IEEE ICBC'24 1570977520

ContractSafeguard: Practical Bug Bounty Platform for Smart Contracts with Intel SGX

No author given

***Abstract*—Bug bounties are a helpful system for identifying vulnerabilities in smart contracts. Many web3 applications use bug bounties today. However, if the bounty is not exchanged atomically with a bug report, hackers worry about non-payment or a reduction in the bounty. This worry can motivate hackers to exploit the identified bug instead of reporting it. Furthermore, other efforts related to bug bounties require modification of existing smart contracts. To solve this, we propose a practical bug bounty platform for smart contracts, ContractSafeguard, that utilizes Intel SGX to ensure the existence of bugs and atomic exchange of bounty and bug reports. Bug hunters issue transactions to smart contracts that are in the Enclave and glitch them. Test code within the Enclave generates a bug report that includes proof of the bug's existence and a method that triggers the bug. In other words, within the Enclave, our mechanism detects and proves that the smart contract under test has entered an abnormal state. The on-chain bug bounty contract then verifies the proof within the bug report and pays the bounty to the bug hunter atomically. Since we generate the proof within the Enclave, no one can generate fraudulent proofs, and the on-chain contract can trust the report. Additionally, the developer can use existing smart contracts without modification since we have a runtime of smart contracts in the Enclave. We design and implement ContractSafeguard for Ethereum smart contracts. Using the historical Ethereum attack cases, The DAO and Parity Wallet Hack, as examples, we verify its practicality in terms of whether it can use existing contracts without modification and whether the gas cost is affordable.**

***Index Terms*—Blockchain, Bug bounty, Smart contract, Intel SGX**

I. Introduction

Smart contract vulnerabilities have resulted in the theft of millions of dollars over the years.. Table I shows the attacks caused by bugs in smart contracts. Hacker attacked a Re-Entrantrancy vulnerability in The DAO's smart contract and leaked $50 million worth of ETH[1]. This attack triggered the split of Ethereum into what is now Ethereum and Ethereum Classic. Parity Wallet's contract had a vulnerability regarding an externally accessible initialization function, resulting in the leak of $180 million[2][3]. Other attacks have also occurred due to similar bugs[4][5][6].

To find vulnerabilities, the Ethereum community recommends using Bug Bounty[7]. Bug Bounty is a system that uses bounties as an incentive for third parties to find and report bugs in software. The bounty system provides a platform to connect third parties with software developers. When a third party discovers a bug, they submit a bug report to the developer via the platform. The developer reviews it and pays a bounty if the bug is indeed found. There are also Bug Bounties for blockchain smart contracts[8][9].

To use Bug Bounty to improve smart contract security, Bug Bounty must meet the following requirements.

R.1 Only the bug hunter and the smart contract developer know the details of the bug.

R.2 The bug report and bounty are exchanged atomically when the report is correct.

R.3 The Bug bounty platform can handle existing smart contracts without modification.

Bug reports must be kept secret from others to avoid exploitation, and bug reports and bounties must be exchanged atomically to prevent bounty fraud. In addition, for convenience, Bug Bounty must use existing smart contracts without modification.

Existing bug bounties do not meet R.2 above because developers verify the bug reports manually[8][9]. Developers do not pay bounties before verifying the correctness of bug reports, but hackers are wary of the risk of not being paid bounties after revealing a bug report[10]. This incentivizes hackers to obtain funding by exploiting vulnerabilities rather than the bug bounty.

Here, there are several efforts to provide bug reports and bounty payments atomically. [11] applies N-version programming to detect bugs by deploying multiple smart contracts with identical logic on-chain and comparing their outputs. When a head contract receives a transaction from a user, it forwards the transaction to several application contracts and compares the execution results. The head contract then pays rewards when it detects differences in execution results (bugs).

However, they cannot use existing smart contracts without modification. Their framework does not support contracts that use $calldata$ or $delegatecall$. We counted how many contracts deployed in the last million blocks of Ethereum, from 17,487,178 to 18,487,178, use $calldata$ or $delegatecall$. Of the 64,625,461 contracts, 26,360,624 contracts, representing 40%, were using $delegatecall$. Some standards recommend $delegatecall$ to save on fees[12][13] and to allow for flexible logic updates[14]. In addition, it is necessary to develop two or three new contracts with the same logic as existing smart contracts due to the N-version programming. This is a time-consuming task for large or complex Web3 applications.

Therefore, we propose a practical bug bounty platform that uses Intel SGX. Intel SGX allows smart contract developers to provision secret data or code into an encrypted area, Enclave, without revealing or tampering by the physical machine administrator such as bug hunter[15][16]. Our Bug Bounty has an Ethereum Virtual Machine (EVM) and testing tools within the user's Enclave, which detects bugs and generates proof that

TABLE I
COMMON SMART CONTRACT ATTACKS

Contract Name	Year	Exploit Value ($)	Root Cause
The DAO[1]	2016	60M	Re-Entrancy
Parity Multisig 1[2]	2017	30M	Function Visibility
Parity Multisig 2[3]	2017	150M	Exposed self-destruct
dForce 1[4]	2020	25M	Re-Entrancy
Grim Finance[5]	2021	30M	Re-Entrancy
dForce 2[6]	2023	3.6M	Re-Entrancy

a method for triggering the bug found by the user is correct. Then, the user sends the encrypted bug reproduction method and proof to the on-chain smart contract as a bug report. The contract then verifies the bug report, saves it, and pays bounties at the same time.

Since we generate the bug reports within the Enclave, we can guarantee the existence of bugs and the correctness of the method that triggers the bug. In addition, the verification of bug reports and the payment of bounties are done on-chain so that the revealing of bug reports and the payment of bounties can be done atomically. Also, by including the EVM in the Enclave, developers can make any smart contracts eligible for our bug bounty without rewriting them. Note that because the bug report is encrypted by the developers' key, only the bug hunter and developers can know the attack methods.

1) Contribution: Our contributions are as follows

- We propose a practical bug bounty platform for smart contracts called ContractSafeguard that does not require modification of existing smart contracts and can exchange bug reports and bounties atomically
- We design and implement ContractSafeguard for Ethereum smart contracts and evaluate its practicality with historical attack cases, The DAO and the Parity Wallet Hack. Our evaluation shows that our bounty can work with existing smart contracts without modification.
- We confirm the fee (gas) for claiming a bounty and saving a bug report is reasonable.

2) Related Work:

[17] considers a decentralized bug bounty framework in which intermediaries resolve complaints such as non-payment of bounties. Due to the human intermediary, exchanging bug reports and bounties may take time. This can lead to others identifying and attacking the vulnerability.

[18] proposes a protocol for verifiable computing, commitment schemes, and zero-knowledge proofs under the assumption that Intel SGX does not guarantee confidentiality. As an example use case, they discuss a bug bounty platform using smart contracts. Note that it is not a platform for smart contracts and does not describe how to identify smart contract bugs. We propose a bug bounty for smart contracts and describe a mechanism to generate proof that a method for triggering the bug is correct.

II. Background

A. Attacks for Smart Contracts

There have been many attacks happened on the blockchain. Table. I shows an example of them caused by bugs in smart contracts.

Re-Entrancy is a type of attack that exploits the fallback function of a contract. This function is triggered when a message is sent to a contract with no arguments, or when no function matches the message. An attacker can use this function to call the functions of the target contract repeatedly. The DAO, which caused the split into Ethereum and Ethereum Classic, was also a victim of this type of attack[1].

The Parity Multisig Wallet was designed to serve as the wallet library via $delegatecall$, but there were mistakes about function access permissions[2]. Additionally, it did not check for double initialization, such as setting the owner. Therefore, the attacker could easily modify the wallet owner. $delegatecall$ is a low-level function to delegate logic execution to another contract while storing data in its own contract. When using proxy patterns[19] or the common library, contracts use $delegatecall$.

B. Bug Bounty Platform

Bug Bounty Platform is a system that uses bounties as an incentive for third parties to find software bugs. It provides a platform to connect third parties with software developers. There is also a platform specialized for smart contracts[9][8].

In Bug Bounty, users first search for software bugs and summarize how to reproduce the bug as a bug report. The user then submits the bug report to the bounty system. Developers check the report, and if they confirm the existence of the bug, they pay the bug hunter a bounty.

C. Intel SGX

Intel Software Guard Extensions (Intel SGX) is a hardware-based technology for application developers who are seeking to protect secret code and data from disclosure or modification[15]. It creates encrypted areas in memory called Enclave that cannot be accessed even by an OS or physical machine administrator with a higher privilege level. It offers integrity and confidentiality guarantees to security-sensitive computations performed on commodity computers.

SGX provides the remote attestation function for proving to a user that the Enclave' s configuration (i.e., code and static data) is correct. After attesting the Enclave, users can then establish a secure channel with the Enclave and provision the encrypted code or other secrets to the Enclave[16].

III. Proposal

This chapter describes a system architecture of the innovative and practical Bug Bounty Platform that meets the requirements described in chapter I.

This Bug Bounty has the following features.

- The user (bug hunter) searches for bugs in a local EVM within the Enclave, and a predefined test program in the Enclave generates proof of the existence of the bug and

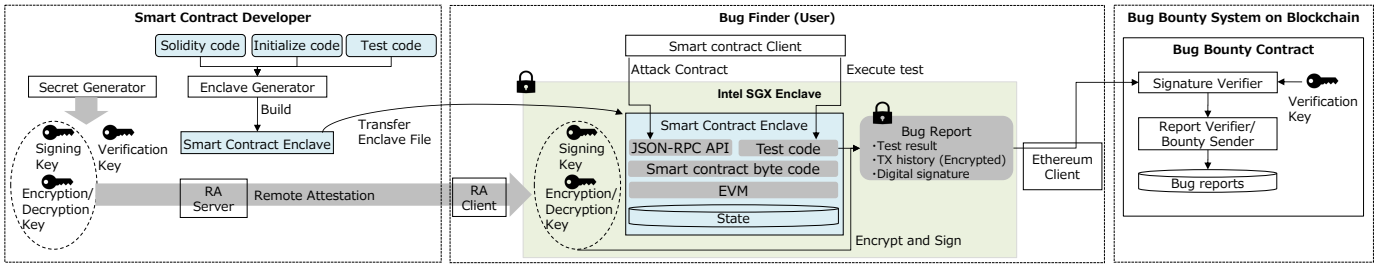

Fig. 1. System Architecture

the correctness of the method triggering the bug as a bug report.
- The bug report is encrypted with a developer's secret key to ensure that bugs are revealed only to the finder and the developer who developed the bounty-targeted program.
- On-chain contract verifies the proof and pays bounty to exchange the bug report for bounty atomically.
- EVM is run within Enclave to allow existing smart contracts to be used without modification

Fig. 1 shows the overall system architecture. It consists of an environment for developers, a test execution environment for users, and a bug bounty contract on Ethereum.

A. Smart Contract Developer Environment

In the developer environment, we generate keys and Enclave file. First, the Secret Generator generates three keys. The *Signing Key* and *Verification Key* are key pairs for public-key cryptography. The *Encryption/Decryption Key* is the symmetric key to encrypt and decrypt bug reports.

Next, the Enclave Generator builds an Enclave file containing EVM, the bytecode of the smart contract, and test codes. The developer inputs three data into the generator. The first one is the source code of the smart contract that is the subject of the bounty. The Enclave Generator compiles it and generates bytecode for EVM. The second one is the initialization code, which initializes the smart contract's state when users do tests. For example, it sets the owner address of a token or the initial token amount held by a user via the API similar to the normal Ethereum JSON-RPC API. The third one is the test code that checks whether the smart contract's state is abnormal, e.g., the balance decreases from its initial state. Since only a secret key known only to the initialization code can move a predefined user's token, if the user's token is decreasing, it can be determined that some vulnerability is the cause. It also obtains the state via the API in the same way as the initialization code. Since EVM is included in the Enclave, developers can make any smart contract eligible for the bug bounty. Note that since initialization code, test code, and smart contract byte code are embedded in the Enclave, users can not tamper them.

Finally, the developer uses the remote attestation function of Intel SGX to provision the previously created signing and encryption Keys to the Enclave in the bug hunter's environment. The remote attestation allows the developer to deploy these keys in the Enclave without revealing the secret data to the end user.

B. User's Environment

In the user's test environment, the user sends a transaction to the smart contract in Enclave, executes the test, and claims the bounty. First, the user downloads the smart contract Enclave from the developer. Then, the user launches the Enclave and executes the initialization code predefined by the developer. The smart contract's bytecode and state and initialization code are in the Enclave, so they cannot be directly modified by the user. The user can only manipulate the smart contract's state via the normal Ethereum JSON-RPC API exported by the Enclave. To make the smart contract's state abnormal, which is not intended by the developer, the user issues several transactions via the API. The user then asks Enclave to execute the test code, which executes the test cases predefined by the developer and generates a bug report if the test fails. The bug report includes a list of past transactions, IDs of failed tests, and a digital signature. The transaction list is encrypted with the *Encryption Key*. This means that even if the bug report becomes public, only the finder and developer can know the attack method. Enclave then digitally signs the bug report with the *Signing Key*. This ensures that the test has failed and the bug report is indeed generated within the Enclave. The bug report is finally saved as a file outside Enclave.

After that, the user sends the bug report to the Bug Bounty Contract on Ethereum using any Ethereum Client to obtain the bounty.

C. Bug Bounty Contract on Ethereum

The bug bounty contract verifies the bug report and pays bounty to users. When the contract receives a bug report from a user, it first verifies its digital signature. The developer should register the *Verification Key* corresponding to the *Signing Key* in the contract in advance. Successful digital signature verification means that the test has indeed failed and the bug report is valid. The contract then checks which tests have failed and stores the bug report. Finally, the contract pays a bounty to the user. The amount of the bounty shall be specified corresponding to the test case in advance by the developer.

IV. Implementation and Experiment

This chapter describes the implementation and experiment of ContractSafeguard.

A. implementation

We implemented ContractSafeguard using Intel SGX SDK 2.18. EVM for Enclave was implemented using eEVM[20], an EVM implementation for Open Enclave, and we added some of the latest opcodes, such as $revert$. As for the API, we implemented only an interface to send arbitrary transactions rather than the entire Ethereum JSON-RPC API.

We used ECDSA on the secp256r1 curve to generate digital signatures. Bug reports were encrypted with AES-128-GCM. Note that since the transaction list can be large, we compressed it before encryption to reduce the file size of bug reports.

B. experiment

We validate the practicality of ContractSafeguard from the following two perspectives.

1) Whether ContractSafeguard can use smart contracts that have been attacked in the past without modification
2) Whether the data size and required gas size to store the bug report in Ethereum is reasonable

We chose two smart contracts for the experiment, one emulating The DAO and the other emulating the Parity Wallet, described in chapter I. The DAO had a large hack that caused a fork of Ethereum, and the Parity Wallet Hack had a contract that uses a $delegatecall$, often used to call a library and update logic flexibly. In fact, 40.8% of the contracts deployed in the last million blocks use $delegatecall$.

We experimented on a machine with Intel Core i5-10210U CPU@1.60GHz, 8GB memory, and HDD. The bug bounty smart contract was deployed on a local Ethereum network.

1) Re-Entrancy Attacks:

We used an ERC-20 contract with a Re-Entrancy vulnerable as a test target contract.

As the developer, we wrote *initialization code* that sets the balance of the ERC-20 contract itself to a specific value. The *test code* verifies that the balance of the contract has not been reduced from its initial value. The contract is intended to allow withdrawals of an amount at most equal to the deposit amount. Therefore, in the absence of vulnerabilities, the balance of the contract itself should not be less than the initial value.

In this experiment, we first deployed a Malicious Withdrawer contract that has a fallback function that calls the withdraw function of the ERC-20 contract and deposited a small amount into the ERC-20 contract. Then, we called the Malicious Withdrawer contract to attack. The contract called the withdraw function of the ERC-20 contract, and the ERC-20 contract transferred ETH to the Malicious Withdrawer contract. Then, the Malicious Withdrawer contract's fallback function was executed, and it called the withdraw function of the ERC-20 contract again, and this loop was repeated. As a result, we could withdraw more than we had first.

The size of the encrypted bug report was 1,085 bytes. The cost of storing it on Ethereum and verifying the signature was 2,492,262 gas or about 150 USD[1].

[1]As of November 2023, 1 ETH is worth roughly 2000 USD, and a gas price is about 30 gwei. $gas\,cost\,[\mathrm{USD}] = gas\,used\,[\mathrm{gas}] \times gas\,price\,[\mathrm{gwei/gas}] \times 2^{-9} \times ETH\,price\,[\mathrm{USD/ETH}]$

2) Default Visibility of Initialization Function:

We tested a smart contract similar to the Parity Wallet Hack. Parity Wallet has two contracts: a Wallet Library contract as a common library and a Wallet contract that users deploy and use themselves. Wallet contract calls Wallet Library contract with $delegatecall$ in a fallback function.

We wrote an *initialization code* that deploys the Wallet Library contract and sets the initial balance of an owner address to a specific value. The *test code* checks whether the Wallet balance for the owner's address has been reduced. Only the developer has the private key that can manipulate the balance of the address, and the balance cannot be changed except by vulnerabilities.

In this experiment, we exploited the fact that the initialization function of the Wallet Library contract is public. Originally, the initialization function was called only once during library initialization. But, since the function's scope remained public, we could call it from outside the library using the fallback function of the Wallet contract. As a result, we could get ownership of the Wallet and withdraw the balance.

The size of the encrypted bug report generated was 241 bytes. The cost of storing it on Ethereum and verifying the signature was 1,904,042 gas or about 114 USD[1].

3) Consideration:

The first experiment confirms that vulnerabilities significantly impacted Ethereum, such as the Re-Entrancy attack, could be tested in ContractSafeguard. In addition, as in the second experiment, contracts using $delegatecalls$, often used with libraries, could also be tested in ContractSafeguard. Furthermore, in both experiments, the gas used to store them on Ethereum and verify signatures was affordable and below the maximum gas usage: Ethereum transactions can use up to 30 million gas. Therefore, ContractSafeguard is practical because any smart contract can be used without modification, and the gas cost is reasonable.

V. Conclusion

We proposed ContractSafeguard, practical bug bounty platform for smart contracts with Intel SGX, which can use existing smart contracts without modification and exchange bug reports for bounties atomically. ContractSafeguard executes EVM, smart contracts, and test codes within the user's Enclave. The test code generates a proof that the bug reproduction method found by the user is correct. The on-chain smart contract then verifies the proof and pays the bounty. ContractSafeguard can prevent users from tampering with the proof by leveraging Intel SGX. In addition, developers can make any smart contract eligible for bug bounties since EVM is run in the Enclave. We implemented ContractSafeguard for Ethereum and tested its practicality on the historical Ethereum attack cases, The DAO and Parity Wallet Hack. The experiment confirmed that ContractSafeguard can handle existing smart contracts without modification, and gas cost is affordable.

References

[1] Zubin Pratap. *Reentrancy Attacks and The DAO Hack Explained — Chainlink*. https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/. 2022.

[2] Lorenz Breidenbach et al. *An In-Depth Look at the Parity Multisig Bug*. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/. 2017.

[3] Santiago Palladino. *The Parity Wallet Hack Reloaded*. https://blog.openzeppelin.com/parity-wallet-hack-reloaded. 2017.

[4] William Foxley and Nikhilesh De. *Weekend Attack Drains Decentralized Protocol dForce of $25M in Crypto*. https://www.coindesk.com/markets/2020/04/19/weekend-attack-drains-decentralized-protocol-dforce-of-25m-in-crypto/. 2021.

[5] Shaurya Malwa. *Fantom DeFi Project Grim Finance Exploited for $30M*. https://www.coindesk.com/tech/2021/12/20/fantom-defi-project-grim-finance-exploited-for-30m/. 2021.

[6] ROB BEHNKE. *EXPLAINED: THE DFORCE HACK (FEBRUARY 2023)*. https://www.halborn.com/blog/post/explained-the-dforce-hack-february-2023. 2023.

[7] Viimeksi muokattu. *Smart contract security — ethereum.org*. https://ethereum.org/fi/developers/docs/smart-contracts/security/. 2023.

[8] *HackenProof*. https://hackenproof.com/.

[9] *immunefi*. https://immunefi.com/.

[10] Ionut Arghire. *Researchers Claim Wickr Patched Flaws but Didn't Pay Rewards*. https://www.securityweek.com/researchers-claim-wickr-patched-flaws-didnt-pay-rewards/. 2016.

[11] Lorenz Breidenbach et al. "Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1335–1352. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/breindenbach.

[12] Peter Murray (@yarrumretep), Nate Welch (@flygoing), and Joe Messerman (@JAMesserman). *ERC-1167: Minimal Proxy Contract*. June 2018. URL: https://eips.ethereum.org/EIPS/eip-1167.

[13] pinkiebell (@pinkiebell). *ERC-3448: MetaProxy Standard*. Mar. 2021. URL: https://eips.ethereum.org/EIPS/eip-3448.

[14] Nick Mudge (@mudgen). *ERC-2535: Diamonds, Multi-Facet Proxy*. Feb. 2020. URL: https://eips.ethereum.org/EIPS/eip-2535.

[15] Frank McKeen et al. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013. ISBN: 9781450321181. DOI: 10.1145/2487726.2488368. URL: https://doi.org/10.1145/2487726.2488368.

[16] *Attestation Services for Intel(R) Software Guard Extensions*. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/attestation-services.html.

[17] Lital Badash et al. "Blockchain-Based Bug Bounty Framework". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 239–248. ISBN: 9781450381048. DOI: 10.1145/3412841.3441906. URL: https://doi.org/10.1145/3412841.3441906.

[18] Florian Tramèr et al. "Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge". In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017, pp. 19–34. DOI: 10.1109/EuroSP.2017.28.

[19] OpenZeppelin. *Proxy Patterns*. https://blog.openzeppelin.com/proxy-patterns.

[20] *eEVM*. https://github.com/microsoft/eEVM.