SolSaviour: A Defending Framework for Deployed Defective Smart Contracts

Zecheng Li
cszcli@comp.polyu.edu.hk
The Hong Kong Polytechnic University
Hong Kong, China

Yu Zhou
csyzhou@comp.polyu.edu.hk
The Hong Kong Polytechnic University
Hong Kong, China

Songtao Guo
guosongtao@cqu.edu.cn
Chongqing University
Chong Qing, China

Bin Xiao*
b.xiao@polyu.edu.hk
The Hong Kong Polytechnic University
Hong Kong, China

ABSTRACT

A smart contract cannot be modified once deployed. Bugs in deployed smart contracts may cause devastating consequences. For example, the infamous reentrancy bug in the DAO contract allows attackers to arbitrarily withdraw ethers, which caused millions of dollars loss. Currently, the main countermeasure against contract bugs is to thoroughly detect and verify contracts before deployment, which, however, cannot defend against unknown bugs. These detection methods also suffer from possible false negative results.

In this paper, we propose SolSaviour, a framework for repairing and recovering deployed defective smart contracts by redeploying patched contracts and migrating old contracts' internal states to the new ones. SolSaviour consists of a voteDestruct mechanism and a TEE cluster. The voteDestruct mechanism allows contract stake holders to decide whether to destroy the defective contract and withdraw inside assets. The TEE cluster is responsible for asset escrow, redeployment of patched contracts, and state migration. Our experiment results show that SolSaviour can successfully repair vulnerabilities, reduce asset losses, and recover all defective contracts. To the best of our knowledge, we are the first to propose a defending mechanism for repairing and recovering deployed defective smart contracts.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Network security**; **Software and application security**.

KEYWORDS

smart contract, trusted execution environment (TEE), blockchain, defence mechanism

*Corresponding author.

ACSAC '21, December 06–10, 2021, Austin, Texas
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Zecheng Li, Yu Zhou, Songtao Guo, and Bin Xiao. 2018. SolSaviour: A Defending Framework for Deployed Defective Smart Contracts. In *ACSAC '21: Annual Computer Security Applications Conference, December 06–10, 2021, Austin, Texas.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The smart contract technology that derived from blockchain systems can be used to implement almost arbitrary business logic. It can be used to create distributed autonomous organizations (DAO), revolutionize the work logic of many areas, and create financial applications such as payments and insurance. Due to the transparent nature and inherent value concentration, smart contracts are now extremely attractive targets for attacks. Vulnerable code or business logic design in smart contracts can be exploited by attackers and have serious security and property consequences. In addition, due to the immutable nature of deployed smart contracts, redeploying a new smart contract without vulnerabilities is often the only solution.

In recent years, the blockchain community has experienced several incidents resulting from smart contract errors. The first was the infamous the DAO hack [8], in which attackers exploited the reentrancy vulnerability in the DAO contract to withdraw approximately 3.6 million ethers. During this attack, honest contract developers and participants were incapable of stopping it. The only thing they could do was to withdraw ethers to a secure account as fast as possible. The Ethereum community eventually decided to reduce the impact of this attack through a hard fork. Another infamous vulnerable contract was the Parity Multisig Wallet, which was hacked twice. In the first time, attackers exploited the vulnerability of delegatecall in fallback function to change the ownership of wallet contract and stole 153,037 ethers [6]. In the second time, attackers managed to destroy the Parity wallet library contract. 587 contracts that rely on it were blocked and 513,774.16 ethers were locked [1]. Furthermore, with the development of decentralized applications (DAPP) and the increased popularity of decentralized finance (DeFi) applications such as UniSwap and flash loan, new types of attacks are occurring. For example, Fei has experienced malicious trading attacks caused by price manipulation [28].

These attacks have spurred community work on detecting smart contract vulnerabilities before deployment. Many software testing techniques are utilized, including symbolic execution [26], formal verification [3], static analysis[7, 15], dynamic analysis [36], and fuzzing testing [30]. However, these detection methods still have certain limitations, such as covering fixed types of vulnerabilities, not solving the emergence of unknown vulnerabilities, limited detection efficiency (i.e., sensitive to some vulnerabilities, but insensitive to others), and possible false negative cases. There are also some work on repairing smart contracts, including EVMPatch [33], SCRepair [38] and sGUARD [16]. We point out that for high-net-worth smart contracts, pre-deployment detection methods are not fully effective, and the possibility of post-deployment vulnerabilities still exists. Though detecting and fixing contract bugs has been extensively studied, how to repair and recover deployed defective contracts remains an unsolved problem.

For this reason, a mechanism that can repair and recover deployed defective smart contracts is required. Several methods to upgrade deployed smart contracts are developed. Proxy pattern is the most promising one. In proxy pattern, a smart contract is separated into data contract and logic contract. The data contract can use delegatecall opcode to invoke the logic contract. The delegatecall executes code of logic contract in the context of data contract. In this way, once a bug is found in a logic contract, it can be upgraded without affecting the data contract. However, proxy pattern cannot save assets inside defective contracts. We emphasized that saving funds inside defective contracts is an unsolved problem. For this reason, we aim to find a way that not only can repair vulnerabilities in defective smart contracts, but also can recover assets according to the stake distribution. Several challenges arise in the way of designing such a mechanism.

- The current method of destroying contracts by setting a trusted owner address is a trust risk. Having a decentralized way to destroy deployed smart contracts is critical to the trust of multi-user contracts, which is challenging.
- Redeploying a patched smart contract cannot inherit the defective contract's internal state. It is challenging to migrate state from old buggy contracts to new patched ones in a secure and trusted way.
- Another key challenge is how to build a TEE cluster, which acts on a unique blockchain address and can alleviate the availability problem of a single TEE node. How to reach the consensus on a unique key in the TEE cluster is also challenging.

In this paper, we propose SolSaviour, a framework that can protect deployed smart contracts against unknown bugs. To the best of our knowledge, SolSaviour is the first work that can repair and recover deployed defective smart contracts. The first important point is that SolSaviour decentralizes the control on smart contracts to multiple parties. Another key to this achievement is a secure and principled combination of blockchain and trusted hardware. In this case, clients can invoke TEE to conduct a series of pre-defined operations. This model is securer and more efficient than the traditional model of sending multiple transactions in succession to invoke a smart contract.

We propose voteDestruct mechanism to enable the decentralized smart contract control. Contract participants (i.e., stake holders) can vote on the future of the contract. They can lock the contract, destroy it, or unlock the locked contract and continue the execution. The weight of their votes depends on the number of ethers (i.e., stake) they deposited. The more stake a stake holder has, the weightier its vote.

We build a TEE cluster to take charge of smart contract exit operations and temporary asset escrow when destroying a defective smart contract. Once the patched smart contract is transferred to TEE cluster, the TEE cluster deploys it and conducts the state migration to transfer all internal assets to the newly-deployed contract. The cluster architecture can alleviate potential TEE failures, such as side channel attacks and unavailability. 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. The main contributions of this paper are summarized as follow:

- We propose voteDestruct mechanism, which allows contract stake holders to vote to destroy the defective smart contract.
- We build a TEE cluster based on Intel SGX to take charge of asset escrow and contract state migration. The TEE cluster can preserve trusted execution of contract patching.
- We collect smart contracts 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 present the overview, workflow, and building blocks of SolSaviour. The detailed implementation is presented in Section 4, and the effectiveness and performance of SolSaviour are evaluated in Section 6. We discuss related work in Section 7 and conclude our work in Section 8.

2 BACKGROUND

2.1 The Life Cycle of a Smart Contract

The life cycle of a smart contract typically consists of four phases: contract creation, contract freeze, contract execution, and contract finalization.

Creation: This phase includes writing and deploying a smart contract. First, participants must agree on the goals of the contract, which can be done online or offline, similar to traditional contract negotiations. After agreeing on the goals and content of the contract, the agreement must be translated into code. Then, the contract source code is compiled into bytecode and embedded in a transaction with a target address of 0 (i.e., contract creation transaction). Finally, this transaction is broadcasted to the blockchain network.

Freeze: After the smart contract creation transactions are committed to the blockchain, the miner includes them in a new block to make it persistent. From then on, the state variables and message calls of the contract will be public. People can access the relevant data directly through RPC calls provided by the underlying blockchain or blockchain browser websites.

Execution: Contracts are stored on miners that maintain the blockchain system. Each miner node stores the corresponding bytecode and state variables according to the address of the contract. In exchange, the initiator of a contract creation transaction pays a transaction fee to the miner. The execution of the contract is carried out through a message call. Smart contracts are called by initiating the corresponding arguments to the functions that can be called. Contracts can also call each other. Different message calls have different effects, some resulting in a transaction (internal transaction) and some changing the state variables inside the contract.

Finalization: After a smart contract is executed, the transaction containing the message call and the new state information is stored in the blockchain and persisted with the confirmation of the new blocks. Since then, we can consider one message call of the contract to be over. In addition, the user can also call the `selfdestruct` function defined inside a contract to destroy it. This will result in the end of the entire contract life cycle. Once a contract is destroyed, miners delete the bytecode and variables corresponding to its address, and the remaining property inside the contract is refunded to the address of the `selfdestruct` function's argument.

2.2 Defending Methods

2.2.1 Repairing Technique. Currently, almost all contract repairing techniques focus on repairing smart contracts before deployment. In this case, they are no different from contract vulnerability detection techniques. Namely, they cannot repair vulnerabilities that are not detected. They are even less capable of fixing bugs in deployed smart contracts.

2.2.2 Recovering Technique. Currently, the only available recovering technique is proxy pattern, where smart contracts are structured as proxy contract and data contract. Message calls are gone through proxy contract that will be redirected to the latest deployed contract logic. Once a bug is exposed, proxy pattern supports upgrade contracts. A new version of smart contract is deployed and the proxy is updated to reference the new contract address. However, this method suffers from the requirement of trusted contract owner. That is, the contract developer will set its address as contract owner in the contract creation step, which results in the fact that contract users need to trust the contract creator, which is not applicable to multi-user contract scenarios.

2.3 Defective Contract

Defective smart contracts can be divided into two categories: exploitable smart contracts and smart contacts may have unexpected internal state. For the first type, either there are some problems with the contract implementation that create bugs (e.g., reentrancy vulnerability), or an attacker can exploit the contract internal logic to launch attacks (e.g., frontrunning attack). By exploiting these bugs, attackers can gain benefits that do not belong to them. For the second type, these bugs may cause a smart contract to an unexpected state, even locked state. For example, a jackpot may never succeed because of a strict equal operation [9].

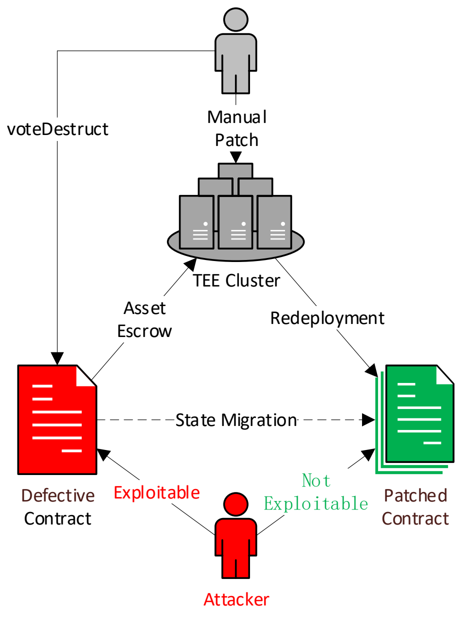

Figure 1: The overview of SolSaviour framework. Honest users can safely exit from a defective contract through voteDestruct mechanism, and can invoke TEE cluster to hold assets, redeploy a patched contract, and conduct state migration to continue contract execution.

2.4 Internal State

The concept of contract internal state includes the values of contract variables and the stake distribution of assets inside the contract. When performing contract migrations, upgrades, as well as SolSaviour-enabled contract repairing and recovering, the consistency of contract internal state should be maintained. Ensuring the consistency of contract states requires us to restore the states of defective contracts and migrate them to patched contracts.

Recovering the value of variables inside a smart contract is easy. For values of public variables, we can get them via the getter function. For private variables, we can use the blockchain history data to determine their values. Once the values of defective contract's variables have been recovered, we can write them to the patched contract.

The method of recovering stake distribution is similar to that of recovering variables' values. 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 contracts' addresses to patched contracts' addresses, in accordance with the stake distribution of the defective contracts. This is one of the main problems addressed by SolSaviour.

3 SOLSAVIOUR

3.1 What is SolSaviour

The overview of SolSaviour is depicted in Fig. 1. SolSaviour consists of two core parts: voteDestruct and TEE cluster. The voteDestruct

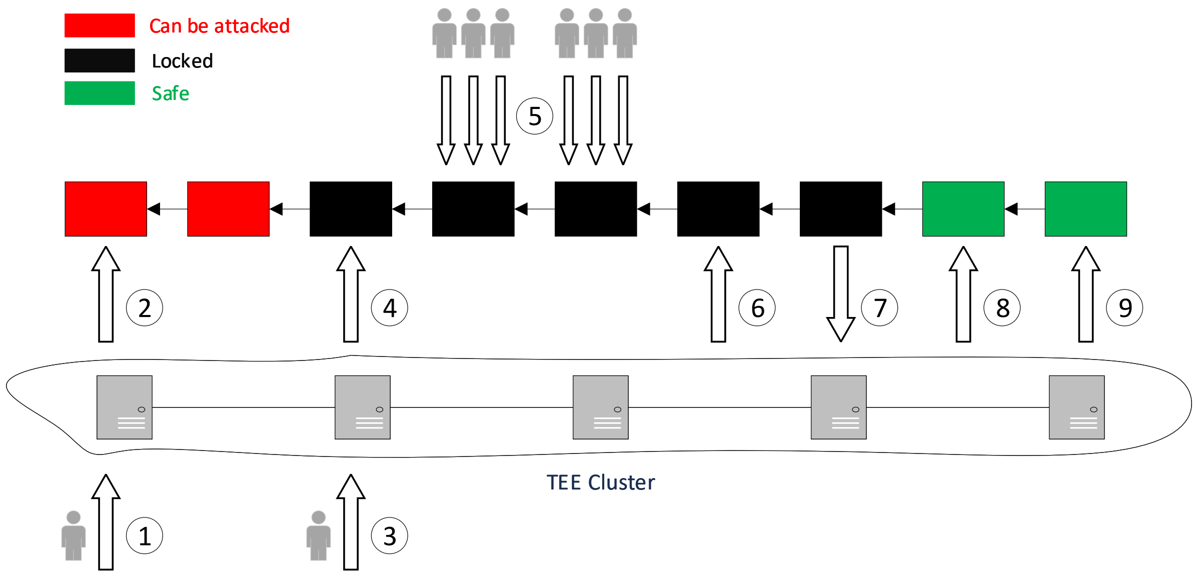

Figure 2: The workflow of SolSaviour framework. The life cycle of a defective contract ends at the rightmost black block.

mechanism is embedded in smart contracts before deployment. It ensures that a smart contract can be destroyed in the voting manner. Once a deployed contract is exposed to some vulnerabilities, the contract stake holders can invoke the voteDestrcut mechanism to destroy it.

TEE cluster provides three functionalities: asset escrow, redeployment (of patched contract), and state migration. For asset escrow, it can hold assets for a destroyed contract temporarily. For redeployment, the TEE cluster can receive patched smart contract from stake holders and deploy it. Finally, the TEE cluster can migrate defective smart contract's state (i.e., assets according to stake distribution) to the newly-deployed patched contract.

The TEE cluster can be bootstrapped by contract stake holders. For example, a group of people consider running a high-net-worth smart contract. Concerned about potential security risks arising from the contract, they could write the smart contract following the voteDestruct pattern and initialize a TEE cluster to protect it. In this way, they enable repairing and recovering capabilities for their deployed smart contract. Even if a bug is exposed in that smart contract, they can fix it without worrying about asset loss. In addition, SolSaviour can also be used as a public service, where a service provider deploys a TEE cluster to provide a mechanism for contract repairing and recovering. The deployment effort of TEE cluster is affordable, all it needs is to adapt the enclave implementation to the data structure of the target running blockchain, and to ensure that the blockchain address is correctly generated in the TEE cluster and that the generated transactions can be received by the target blockchain.

3.2 Workflow

During the contract execution (i.e., blockchain growth), an unknown bug may be disclosed. Then, contract stake holders can check whether their smart contracts are vulnerable to this new bug. If so, they can invoke the TEE cluster to to lock their smart contracts and prevent them from further interaction. After that, people who have assets inside the smart contract can decide a recovering scheme and vote whether to execute it via a cumulative voting way, where people's stake means how much they have in the contract. Since then, contract stake holders can launch a patched contract as well as the recover policy into the TEE cluster. The TEE cluster can destroy the defective smart contract and execute the recover policy (e.g., return all assets back to their owners). For a naive safe exit, TEE cluster can generate refund transactions according to the stake distribution and broadcast them to the blockchain. All locked assets can be returned to their stake holders. For contract recovering, TEE cluster can deploy a patched contract onto the blockchain and migrate all assets as well as the stake distribution to it. Since then, stake holders can continue to execute the contract without the vulnerability. The detailed workflow is summarized below as shown in Fig. 2:

① Users can initialize a deployment instruction and send its voteDestruct-enabled contract to the TEE cluster.
② The TEE cluster generates a contract creation transaction with the voteDestruct-enabled contract and broadcast it to the blockchain.
③ Once a contract bug is discovered, contract stake holders invoke the TEE cluster to turn the contract state to 'Locked', namely no further operations except vote can be conducted in the contract. In addition, if the contract stake holders want to deploy a patched contract, they should submit it in this step.
④ The TEE cluster turns the smart contract to be 'Locked'.
⑤ The contract stake holders vote to decide whether to destroy the smart contract.

⑥ The TEE cluster invokes the destroy function to execute the `selfdestruct` opcode and all ethers are transferred to an address held by the TEE cluster.
⑦ The TEE cluster receives the assets from the destroyed defective contract.
⑧ A naive approach is to distribute these assets back to their stake holders. Another way is to redeploy a patched contract provided by contract stake holders in Step ③.
⑨ The TEE cluster migrates the internal state of the defective contract to the newly-deployed patched contract by injecting the stake distribution in the patched contract.

3.3 Building Blocks

3.3.1 voteDestruct Mechanism. As an autonomous community, a smart contract stake holders can reach a consensus on whether to destroy the smart contract among themselves. Currently, a typical way to refund all assets inside a smart contract is to invoke a pre-defined destroy function. Then, all smart contract assets can be transferred to a specified address, which usually belongs to contract's owner. However, this setting cannot support multi-user smart contracts such as the DAO contract, whose assets belongs to different entities. Facing this problem, we propose voteDestruct mechanism to allow smart contract stake holders to vote whether to destroy the smart contract and withdraw all inside funds.

The voteDestruct is processed in three steps. The contract first forms the stake distribution during its execution. Then, once a vulnerability is exposed, stake holders can invoke the contract via TEE cluster to vote whether to destroy the contract. After completing the voting, the contract stake holders can invoke the TEE cluster to destroy the defective contract.

Stake Distribution: Each depositing transaction will be recorded. Specifically, the address of depositor and amount of deposited ethers will be recorded.

Voting: The voting function can be invoked by any stake holder. During the voting process, the smart contract is in the locked state so that no external users can deposit or withdraw ethers. This ensures that the stake distribution (i.e., voting power) remains the same during the whole voting process. After completing the voting, contract calculates the votes on supporting and opposing based on the stake, and then decides whether to execute the destroy function.

Destroying: Only when the cumulative voting on supporting destroying exceeds 2/3 stake, the contract is capable of being destroyed. Contract stake holders can instruct the TEE cluster to invoke the destroy function once the voting process completes.

3.3.2 TEE Cluster. We establish a TEE cluster to mitigate the availability problem of TEE nodes. We present the details of TEE cluster in Figure. 3. 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 distinguish 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 broadcasting transactions onto the blockchain.

Bootstrapping: In the bootstrapping step, all TEE nodes inside the same cluster need to reach an agreement on a key that used to generate the blockchain address. Each enclave is assigned a

```
 1: procedure BOOTSTRAPPING($\sigma_i, i \in 0, n-1$)
 2:     load $\sigma_i$                                ▷ initialize node $i$
 3:     generate {sgx_quote$_i$, $K_i$}
 4:     broadcast {sgx_quote$_i$, $K_i$}
 5:     verify sgx_quote$_j$
 6:     generate {$K$,addr}
 7: end procedure
 8: procedure TRANSACTION GENERATION
 9:     on receiving contract $c$                      ▷ deploy contract
10:     tx.payload($c$), goto line 15
11:     on receiving address $addr$           ▷ lock defective contract
12:     tx.payload(addr, func vote_initial), goto line 15
13:     on receiving address $addr$                ▷ destroy contract
14:     tx.payload(addr, func destroy), goto line 15
15:     Sign$_K$tx
16:     upload tx
17:     broadcast(++nonce)
18: end procedure
19: procedure STATE RECOVERY
20:     mapping stake_dist(addr→stake_amount)
21:     stake_dist ← addr.st_map_getter()
22: end procedure
23: procedure SAFE EXIT
24:     for addr ∈ stake_dist do
25:         addr.transfer(stake_amount)
26:     end for
27: end procedure
28: procedure PATCHED CONTRACT DEPLOYMENT
29:     on receiving $C_p$
30:     go to line 9          ▷ Redeployment of patched contrct
31: end procedure
32: procedure STATE MIGRATION(stake_dist)
33:     tx.payload(addr, stake_dist), go to line 15
34: end procedure
```

Figure 3: The working logic of TEE cluster. The TEE cluster receives instructions from contract stake holders and generates transactions to call smart contract functions.

number before the deployment. During the bootstrap process, p is the module of computation and g is the generator of the cyclic group.

Step 1. Each TEE node U_i generates a random number $x_i = g^{x_i} \bmod p$ and broadcast it to the other TEE nodes.

Step 2. Each node broadcasts $K_i = (\frac{k_{i+1}}{k_{i-1}})^{x_i} \bmod p$

Step 3. Each node calculates:

$$K_i = K_{i-1}^{nx_i} \cdot K_i^{n-1} \cdot K_{i+1}^{n-2} \cdots K_{i-1} \bmod p$$

If all parties follow the above steps, they can reach an agreement on the key:

$$K = g^{x_1x_2} \cdot g^{x_2x_3} \cdot g^{x_3x_4} \cdots g^{x_nx_1}$$

Then, following the standard way, each node can generate the same blockchain address. Once the TEE cluster has been set up, people can instruct it to deploy a contract as shown in line 9. Then, the contract can run safely under the protection of SolSaviour until the end, or if there is a bug.

Asset Escrow: Once a bug is found in a SolSaviour-protected smart contract, stake holders can instruct the TEE cluster to lock the contract as shown in line 11 and then vote whether to destroy

```
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));}}
```

Figure 4: A Sample of voteDestruct Contract.

it. After completing the voting process, contract stake holders can instruct the TEE cluster to invoke the destroy function as shown in line 14. As soon as the amount of stake in favour of destruction exceeds a specified threshold, the contract is successfully destroyed and all assets are transferred to the address held by the TEE cluster. During the contract recovering process, TEE cluster hold the asset of the destroyed contract until deploying its patched version.

State Migration: During the locked state, the TEE cluster obtains contract's stake distribution via the getter function and saves it. Based on this internal state, the contract owner can call the TEE cluster to perform a safe exit or state migration. The TEE cluster also takes charge of migrating the internal state from the buggy contract to the patched new one. In this process, the TEE cluster ensures that the states are consistent.

4 IMPLEMENTATION

4.1 Repairing Smart Contract

4.1.1 Patch Generation. In SolSaviour, patches for defective smart contracts are provided by the contract stake holders. This is because the main purpose of SolSaviour is to provide a framework for repairing and recovering deployed defective smart contracts, rather than providing a system that can automatically generate patches. Smart contract patches can be generated manually or using existing tools such as sGuard [16] and SCRepair [38]. Once a patched contract is prepared, contract stake holders can pass it to the TEE cluster for redeployment.

For known bugs such as reentrancy and integer overflow, stake holders can leverage existing tools to generate patched contracts. For unknown bugs, patched contracts should be developed by experts, which are trusted by the stake holders. After patching, the contract should be tested thoroughly before deploying by re-executing all previous related transactions. This can test the whether the patched contract functions well and has fixed all related bugs.

4.1.2 voteDestruct. Currently, we can use `selfdestruct` primitive to destroy deployed smart contracts and refund all inside assets. When writing a smart contract, there are usually restrictions set to limit that only privileged owners can invoke the `selfdestruct` primitive, otherwise this contract can be destroyed by anyone. However, this method requires contract stake holders to trust the privileged owner since he/she is able to withdraw all inside assets. In this case, we introduce voteDestruct mechanism, which allows contract stake holders to destroy a deployed smart contract in a decentralized way. Fig. 4 shows the sample of voteDestruct mechanism. We emphasize that its implementation does not require new EVM instructions. It is constructed based on pure Solidity language. Moreover, the voteDestruct mechanism can be implemented in different versions 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 stake holders `st_holder` and the amount of their deposited ethers as `st_amount`. Once a stake holder found that this smart contract is vulnerable to some newly discovered bugs, it can invoke the `vote_initial` function via the TEE cluster. The smart contract then starts the cumulative vote in which each stake holder chooses whether to destroy this contract and return all funds.

4.1.3 Safe Exit. Through voteDestruct mechanism, SolSaviour enables contract stake holders to safely exit from a defective smart contract. SolSaviour utilizes a cumulative voting algorithm, which calculates votes based on stake. Cumulative voting is the procedure followed by electing whether to destroy the smart contract. Typically, each stake holder should choose to support or oppose the contract destruction. Once the vote completes, different choice will be counted according to the amount of stake.

If the contract stake holders vote not to destroy the contract to resolve the defect, they can instruct TEE cluster to invoke `vote_halt` function to unlock contract. Otherwise, the contract executes its `selfdestruct` operations and send all inside funds to an account controlled by the TEE cluster. In this way, contract stake holders can safely exit from a contract that is exposed to some critical bugs or under attacks. Compared with traditional `selfdestruct` operations, safe exit not only saves all preserved funds, but also avoids the requirement to trust a privileged owner.

4.1.4 Redeploy a Patched Contract. Though safe exit can save almost all buggy contract stake holders from losing assets. There exist cases that stake holders need to fix vulnerabilities and continue the contract execution. SolSaviour therefore provides an alternative way, namely redeploying a patched contract and migrating the internal state from the defective contract to the patched one.

First, contract stake holders invoke the TEE cluster to lock the defective contract and prepare a patch. Once the patched smart contract is generated and tested, it can be sent to TEE cluster for deployment. Then, stake holders can instruct the TEE cluster to invoke the voteDestruct mechanism to destroy the defective contract and withdraw all assets to TEE cluster safely. After safe exit, stake holders can invoke TEE cluster to deploy the patched contract.

During redeployment, the TEE cluster takes charge of injecting the initial state of the patched contract and generating a contract creation transaction. To be consistent with the destroyed defective contract, SolSaviour makes sure that the newly-deployed patched contract shares the same internal state with the destroyed one. The TEE cluster injects a list of stake holder addresses and the amount of their stakes to the patched contract, which indicates the amount of ethers 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 stake holders, the internal state of the redeployed contract remains the same as the previous defective contract, but SolSaviour has already fixed the vulnerabilities.

4.2 Recovering Smart Contracts

In SolSaviour, the recovering process proceeds in three phases. During the *setup phase*, the contract is destroyed according to the voting results. Then, assets of destroyed defective smart contracts are temporarily held by the TEE cluster in the *escrow phase.* After that, the protocol proceeds to the *recovering phase* to deploy the patched contract and migrate the previous contract's internal state to the new one.

Setup Phase. In the setup phase, once a contract stake holder notices a potential vulnerability in the contract, it can broadcast the vulnerability to attract other stake holders' attention. Then, all contract stake holders can vote whether 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 stake holders can discuss the exposed vulnerability and develop corresponding patches.

If most stake holders think that this vulnerability is a false positive case, they can vote to unlock the smart contract and continue to work with it. The voting threshold for unlocking a locked smart contract is the same as the threshold for locking one. Afterwards, the contract can continue to operate normally. If the discussion thinks the vulnerability may lead to serious consequences, stake holders need to develop a valid patch and test it. Then, they could vote whether 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 by TEE cluster. All internal assets are transferred to the TEE cluster for temporary escrow.

Escrow Phase. When entering the escrow phase, the old vulnerable smart contract has been destroyed. All assets inside the defective contract are transferred to an account controlled by the TEE cluster. Before redeploying a patched smart contract, TEE cluster extracts the internal state of the old defective smart contract, namely values of state variables and stake distribution. By analyzing the internal state of defective contract, the TEE cluster is able to migrate the state of defective contract to the patched one.

Recovering Phase. In the recovering phase, the TEE cluster first modifies the patched smart contracts provided by the contract stake holders. The purpose of this modification is to migrate the internal state from the old, vulnerable contract to the new, patched smart contract. TEE cluster ensures that variables in the patched contract are the same with before by initializing these variables. Then, TEE cluster deploys the patched contract and 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. Moreover, only the corresponding accounts can extract these assets from the newly deployed patched smart contract.

5 SECURITY ANALYSIS

5.1 Threat Model

We assume that parties who do not trust each other use the blockchain to execute smart contracts and mine new blocks. We assume most machines are equipped with TEE, which is based on the observation that most computers have SGX-capable Intel CPU. We assume that the TEE (i.e., programs inside enclaves) on a machine is trusted, but some TEE nodes may suffer from integrity and confidentiality problem. They may be destroyed by other parties or external attackers. All parties are rational and potentially malicious. When there are benefits, they may try to steal funds that belong to others and force the TEE to modify the stake distribution. All parties are connected via the network, and they can discard and replay the information. Malicious hosts can delay or prevent others from accessing the blockchain for an unlimited time, but we assume that this will not happen indefinitely. We also assume that an adversary $\mathcal{A}$ can corrupt up to t of n hosts in the TEE cluster. We consider the adversary can cause corrupted hosts to deviate from the specified protocol, namely drop or delay messages between enclaves. Our adversary $\mathcal{A}$ is static, namely chooses the corrupted hosts at the beginning of the protocol.

5.2 Trusted Execution as a Root-of-Trust

In SolSaviour, participants can monitor the blockchain to detect deviations from the protocol and react appropriately. Here, we propose a new trust mechanism with a TEE cluster as the root of trust. The TEE cluster is independent of the blockchain and can ensure the faithful execution of SolSaviour. TEEs are encrypted and integrity-protected memory regions that are isolated from the rest of the software stack by the CPU hardware, including higher-privilege system software. By using the TEE cluster as an independent root of trust, SolSaviour can ensure secure exit, redeployment, and effective state migration of defective smart contracts. In addition, the cluster architecture improves the overall fault tolerance of the system. Single point failures will not affect the overall availability of SolSaviour.

5.3 Threat Analysis

voteDestruct. We first analyze the case where there are some malicious stake holders. As malicious stake holders are profit oriented, what they want is to acquire the assets of honest stake holders. We prove the security of voteDestruct mechanism by showing that honest stake holders 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 statuses, active but potentially defective, locked, and destroyed. In locked status, a contract is protected by blockchain miners that reject all function calls except those initiated from the TEE cluster. In this case, malicious stake holders cannot steal assets. In destroyed status, assets in a contract are held in custody by the TEE cluster, so that malicious stake holders cannot profit either. The only chance for malicious stake holders to profit is during the active but potentially defective status. In SolSaviour, the threshold of required stake amount to lock a contract is 1/3. Only when the amount of stake held by malicious stake holders exceeds 2/3, they can prevent the contract from entering the locked status.

During the active but potentially defective status, when a hidden vulnerability is exposed, malicious stake holders can prevent the contract from entering the locked status and exploit the vulnerability. However, due to the unknown nature of the vulnerability, it may simply causes the contract to an inexecutable state, which would also be unprofitable for malicious stake holders. Therefore, the only feasible way for attackers to exploit a contract is to inject an exploitable vulnerability during the initial contract deployment or redeployment of patched contract. However, as the deployed contract needs to pass the checks of all stake holders, honest stake holders can reject a potentially vulnerable contract. Even if an attacker possesses an unknown vulnerability and successfully deploys a malicious contract with it, SolSaviour can increase the attack cost and reduce the losses of honest stake holders by lowering the threshold for entering the locked state. For example, if the threshold is lowered to 1/10, the attacker must have 9/10 of the stake to guarantee that this contract will not enter the locked stake, which will also reduce the losses of honest stake holders.

TEE cluster. We prove that the bootstrapping process of TEE cluster is secure 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 an index i, 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. Then, we prove that TEE cluster can reach an agreement on an identical key as long as more than t out of n nodes are honest.

We prove that TEE cluster can safely perform a state migration after destroying the defective contract. First, assets are held by TEE cluster after safe exit. As assumed before, attackers can only control no more than t of n TEE nodes and cannot derive the account private key. Thus, attackers cannot perform asset transfer directly. Since the execution logic of enclaves is fixed once encapsulated, there is no way for attackers to tamper with the internal logic of the TEE cluster. In addition, the history of defective smart contracts is stored on the blockchain and publicly available. TEE cluster can crawl the contract history to determine the values of defective contract's internal variables. The stake distribution can also be calculated by querying the transaction history of the contract and thus acquiring the internal state of the defective contract. In this way, TEE cluster can ensure safe and successful migration of contract internal state from the old vulnerable smart contract to the new patched one.

5.4 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. In addition, SolSaviour focuses primarily on the protection and migration of contract assets, namely the blockchain native currency. However, as DeFi technology has evolved, a large number of projects have been launched. Many assets in smart contracts are in the form of tokens. SolSaviour does not yet have the ability to protect these DeFi smart contracts. Enabling repairing and recovering ability in DeFi contracts is the focus of our future work.

As TEE is an evolving technology that may have unknown vulnerabilities, the potential risk of using a TEE cluster is that a newly discovered TEE vulnerability could compromise the security of the entire system and the assets within the contract protected using that TEE cluster.

6 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.

6.1 Dataset Collection

To accurately evaluate the effectiveness and performance of SolSaviour, we collect contracts that that went through attacks. We also crawl the transaction history of these victim contracts to extract attack patterns.

Our collected contracts are the DAO [8], PoWH Coin [2], 1st [6] and 2nd [1] Parity Multisig Wallet, King of Ether [34], Bancor Exchange [4], GovernMental [37], and Rubixi [22]. We list these contracts in Table 1, accompanying with contract address, vulnerability type, architecture (single or multiple), and damage caused as well as accurate loss (if so).

We also prepare the 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 compatible. We manually patch the collected defective contracts on the source code level by replacing the vulnerable operations. The voteDestruct-enabled contracts and

Table 1: The List of Contracts that Have Been Attacked.

Contract	Address	Vulnerability Type	Architecture	Damage
The DAO	0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413	Reentrancy	Multiple	3.6M ETH Loss ($150M)
PoWH Coin	0xA7CA36F7273D4d38fc2aEC5A454C497F86728a7A	Integer Underflow	Single	866 ETH Loss ($800k)
1st Parity Multisig	0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4	Delegatecall	Multiple	153,037 ETH Loss ($31M)
2nd Parity Multisig	0x863DF6BFa4469f3ead0bE8f9F2AAE51c91A907b4	Denial of Service	Multiple	513,774.16 ETH Locked ($300M)
King of Ether	0x2464d1d97f8D0180CFaD67BdB19bc30ccA69DdA0	Unchecked Return Values	Single	Ownership Loss
Bancor Exchange	0x1F573D6Fb3F13d689FF844B4cE37794d79a7FF1C	Front Running	Multiple	Economic Earns ($150)
GovernMental	0xF45717552f12Ef7cb65e95476F217Ea008167Ae3	Timestamp Dependence	Single	DoS
Rubixi	0xe82719202e5965Cf5D9B6673B7503a3b92DE20be	Bad Constructor	Single	Ownership Loss

patched contracts are then compiled with the exact same Solidity compiler version as used in the original contract.

We apply SolSaviour to these generated comparative smart contracts and validate outcomes. This allows us to verify whether voteDestruct mechanism works and patching approaches abort the attack transactions. Apart from the valid attack transactions, the execution traces of the re-executed transactions match those of the original transactions, confirming that our voteDestruct mechanism and patches do not break functionalities of the original contract.

6.2 Effectiveness

The effectiveness of SolSaviour is evaluated in two aspects: qualitative and quantitative.

For qualitative part, we check whether we can leverage SolSaviour to safe exit from all collected defective contracts, refund locked assets back to stake holders, 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. This is conducted through observing and analyzing rejected transactions, which arise from one of the following reasons: (1) a malicious transaction is successfully prevented, (2) the reported vulnerability was a false positive and should not have been patched, or (3) the contract's functionality is unintentionally changed.

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. For the original one, we also record the loss when taking traditional defence measures and doing nothing. We checked to what extent can SolSaviour save loss when facing different vulnerabilities.

6.2.1 *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 stake holders and each has deposited 100 ethers. Then, a random stake holder initializes the `vote_initial` and provides a patched contract to the TEE cluster. In our experiments, we omit the security assumption of potential malicious stake holders 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.

Table 2: The Capability of SolSaviour (Qualitative).

Contract	State Migration	Functionality	Defence
The DAO	Yes	Yes	Yes
PoWH Coin	Yes	Yes	Yes
1st Parity Multisig	Yes	Yes	Yes
2nd Parity Multisig	Yes	Yes	Yes
King of Ether	Yes	Yes	Yes
Bancor Exchange	Yes	Yes	Yes
GovernMental	Yes	Yes	Yes
Rubixi	Yes	Yes	Yes

Table 3: The Comparison of Loss Affected by Actual Attack, Traditional Defence Methods and SolSaviour (Quantitative).

Contract	Actual	Traditional	SolSaviour
The DAO	100	(48.6, 48.6%)	(6.5, 6.5%)
PoWH Coin	866	(866, 100%)	(0, 0%)
1st Parity Multisig	100	(100, 100%)	(0, 0%)
2nd Parity Multisig	100	(100, 100%)	(0, 0%)
King of Ether	Lose Onwership	No Mitigation	Fix
Bancor Exchange	100	(69.6, 69.6%)	(0, 0%)
GovernMental	Lose Ownership	No Mitigation	Fix
Rubixi	Lose Ownership	No Mitigation	Fix

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 buggy contract and patched contract are identical. Not only the stake distribution, but also the ownership. We check this by letting each stake holder withdraw their previously-deposited ethers. We found that stake holders 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. We list the results in Table. 2.

6.2.2 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 deployed defective contract contains 100 ethers. Then, we start to attack and recover it at the same time. Attackers can arbitrarily withdraw ethers until honest stake holders lock the contract. Then, we follow the safe exit way to refund all locked ethers to stake holders 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.

6.3 Performance

6.3.1 Contract Size Increase. On Ethereum, deploying smart contracts requires costs proportionally to the size of the deployed contract. More specifically, Ethereum charges 200 gas per byte to store the contract code on the blockchain. In SolSaviour, as we implement the voteDestruct mechanism in contracts, the extra overhead is introduced in the storage variables. The manually-generated patch may also increases the code of contracts, which depends on the specific location of the vulnerability.

However, since patches are generated manually, it is not possible 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 voteDestruct framework that makes defective contracts have the ability to iterative upgrades under SolSaviour. In this case, we compare the size of compiled contract in terms of defective, voteDestruct but not patched, and patched version. Results are listed in Table. 4. 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 Milti-Sig contract, we note that injecting voteDestruct mechanism naturally resolves the vulnerability so that the patched contract and voteDestruct-enabled contract have the same size.

6.3.2 Gas Consumption. In this section, we evaluate the additional gas cost incurred by SolSaviour, which arises from two aspects: the voteDestruct mechanism and the redeployment of the patched contract.

For voteDestruct mechanism, 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. We evaluated the gas consumption by deploying the prepared voteDestruct-enabled contract.

In the deployment of a patched contract, 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 already shown in Table. 4, the overhead introduced by the patch is usually small.

We also test the gas consumption to deploy the original version of collected defective contracts for comparison. Results are summarized in Table. 5.

6.3.3 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 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

Table 4: Code Size Increase in SolSaviour.

Contract	Original (B)	voteDestruct (B)	Patched (B)
The DAO	47,276	49,456	49,586
PoWH Coin	11,540	13,816	14,536
1^{st} Parity Multisig	2,888	5,740	5,740
2^{nd} Parity Multisig	2,888	5,740	5,740
King of Ether	16,812	18,934	19,648
Bancor Exchange	16,464	18,988	19,124
GovernMental	7,476	9,654	10,436
Rubixi	8,150	10,132	10,864

Table 5: Gas Consumption of SolSaviour.

Contract	Original	voteDestruct	Redeployment
The DAO	5,251,220	5,485,988	5,486,012
PoWH Coin	1,289,254	1,583,188	1,588,982
1^{st} Parity Multisig	317,780	625,375	625,375
2^{nd} Parity Multisig	317,780	625,375	625,375
King of Ether	1,858,337	2,093,078	2,174,754
Bancor Exchange	1,816,565	2,105,104	2,112,419
GovernMental	888,108	1,122,930	1,128,544
Rubixi	970,045	1,183,896	1,191,574

Table 6: The Overhead of TEE Cluster. 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%)

during this experiment. We summarize the results in Table. 6. 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. In this case, if all SGX-capable miners launch an enclave instance, we can reduce the overhead of TEE cluster to the minimum.

7 RELATED WORK

In this section, we first discuss the work on developing vulnerability detection tools for smart contracts. Then, we introduce some efforts on defending smart contracts and generating contract patches. Finally, we present the work that combines smart contracts with TEE.

7.1 Smart Contract Vulnerability Detection

The infamous reentrancy bug in "TheDAO" contract [8] has spurred community to work on detecting smart contract vulnerabilities. Luu et al. first proposed Oyente [26] based on symbolic execution, which automates the reentrancy bug detection. Then, a lot of symbolic execution tools are proposed such as Osiris [35], teEther [23], Maian [31], and Manticore [29]. Furthermore, Frank et al. proposed EthBMC [18], a bounded model checker based on symbolic execution. Kalra et al. presented Zeus [21], which leverages both abstract interpretation and symbolic model checking. Chen et al. found and defined 20 types of contract defects [9] and proposed the corresponding defect detection tools to find them on the bytecode level [10].

There are also some work on developing smart contract static analysis tools. Feist et al. proposed Slither [15] to analyze the contract on source code level, and Tsankov et al. presented Security [36] to analyze the contract on bytecode level. Furthermore, Brent et al. proposed Ethainter [7], which conducts the information flow analysis and data sanitization to reveal composite vulnerabilities. There are also work on building modular dynamic analysis frameworks for protecting smart contracts. Chen et al. proposed SODA [11], which accepts user-defined vulnerability pattern. Furthermore, method like formal verification has been introduced to smart contracts [21] and the semantics of Solidity have been formalized [20].

However, these proposed detection tools are limited so that there exist the requirement to develop contract repairing and recovering techniques. For example, teEther [23] and Maian [31] cannot identify the location of integer overflow since they focus on generating exploits for smart contracts. In addition, aforementioned detection tools cannot identify unknown vulnerabilities, which can be resolved by SolSaviour.

7.2 Smart Contract Defence and Patch

Apart from work on proving the correctness or identifying certain types of vulnerabilities in smart contracts, some research was proposed to protect deployed smart contracts.

Rodler et al. proposed *Sereum* [32] to address this problem in the context of reentrancy exploits. Experiment results demonstrate that *Sereum* covers the actual transaction sequence and execution flow of a smart contract. In this case, it can accurately detect and prevent reentrancy attacks. Furthermore, Ferreira et al. proposed ÆGIS [17], which extends the contract defence to more type of attacks. Users can identify different types of attack patterns and use them to secure deployed smart contracts.

Ellul et al. proposed a runtime verification mechanism [14] to ensure that violating party provides insurance for correct behavior. Li et al. proposed Solythesis [24] to address the high overhead in runtime validation. Their experiment results show that Solythesis can work with little overhead and enforce the security of smart contracts. In addition, Grossman et al. proposed ECFChecker [19], which defines the notion of Effectively Callback Free (ECF) objects and can detect live reentrancy attacks on vulnerable contracts.

For contract patch, Yu et al. proposed SCRepair [38], which can automatically detect and repair bugs in smart contracts before deployment. Furthermore, bytecode rewriting for patching smart contracts has been explored by zhang et al. in SMARTSHIELD [41]. SMARTSHIELD implements custom bytecode analysis tools to detect vulnerabilities, which may not be as accurate as specialized analysts. But experiment results show that SMARTSHIELD can fix more than 91.5% vulnerable samrt contracts. Rodler et al. proposed EVMPatch [33] for instantly and automatically patching defective smart contracts. EVMPatch leverages a bytecode rewriting engine for smart contracts on Ethereum. However, EVMPatch can only fix limited types of vulnerabilities.

7.3 Smart Contract and TEE

Zhang et al. proposed Town Crier [39], which meets the requirement of a trusted oracle for smart contracts. Through Town Crier, information on existing websites can be transferred to smart contract in a trusted way. Matetic et al. proposed Bite [27], which is a lightweight client based on TEE. In Bite, full nodes are equipped with SGX enclaves that can serve privacy preserving requests from light clients. In this case, Bite can protect the privacy of lightweight nodes, which usually query its related transactions.

A series of work focus on offloading contract execution to TEE, which brings in benefits such as privacy-preserving and performance improvement. Bowman et al. proposed Private Data Objects (PDOs) [5], which allows mutually untrusted parties to run smart contracts over private data. In PDOs, contracts run off-chain in secure enclaves. Cheng et al. proposed Ekiden [12], which leverages the TEE to execute private smart contracts. The architecture of Ekiden separates the consensus from execution by letting TEE nodes execute smart contracts and miners maintain the consensus.

The contract content and state transitions are encrypted on chain so that contract confidentiality is preserved. Das et al. proposed FastKitten [13] to enable the execution of arbitrarily complex smart contracts on Bitcoin. In FastKitten, smart contracts are executed in enclaves and the Bitcoin only records state transitions. FastKitten can extend its work to execute smart contracts on more cryptocurrencies which are designed to only support simple transactions.

In addition, the combination of blockchain and TEE shows promise in many other areas. Zhang et al. proposed REM (Resource-Efficient Mining) [40], a blockchain mining algorithm that work on useful computation. REM notices that current Proof-of-Work algorithm is energy-consuming so that leverages the partially decentralized trust model inherent in SGX to achieve a fraction of the waste of PoW. Lind et al. leveraged TEE as trusted nodes in payment network and proposed Teechain [25], which is a layer-two network that can processes off-chain transactions asynchronously. Teechain can prevent parties from misbehaving by establishing off-chain payment channels with TEE.

8 CONCLUSION

In this paper, we propose the SolSaviour framework for the problem of repairing and recovering defective deployed smart contracts. SolSaviour enables the patching of deployed defective smart contracts through the voteDestruct mechanism and the asset escrow and state migration provided by TEE cluster. 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 participation of a trusted third party. For all collected contracts that were attacked, our experiments demonstrate that SolSaviour can effectively repair and recover all of them with affordable overhead.

ACKNOWLEDGMENTS

This paper is partially supported by HK RGC GRF PolyU 152124/19E and 15216220.

REFERENCES

[1] Anthony Akentiev. 2017. Parity Multisig Hacked. Again. https://medium.com/chain-cloud-company-blog/parity-multisig-hack-again-b46771eaa838

[2] Eric Banisadr. 2018. How $800k Evaporated from the PoWH Coin Ponzi Scheme Overnight. https://medium.com/@ebanisadr/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530

[3] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security.* 91–96.

[4] Ivan Bogatyy. 2017. Implementing Ethereum trading front-runs on the Bancor exchange in Python. https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798

[5] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. 2018. Private Data Objects: An Overview. *arXiv preprint arXiv:1807.05686* (2018).

[6] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. 2017. An In-Depth Look at the Parity Multisig Bug. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

[7] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 454–469.

[8] Vitalik Buterin. 2016. CRITICAL UPDATE Re: DAO Vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/

[9] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2020. Defining smart contract defects on Ethereum. *IEEE Transactions on Software Engineering* (2020), 1–17.

[10] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing EVM bytecode. *IEEE Transactions on Software Engineering* (2021), 1–20.

[11] Ting Chen, Rong Cao, Ting Li, Xiapu Luo, Guofei Gu, Yufei Zhang, Zhou Liao, Hang Zhu, Gang Chen, Zheyuan He, et al. 2020. SODA: A generic online detection framework for smart contracts. In *NDSS.* 1–17.

[12] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. In *2019 IEEE European Symposium on Security and Privacy.* 185–200.

[13] Poulami Das, Lisa Eckey, Tommaso Frassetto, David Gens, Kristina Hostáková, Patrick Jauernig, Sebastian Faust, and Ahmad-Reza Sadeghi. 2019. Fastkitten: Practical Smart Contracts on Bitcoin. In *28th USENIX Security Symposium.* 801–818.

[14] Joshua Ellul and Gordon J Pace. 2018. Runtime verification of Ethereum smart contracts. In *2018 14th European Dependable Computing Conference (EDCC).* IEEE, 158–163.

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

[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2021. sGUARD: Towards fixing vulnerable smart contracts automatically. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 1–15.

[17] Christof Ferreira Torres, Mathis Baden, Robert Norvill, Beltran Borja Fiz Pontiveros, Hugo Jonker, and Sjouke Mauw. 2020. Ægis: Shielding vulnerable smart contracts against attacks. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (AsiaCCS).* 584–597.

[18] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A bounded model checker for smart contracts. In *29th USENIX Security Symposium.* 2757–2774.

[19] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–28.

[20] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2020. Semantic understanding of smart contracts: executable operational semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 1695–1712.

[21] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing safety of smart contracts.. In *NDSS.* 1–12.

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

[23] Johannes Krupp and Christian Rossow. 2018. teether: Gnawing at Ethereum to automatically exploit smart contracts. In *27th USENIX Security Symposium.* 1317–1333.

[24] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 438–453.

[25] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: A secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP).* 63–79.

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

[27] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. 2019. BITE: Bitcoin lightweight client privacy using trusted execution. In *28th USENIX Security Symposium.* 783–800.

[28] Brianna Montgomery. 2021. Fei Bonding Curve Bug Post Mortem. https://medium.com/fei-protocol/fei-bonding-curve-bug-post-mortem-98d2c6f271e9

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

[30] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE).* 778–788.

[31] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC).* 653–663.

[32] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2019. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *NDSS.* 1–15.

[33] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2021. EVMPatch: timely and automated patching of Ethereum smart contracts. In *30th USENIX Security Symposium*. 1–18.

[34] KoET Team. 2016. King of Ether Throne Post-Mortem Investigation. https://www.kingoftheether.com/postmortem

[35] Christof Ferreira Torres, Julian Schütte, and Radu State. 2018. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. 664–676.

[36] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 67–82.

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

[38] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart contract repair. *ACM Transactions on Software Engineering and Methodology* 29, 4 (2020), 1–32.

[39] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 270–282.

[40] Fan Zhang, Ittay Eyal, Robert Escriva, Ari Juels, and Robbert Van Renesse. 2017. REM: Resource-Efficient Mining for Blockchains. In *26th USENIX Security Symposium*. 1427–1444.

[41] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. Smartshield: Automatic smart contract protection made easy. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 23–34.