. Four nodes are set up in the TEE cluster. The experiments are conducted in two aspects: effectiveness and performance.

7.1 Dataset Collection

To accurately evaluate the effectiveness and performance of SolSaviour, we collected ordinary and DeFi contracts that were exposed to vulnerabilities. Some contracts have experienced real attacks that have caused asset losses, some have not been exploited but their defects have been confirmed. The reason we chose these contracts is that we want to prove the effectiveness of SolSaviour with the deduction of loss for these contracts when they are under the protection of SolSaviour.

Our collected contracts are the DAO [1], PoWH Coin [20], 1st [18] and 2nd [19] Parity Multisig Wallet, King of Ether [14], Bancor Exchange [21], GovernMental [15], and Rubixi [16]. We also collected DeFi contracts that were exposed to some severe bugs such as SushiSwap [22], ENS Name Wrapper [17], Fei [2], and Uniswap [23]. We list these contracts in Table. 2, accompanying with contract vulnerability type and caused damage as well as losses in monetary form (if so).

For our collected contracts, we also prepare corresponding voteDestruct-enabled contracts and patched contracts. The voteDestruct mechanism is injected on the source code level. As collected contracts are written in different versions of Solidity, we make minor modifications to make our voteDestruct mechanism compatible in all versions of Solidity. For patched contract, our collected contracts are also patched manually by modifying the code. We also ensure that the compiled voteDestruct-enabled contracts and patched contracts following the same version of Solidity as original contracts.

SolSaviour is then applied to these generated comparative smart contracts so that we could verify the effectiveness and performance by validating the results. We can verify whether the voteDestruct mechanism is effective and that the patched smart contract redeployed by SolSaviour fixed vulnerability.

7.2 Effectiveness

The effectiveness of SolSaviour is evaluated from two perspectives: qualitative and quantitative.

For qualitative part, we check whether we can leverage SolSaviour to safely exit from all collected defective contracts, refund locked assets back to stakeholders, and redeploy a patched contract. To test whether SolSaviour can recover a buggy smart contract, we generate a large and representative evaluation dataset by collecting transactions sent to the collected contracts from the Ethereum. Replaying those transactions and observing outcomes can check the functionality and defence of patched contracts. Specifically, we test whether SolSaviour can successfully destroy a defective smart contract with voteDestruct mechanism and redeploy a patched one with TEE cluster. In addition, we test whether the TEE cluster can successfully migrate the previous state to the new contract to ensure the state consistency.

For the quantitative part, we set up two contract instances for each collected defective contract: an original contract instance and a SolSaviour-protected instance. We compare the loss between the original one and SolSaviour-protected one. We think the effectiveness of SolSaviour is reflected in the loss deduction when a vulnerable smart contract is protected by SolSaviour. That's the reason we compare the loss of smart contracts in difference settings. For the original one, we also record the loss when taking traditional defence measures and doing nothing. We test to what extent can SolSaviour save loss when facing different vulnerabilities.

7.2.1 Qualitative

We evaluate the effectiveness of SolSaviour in three aspects: successful state migration, identical functionalities, and successful defence. For each contract, we use Ganache to simulate 10 accounts, who play the role of contract stakeholders and each has deposited 100 ethers. Then, a random stakeholder initializes the `vote_initial` and provides a patched contract to the TEE cluster. In our experiments, we omit the security assumption of potential malicious stakeholders and assume all of them will vote to destroy the defective contract. Once the voting completes, the TEE cluster destroys the defective contract, redeploys a patched one, and conducts state migration.

By checking the patched contracts deployed by the TEE cluster, we can evaluate whether state migration successes. A successful state migration means the internal states of

TABLE 2
The List of Contracts That Have Been Attacked or Exposed to Serious Bugs

Contract	Address	Vulnerability Type	Caused Damage
King of Ether [14]	0x2464d1d97f8D0180CFaD67BdB19bc30ccA69DdA0	Unchecked Return Values	Ownership Loss
GovernMental [15]	0xF45717552f12Ef7cb65e95476F217Ea008167Ae3	Timestamp Dependence	DoS
Rubixi [16]	0xe82719202e5965Cf5D9B6673B7503a3b92DE20be	Bad Constructor	Ownership Loss
ENS Name Wrapper [17]	0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e	Access Control	Domain Ownership Loss
1st Parity Multisig [18]	0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4	Delegatecall	153,037 ETH Loss ($31M)
2nd Parity Multisig [19]	0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4	Denial of Service	513,774.16 ETH Locked ($300M)
The DAO [1]	0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413	Reentrancy	3.6M ETH Loss ($150M)
PoWH Coin [20]	0xA7CA36F7273D4d38fc2aEC5A454C497F86728a7A	Integer Underflow	866 ETH Loss ($800k)
Bancor Exchange [21]	0x1F573D6Fb3F13d689FF844B4cE37794d79a7FF1C	Front Running	Economic Earns ($150)
SushiSwap [22]	0x6B3595068778DD592e39A122f4f5a5cF09C90fE2	Supply Chain Attack	864.8 ETH
Fei [2]	0x956F47F50A910163D8BF957Cf5846D573E7f87CA	Price Manipulation	60k ETH at risk
Uniswap [23]	0x1f9840a85d5aF5bf1D1762F925BDADdC4201F984	ERC777 Reentrancy Exploit	$320M

buggy contract and patched contract are identical. Not only the stake distribution, but also the ownership. We check this by letting each stakeholder withdraw their previously-deposited ethers. We found that stakeholders can withdraw their assets successfully from all contracts. Then, we check the functionality and defence of patched contracts by replaying collected transactions. We compare the execution results of patched contract with buggy contract's history state transition.

7.2.2 Quantitative

We quantitatively evaluate the effectiveness of SolSaviour by attacking and recovering defective contracts with SolSaviour simultaneously. Then, we evaluate to what extent can SolSaviour reduce loss. Since different contracts are tested in different scenarios, where the amount of loss are different, we use a two-tuple (loss, percentage) to show results.

For the DAO contract, we simulate a scenario, where the defective deployed contract contains 100 ethers. Then, we start to attack and recover it at the same time. Attackers can arbitrarily withdraw ethers until honest stakeholders lock the contract. Then, we tried to refund all locked ethers to stakeholders and calculate the loss. Similar steps are conducted to evaluate the loss when using SolSaviour. For the PoWH coin contract, since the real attack transactions are limited, we simply replay these attack transactions and check the execution results. We also test the loss when doing nothing and taking traditional defence. Each contract are tested 5 times and the average loss is recorded. The results are listed in Table. 3.

7.3 Performance

7.3.1 Contract Size Increase

In this section, we evaluate the additional code required to use SolSaviour. On Ethereum, deploying smart contracts consumes gases, which are proportionally to the size of the deployed contract. In SolSaviour, as the voteDestruct mechanism is implemented inside contracts, extra codes are introduced. The manually-generated patch may also increases the code of contracts. Moreover, the method on recovering patched contract without TEE also introduces extra codes in the contract. Results are summarized in Fig. 8. The code size of the original version of collected defective contracts are listed as the basis. Each subplot has six bars. The left three bars represent the size of contract when recovering with TEE, while the right three bars represent the size of contract when recovering without TEE. In each subplot with three bars, from left to right are the size of the original contract, the size voteDestruct-enabled contract and the size of the patched contract.

However, since patches are generated manually, it is impossible to determine the performance of the system in terms of the number of lines of code added by the patch. The only additional code introduced by the system is the

TABLE 3
Comparison of Losses/Damages in an Attack Against Our Collected Contracts

Contract	Actual	Traditional	SolSaviour
King of Ether [14]	Lose Onwership	No Mitigation	Fix
GovernMental [15]	Lose Ownership	No Mitigation	Fix
Rubixi [16]	Lose Ownership	No Mitigation	Fix
ENS Name Wrapper [17]	Lose Ownership	No Mitigation	Fix
1st Parity Multisig [18]	100	100	0
2nd Parity Multisig [19]	100	100	0
The DAO [1]	100	48.6	6.5
PoWH Coin [20]	100	100	0
Bancor Exchange [21]	100	69.6	0
SushiSwap [22]	100	100	3.3
Fei [2]	100	61.2	2.3
Uniswap [23]	100	54.3	4.6

"Actual" indicates that no action is taken; "Traditional" indicates that traditional defence methods are used; "SolSaviour" indicates that the contract is maintained by SolSaviour.

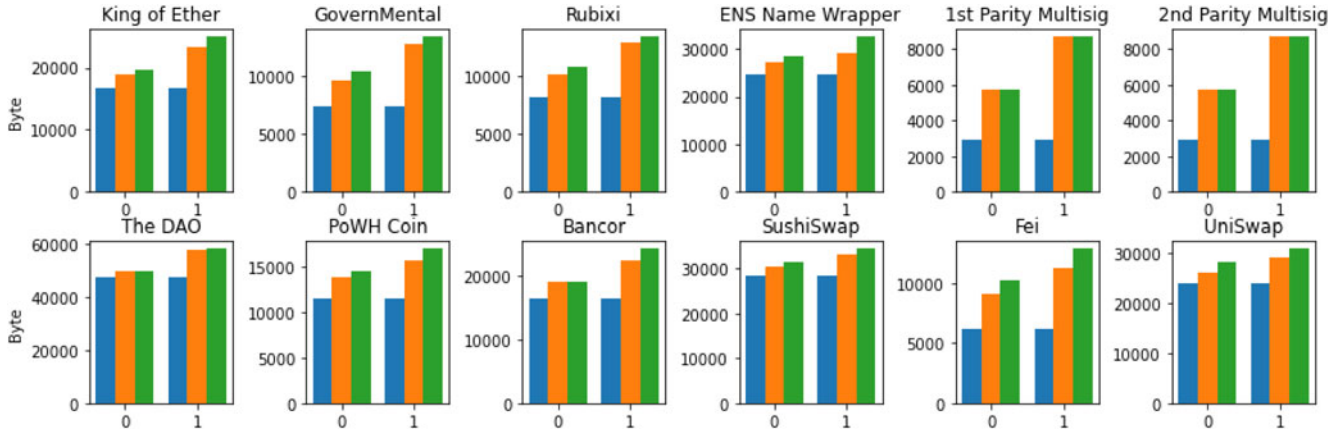


Fig. 8. Contract size increased in SolSaviour. The left three bars represent using TEE cluster for asset escrow while the right three bars using patched contract. Blue bar indicates original contract, orange bar indicates voteDestruct-enabled contract, and green bar indicates patched contract.

voteDestruct framework that makes defective contracts have the ability to iterative upgrades under SolSaviour. In this case, we compare the size of compiled contract. We found that voteDestruct mechanism introduces limited size to the original contract. These code size increases are worth compared to the security enhancement that SolSaviour brings. For Parity Miltisig contract, we note that injecting voteDestruct mechanism naturally resolves the vulnerability so that the patched contract and voteDestruct-enabled contract have the same size.

7.3.2 Gas Consumption

In this section, we evaluate the additional gas consumption incurred by SolSaviour, which mainly arises from two aspects: the voteDestruct mechanism and the redeployment of the patched contract. We also evaluate the gas consumption on redeploying a patched contract without TEE asset escrow. Results are summarized in Fig. 9. The gas consumption to deploy the original version of collected defective contracts are evaluated as the basis. Each subplot has six bars. The left three bars represent the gas consumption of original contract, voteDestruct-enabled contract, and patched contract respectively when letting TEE conduct asset escrow. By contrast, the right three bars represent the gas consumption when recovering without TEE asset escrow.

For voteDestruct mechanism, the evaluation is conducted by deploying our prepared voteDestruct-enabled contracts. From the results, we can see the voteDestruct mechanism introduced minimal gas overhead. The gas cost are mainly introduced by additional storage of state variables and corresponding logic. Storing data on Ethereum is expensive, which leads to a lot of gases to be consumed. For the redeployment of patched contract, the extra gas consumption are mainly introduced by the patcg. As the gas consumption depends on the contract size, namely the size of original contract plus the size of the patch as well as the voteDestruct mechanism for future protection. As shown in the results, the overhead introduced by the patch is not stable. This is because different contracts require different type of patches.

7.3.3 TEE Cluster Overhead

In SolSaviour, state migration and asset escrow are conducted by TEE cluster. We therefore evaluate the overhead introduced by TEE cluster. We build a Ethereum private network with four nodes (i.e., node A, B, C, and D), each is installed with an Ethereum endpoint. We record the number of blocks mined by them in one day. Then, we initialize the enclave in one node and monitor the blocks mined by each node. After that, we sequentially initialize enclaves in

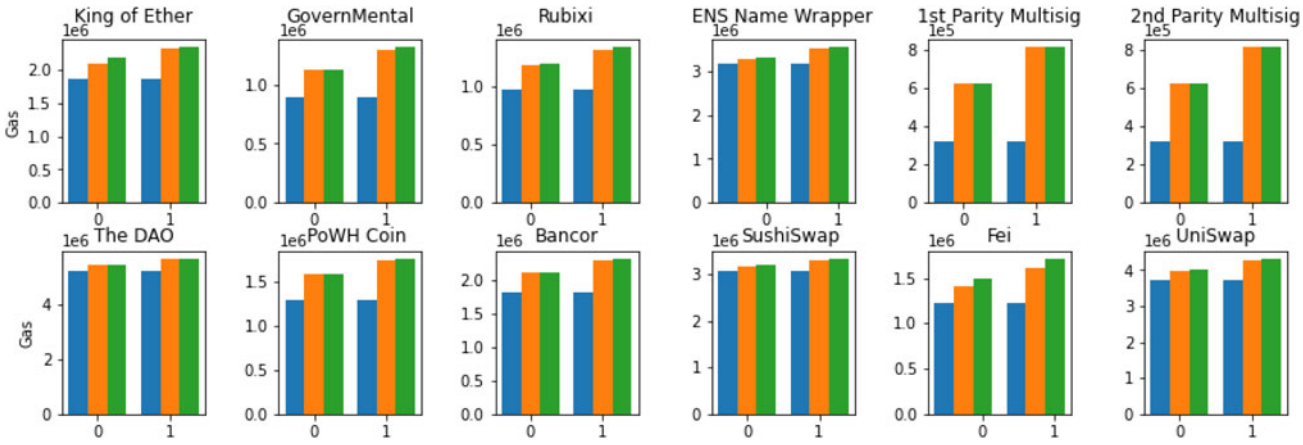


Fig. 9. Gas consumption increased in SolSaviour. The left three bars represent using TEE cluster for asset escrow while the right three bars using patched contract. Blue bar indicates original contract, orange bar indicates voteDestruct-enabled contract, and green bar indicates patched contract.

TABLE 4
The Overhead of TEE Cluster, Which is Counted in the Number of Blocks Mined by Different Combinations of TEE Nodes

Node		A	AB	ABC	ABCD
All	5780	5685(-1.6%)	5507(-4.7%)	5469(-5.3%)	5391(-6.7%)
A	1451	1368(-5.7%)	1312(-9.5%)	1340(-7.6%)	1341(-7.6%)
B	1446	1439	1325(-8.3%)	1339(-7.4%)	1335(-7.7%)
C	1438	1441	1439	1351(-6.1%)	1349(-6.2%)
D	1445	1437	1431	1439	1366(-5.5%)

the other three nodes and add them to the TEE cluster. During this time, we continuously monitor the number of blocks mined by each node. The mining difficulty remains the same during this experiment. We summarize the results in Table. 4. As we can see, for nodes without TEE cluster, they can mine around 1440 blocks per day, which satisfies the Ethereum blockchain generation speed, namely a block per 15 seconds. For nodes with TEE cluster, the mining rate is slightly affected. The impact is greatest when only half nodes participate the TEE cluster, and tends to become smaller when all nodes initialize TEE.

8 Related Work

In this section, we first present work on detecting contract vulnerabilities, protecting deployed contracts, and generating patches for vulnerable contracts. Then, we introduce recent work on investigating the security of DeFi. Finally, we present work on combining blockchain and TEE.

8.1 Smart Contract Vulnerabilities

Most work on smart contract vulnerability detection relies on techniques used in traditional software analysis. Luu et al., proposed Oyente [3] based on symbolic execution to automate the reentrancy bug detection. After that, lots of symbolic execution tools are proposed such as Osiris [24], teEther [25], Maian [26], EthBMC [27], and Zeus [28]. Feist et al., proposed Slither [5] to analyze the contract on source code level. Tsankov et al., presented Security [7] to analyze the contract on bytecode level. Brent et al., proposed Ethainter [6] to conduct information flow analysis to reveal composite vulnerabilities. Chen et al., proposed SODA [29], which accepts user-defined vulnerability pattern. Pan et al., proposed ReDefender [30], which detects reentrancy vulnerabilities with fuzz testing. Furthermore, the semantics of Solidity was formalized [31]. However, these proposed detection methods still have limitations, such as the inability to identify unknown vulnerabilities. Therefore, there has also been some work focused on protecting deployed smart contracts.

For contract defence, Rodler et al., proposed *Sereum* [32] to defend against reentrancy exploits through analyzing the EVM execution traces. Ferreira et al., proposed ÆGIS [33] to defend deployed contracts against known attacks. Specifically, it records a number of known contract attack execution traces through a proposed domain-specific language and integrates within the EVM to revert the execution of transactions that match the attack traces. Ellul et al., proposed a runtime verification mechanism [34] to ensure that violating party provides insurance for correct behavior. Li et al., proposed Solythesis [35] to address the high overhead in runtime validation.

For contract patch, Yu et al., proposed SCRepair [13] to detect and repair bugs in smart contracts before deployment. Zhang et al., proposed SMARTSHIELD [36], which leverages bytecode rewriting techniques to fix contract vulnerabilities. Bytecode rewriting technique was also used in the EVMPatch [37] to repair defective contracts.

8.2 DeFi Vulnerabilities

As DeFi has evolved greatly, more work has been carried out to investigate its security. Qin et al., explored attack vectors that leverage the Flash Loan in [38]. They quantitatively show how transaction atomicity affects arbitrage revenue. Zhou et al., proposed sandwich attack to exploit the information that may change the asset price [39]. Zhou et al., worked on crafting profitable transactions across the intertwined DeFi platforms automatically [40]. They investigated two methods that focus on arbitrage and complicated DeFi setting, respectively. Qin et al., analyzed the danger of blockchain extractable value (BEV) quantitatively [41]. They deduced the amount of possible profits for the attacker in the case of sandwich attacks, liquidations, and decentralized exchange arbitrage.

8.3 Smart Contract and TEE

There is a range of work focused on offloading the execution of smart contracts into the TEE, which enables the implementation of private and high-performance smart contracts. Cheng et al., proposed Ekiden [42], whose execution of smart contracts is deployed inside enclaves. With the enclave's public key, users can encrypt the input data in a confidential message call transaction. Das et al., proposed FastKitten [43] to enable the execution of complex, high-performance smart contracts on Bitcoin. In FastKitten, enclaves take charge of executing smart contracts and generating state transitions, which are recorded by the Bitcoin. FastKitten can extend its work to execute smart contracts on more blockchains which were designed to only support naive transactions. Liu et al., proposed Saber [44], a parallel and asynchronous execution paradigm of smart contracts boosted by TEE.

In addition, the combination of blockchain and TEE shows promise in many other areas. Zhang et al., proposed Town Crier to address the problem that smart contracts running on the blockchain cannot access information in a trusted way [45]. Zhang et al., proposed REM [46], a resource efficient mining algorithm that works on useful computation workloads. Considering the problem that traditional PoW consensus algorithms consume lots of power, REM converts the traditional meaningless hash computation into meaningful computation workload, accompanied by a SGX-based verification mechanism. Lind et al., proposed Teechain [47], a layer-2 payment network that can process off-chain transactions asynchronously with TEE. Li et al., proposed Pistis [48], which leverages smart contract and TEE to issue trusted and authorized certificates to address the problem of rogue certificates. Matetic et al., proposed Bite [49], a SGX-based lightweight node, to address

the privacy issue in lightweight nodes. In BITE, full nodes leverage SGX enclaves to process privacy preserving requests from lightweight clients.

9 CONCLUSION

In this paper, we present SolSaviour that can protect deployed but defective smart contracts and DeFi protocols. Compared with existing work that requires a trusted third party to redeploy patched contract and can only migrate contract data, SolSaviour can achieve effective migration of contract assets and does not require the involvement of a trusted third party. SolSaviour enables the offline patching of defective smart contracts through the decentralized control provided by voteDestruct mechanism and the state migration provided by TEE cluster. For all collected contracts and DeFi, our experiment results demonstrate that SolSaviour can effectively repair and recover all of them with affordable overhead.

REFERENCES

[1] V. Buterin, "Critical update re: Dao vulnerability," 2016. [Online]. Available: https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/

[2] R. Behnke, "Explained: The Fei protocol hack (April 2022)," 2022. [Online]. Available: https://halborn.com/explained-the-fei-protocol-hack-april-2022/

[3] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 254–269.

[4] K. Bhargavan et al., "Formal verification of smart contracts: Short paper," in *Proc. ACM Workshop Program. Lang. Anal. Secur.*, 2016, pp. 91–96.

[5] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. IEEE/ACM 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2019, pp. 8–15.

[6] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 454–469.

[7] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 67–82.

[8] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.*, 2020, pp. 778–788.

[9] R. Behnke, "The Nomad bridge hack: A deeper dive," 2022. [Online]. Available: https://halborn.com/the-nomad-bridge-hack-a-deeper-dive/

[10] Z. Li, Y. Zhou, S. Guo, and B. Xiao, "SolSaviour: A defending framework for deployed defective smart contracts," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2021, pp. 748–760.

[11] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Trans. Softw. Eng.*, vol. 48, no. 1, pp. 327–345, Jan. 2022.

[12] J. Feist, G. Grieco, and A. Groce, "sGUARD: Towards fixing vulnerable smart contracts automatically," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 1215–1229.

[13] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart contract repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 1–32, 2020.

[14] K. Team, "King of ether throne post-mortem investigation," 2016. [Online]. Available: https://www.kingoftheether.com/postmortem

[15] u/ethererik, "GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas," 2016. [Online]. Available: https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/

[16] Katatsuki, "Re: Hi! My name is Rubixi. I'm a new Ethereum Doubler. Now my new home - Rubixi.tk," 2016. [Online]. Available: https://bitcointalk.org/index.php?topic=1400536.60

[17] samczsun, "The dangers of surprising code," 2021. [Online]. Available: https://samczsun.com/the-dangers-of-surprising-code/

[18] H. Qureshi, "A hacker stole 31m of ether - how it happened, and what it means for ethereum," 2017. [Online]. Available: https://www.freecodecamp.org/news/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce/

[19] A. Akentiev, "Parity multisig hacked. Again," 2017. [Online]. Available: https://medium.com/chain-cloud-company-blog/parity-multisig-hack-again-b46771eaa838

[20] E. Banisadr, "How 800k Evaporated from the PoWH coin ponzi scheme overnight," 2018. [Online]. Available: https://medium.com/@ebanisadr/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530

[21] I. Bogatyy, "Implementing ethereum trading front-runs on the bancor exchange in Python," 2017. [Online]. Available: https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798

[22] samczsun, "Two rights might make a wrong," 2021. [Online]. Available: https://samczsun.com/two-rights-might-make-a-wrong/

[23] PeckShield, "Uniswap/Lendf.me hacks: Root cause and loss analysis," 2020. [Online]. Available: https://peckshield.medium.com/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09

[24] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 664–676.

[25] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1317–1333.

[26] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 653–663.

[27] J. Frank, C. Aschermann, and T. Holz, "EthBMC: A bounded model checker for smart contracts," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 2757–2774.

[28] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–12.

[29] T. Chen et al., "Soda: A generic online detection framework for smart contracts," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–17.

[30] Z. Pan, T. Hu, C. Qian, and B. Li, "Redefender: A tool for detecting reentrancy vulnerabilities in smart contracts effectively," in *Proc. IEEE 21st Int. Conf. Softw. Qual., Rel. Secur.*, 2021, pp. 915–925.

[31] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *Proc. IEEE Symp. Secur. Privacy*, 2020, pp. 1695–1712.

[32] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proc. Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[33] C. Ferreira Torres, M. Baden, R. Norvill, B. B. Fiz Pontiveros, H. Jonker, and S. Mauw, "Ægis: Shielding vulnerable smart contracts against attacks," in *Proc. 15th ACM Asia Conf. Comput. Commun. Secur.*, 2020, pp. 584–597.

[34] J. Ellul and G. J. Pace, "Runtime verification of ethereum smart contracts," in *Proc. 14th Eur. Dependable Comput. Conf.*, 2018, pp. 158–163.

[35] A. Li, J. A. Choi, and F. Long, "Securing smart contract with runtime validation," in *Proc. 41st ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2020, pp. 438–453.

[36] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *Proc. IEEE 27th Int. Conf. Softw. Anal., Evol. Reengineering*, 2020, pp. 23–34.

[37] M. Rodler, W. Li, G. O. Karame, and L. Davi, "EVMPatch: Timely and automated patching of ethereum smart contracts," in *Proc. 30th USENIX Secur. Symp.*, 2021, pp. 1289–1306.

[38] K. Qin, L. Zhou, B. Livshits, and A. Gervais, "Attacking the defi ecosystem with flash loans for fun and profit," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2021, pp. 3–32.

[39] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, "High-frequency trading on decentralized on-chain exchanges," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 428–445.

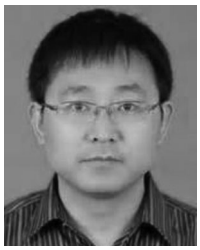
[40] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," in *Proc. IEEE Symp. Secur. Privacy*, 2021, pp. 919–936.

[41] K. Qin, L. Zhou, and A. Gervais, "Quantifying blockchain extractable value: How dark is the forest?," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 198–214.

[42] R. Cheng et al., "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2019, pp. 185–200.
[43] P. Das et al., "Fastkitten: Practical smart contracts on bitcoin," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 801–818.
[44] J. Liu, P. Li, R. Cheng, N. Asokan, and D. Song, "Parallel and asynchronous smart contract execution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 5, pp. 1097–1108, May 2022.
[45] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 270–282.
[46] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. Van Renesse, "REM: Resource-efficient mining for blockchains," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1427–1444.
[47] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch, "Teechain: A secure payment network with asynchronous blockchain access," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 63–79.
[48] Z. Li, H. Wu, L. H. Lao, S. Guo, Y. Yang, and B. Xiao, "Pistis: Issuing trusted and authorized certificates with distributed ledger and TEE," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 7, pp. 1636–1649, Jul. 2022.
[49] S. Matetic, K. Wüst, M. Schneider, K. Kostiainen, G. Karame, and S. Capkun, "Bite: Bitcoin lightweight client privacy using trusted execution," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 783–800.

Zecheng Li received the BEng degree from the School of Information Science and Technology, Southeast University, Nanjing, China, in 2017 and the PhD degree from the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, in 2022. His research interest lies in the network security, blockchain security, and smart contract security.

Bin Xiao (Senior Member, IEEE) received the BSc and MSc degrees in electronics engineering from Fudan University, China, and the PhD degree in computer science from University of Texas at Dallas, USA. He is a full professor with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong. After his PhD graduation, he joined the Department of Computing of the Hong Kong Polytechnic University as an assistant professor. His research interests include AI and network security, data privacy, and blockchain systems. He published more than 180 technical papers in international top journals and conferences. Currently, he is the associate editor for *IEEE Internet of Things Journal*, *IEEE Transactions on Cloud Computing*, and *IEEE Transactions on Network Science and Engineering*. He is the vice chair of IEEE ComSoc CISTC committee. He has been the symposium track co-chair of IEEE ICC 2020, ICC 2018 and Globecom 2017, and the general chair of IEEE SECON 2018. He is the member of ACM and CCF.



Songtao Guo (Senior Member, IEEE) received the BS, MS and PhD degrees in computer software and theory from the Chongqing University, Chongqing, China, in 1999, 2003 and 2008, respectively. At present, he is a full professor with Chongqing University, China. His research interests include wireless sensor networks, wireless ad hoc networks and parallel and distributed computing. He has published more than 80 scientific papers in leading refereed journals and conferences. He has received many research grants as a principal investigator from the National Science Foundation of China and Chongqing and the Postdoctoral Science Foundation of China.

Yuanyuan Yang (Fellow, IEEE) received the BEng and MS degrees in computer science and engineering from Tsinghua University, Beijing, China, and the MSE and PhD degrees in computer science from Johns Hopkins University, Baltimore, Maryland. She is a SUNY distinguished professor of computer engineering and computer science and the associate dean for Academic Affairs in the College of Engineering and Applied Sciences with Stony Brook University, New York. Her research interests include data center networks, cloud computing and wireless networks. She has published more than 380 papers in major journals and refereed conference proceedings and holds seven US patents in these areas. She is currently the associate editor-in-chief for *IEEE Transactions on Cloud Computing*. She has served as an associate editor-in-chief and an associate editor for *IEEE Transactions on Computers* and associate editor for *IEEE Transactions on Parallel and Distributed Systems*. She has also served as a general chair, program chair, or vice chair for several major conferences and a program committee member for numerous conferences.