Poster: Post-Deployment Privacy Enhancement for Smart Contracts

Tongqi Wang, Jin Wu, Jian Dong
Harbin Institute of Technology, Harbin, China
{tongqi6, wujin, dan}@hit.edu.cn

***Abstract*—Addressing privacy exposures for post-deployed smart contracts on the blockchain is challenging due to its transparent nature. Our studies reveal that privacy leakage may persist even after applying state-of-the-art protection methods, such as zero-knowledge proofs. This work thoroughly characterizes the privacy risks that persist after contract upgrades aimed at enhancing privacy. To the best of our knowledge, this is the first work to highlight the issue of post-upgrade privacy risks. We propose an approach to identify these risks by analyzing variable dependencies during complex transaction sequences. Additionally, we propose a mitigation strategy to trace the sources of privacy leakages and conceal sensitive information by manipulating involved variables through transaction sequences.**

I. Introduction

Extensive research efforts have been undertaken to address data privacy concerns in Ethereum smart contracts. The mainstream is the zero-knowledge proof (ZKP)-based privacy protection approaches [1], [2] (denoted by ZKAPPs), which does not rely on trusted third parties, is more suitable for non-interactive blockchain scenarios, and can support a wider range of applications. In the ZKAPP, privacy protection involves concealing/encrypting sensitive data and validating state-correct updates using ZKPs. To put the ZKAPP into practice, it is imperative to delineate clear strategies for privacy protection *prior to* the deployment. However, smart contracts *after* deployment may still have privacy vulnerabilities due to various causes, such as coding defects (e.g., SWC-136) [3] or emerging privacy risks [4].

A straightforward fix for a deployed contract with privacy issues is to replace it with an enhanced version using ZKAPP. However, this leads to the loss of accumulated historical data and states, causing unexpected losses and inconvenience for owners and users. Contract upgrading offers a solution by changing the business logic while preserving the state. Privacy protections like ZKAPP can be implemented through upgrading. Despite this, there are fundamental limitations. ZKAPP encrypts data, but attackers might infer plaintext from previous public data since the blockchain state is continuous. Thus, upgraded contracts might still be vulnerable to privacy leaks. The leakage occurs in unexpected ways due to unpredictable user behavior, complicating the issue. No current work addresses contract repair for post-deployment privacy issues.

Our research aims to enhance the privacy of deployed contracts where privacy vulnerabilities are identified post-deployment. We implement privacy-focused upgrading to specifically protect already exposed private data and *prevent further leakage*. To thoroughly understand post-upgrade privacy risks (PUPRs), we conduct an extensive study, revealing that these risks can persist even after applying ZKAPPs through contract upgrades. For contracts with potential privacy risks, we propose a strategy to analyze PUPRs by identifying vulnerable variables that might expose sensitive data. Initially, we identify these vulnerable variables. Subsequently, we trace the source of potential PUPRs using ciphertext data dependency analysis. To mitigate PUPRs, we introduce a strategy to eliminate their source. This involves using a transaction sequence generator to securely handle sensitive data within the vulnerable variables, ensuring it remains protected. To validate our approach, we have developed a prototype, PD-Priv, which is the *first-ever* tool designed for post-deployment privacy enhancement in Ethereum applications.

II. Persistence of PUPRs

We first discuss the persistence of potential PUPRs in upgraded contracts using ZKAPP. When a privacy breach is found, developers must provide a new contract or patch with ZKAPP to upgrade the contract. The upgraded contract moves sensitive data computations to functionality-equivalent off-chain computations. Besides, sensitive data is changed from plaintext to ciphertext in new variables. Therefore, all new ciphertext-type variables have an important property that *everyone can infer the original plaintext of the sensitive data stored in these variables*. As a result, in the new transaction, the reading and writing of sensitive data affected by these variables also become insecure, leading to the exposure of the affected sensitive data. These variables are termed "sources" of PUPRs. There are three types of PUPRs, including *Reading Risk*, *Writing Risk - Leakage*, and *Writing Risk – Manipulation*.

- *Reading Risk* (denoted by RR) represents a type of PUPR where the "victim" ciphertext variable to be revealed in the future is already public revealed.
- *Writing Risk - Leakage* (denoted by WRL) represents a type of PUPR where a newly written "victim" ciphertext variable exposed its sensitive data to public.
- *Writing Risk – Manipulation* (denoted by WRM) represents a type of PUPR where newly written "victim" ciphertext variable is *maliciously manipulated* by someone.

III. Elimination of PUPRs

We propose eliminating PUPRs by identifying potential PUPRs and their "sources" to protect user information. To

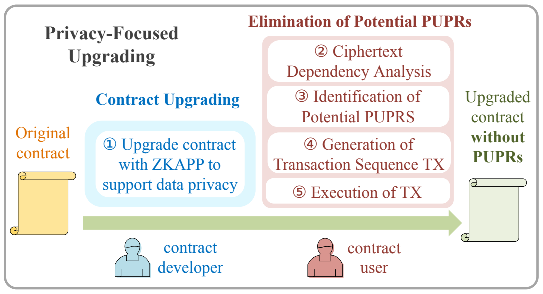

Privacy-Focused Upgrading
Elimination of Potential PUPRs
② Ciphertext Dependency Analysis
③ Identification of Potential PUPRS
④ Generation of Transaction Sequence TX
⑤ Execution of TX
Contract Upgrading
① Upgrade contract with ZKAPP to support data privacy
Original contract
Upgraded contract without PUPRs
contract developer
contract user

Fig. 1. Workflow of *privacy-focused upgrading* supported by PD-Priv.

```
1  contract Odd_Even{
2     address public admin;
3     uint public counter;
4     uint private sum$_{admin}$;
5     mapping(address => uint) private balance[$\alpha$]$_{\alpha}$;
6     mapping(uint => address) public players;
7     address public winner;

8     function constructor(){
9        admin = me;
10    }

11    function buy(){
12       balance[me]$_{me}$ += uint(msg.value);
13    }

14    function transfer(uint amount$_{me}$, address to){
15       ...
16       balance[me]$_{me}$ -= amount$_{me}$;
17       balance[to]$_{to}$ += amount$_{me}$;
18    }

19    function sell(uint amount) {
20       ...
21    }

22    function start(uint number$_{admin}$) {
23       require(me == admin);
24       ...
25       sum$_{admin}$ += number$_{admin}$;
26    }

27    function play(uint number$_{me}$) {
28       ...
29       require(me != admin);
30       require(counter < 2);
31       balance[me]$_{me}$ -= 1;
32       sum$_{admin}$ += number$_{me}$;
33       players[counter] = me;
34       counter += 1;
35    }

36    function select_winner() {
37       require(me == admin);
38       ...
39       if (counter == 2) {
40          uint index = sum$_{admin}$ % 2;
41          winner = players[index];
42          balance[winner]$_{winner}$ += 2;
43       }
44       counter = 0;
45    }
46 }
```

Fig. 2. The upgraded contract Odd_Even. The original contract version disregards the red text.

this end, this work first performs data dependency analysis to trace ciphertext back to its plaintext. Once a ciphertext can be connected to a plaintext, a risk, and its "source" are identified. Eliminating "sources" prevents PUPRs by ensuring sensitive data is not exposed. Since upgraded ciphertext leaks information, involving privacy parameters that don't leak information in computations ensures the resulting ciphertext remains secure. This work applies a transaction sequence generator to output a sequence that can eliminate the "sources" of potential PUPRs, which can be formally proved.

We design a tool PD-Priv to perform *privacy-focused upgrading* to contracts with privacy issues. Unlike traditional contract upgrade, PD-Priv ensures private information appropriately protected after upgrades, as illustrated in Fig. 1.

IV. Case Study

In this section, we perform the *privacy-focused upgrading* proposed in this work on a contract application (Odd_Even, as shown in Fig. 2) with the SWC-136 privacy issue[1]. We first illustrate three PUPRs on Odd_Even. Then, we show the elimination of these PUPRs.

In this contract, each user has a private account to store their token balance, supporting players participating in the game. To participate, players call the `play` function, each providing a `number` to be added to `sum`. Whether `sum` is odd or even will

[1]Publicly evaluated on the Sepolia Testnet: https://sepolia.etherscan.io/address/0xb8664b572118716dded6ad2ea51be7c4411b5d69.

TABLE I
PUPRs in the upgraded Odd_Even.

Type of PUPRs	"Victim" of PUPRs	Location of PUPRs
RR	sum_{admin}	`select_winner`, line 40
WRL	$balance_{me}$	`buy`, line 12
WRM	sum_{admin}	`play`, line 32

TABLE II
Elimination for PUPRs in the upgraded Odd_Even.

Contract User	Type of PUPRs	"Source" of PUPRs	Transaction Sequence for Contract User
admin	RR WRL WRM	sum_{admin} $balance_{admin}$ sum_{admin}	`[transfer, start]`
non-admin	WRL	$balance_{non-admin}$	`[transfer]`

determine the winner between the two players. The contract developer aims to prevent attackers from reading sensitive data, such as `sum` and `balance`, so these data types are set to `private`. However, there is a common misconception that private type variables cannot be read, leading to SWC-136 privacy issues in the original Odd_Even contract.

Once developers become aware of the exposure of `private` variables, they upgrade the contract code, as shown in phase 1 of Fig. 2. In the upgraded version, a variable with a subscript indicates that its data type has been converted into ciphertext, with the subscript representing the owner of the ciphertext's private key. The transaction sender is denoted by me. We list the PUPRs in the upgraded Odd_Even contract, along with their "victim" ciphertext variable and occurrence locations, as shown in Table I.

For each contract user, PD-Priv analyzes each vulnerable variable and identifies the "source" of each PUPR. It then provides a transaction sequence to eliminate the "source", as shown in Table II. Involved users can eliminate PUPRs by executing these generated sequences.

V. Conclusion

For smart contracts with post-deployment privacy risks, a mere regular contract upgrade proves to be inadequate. In this paper, we highlight the persistence of privacy risks even after such upgrades. Subsequently, we identify and mitigate potential privacy risks to prevent their continuation.

References

[1] S. Steffen, B. Bichsel, M. Gersbach, N. Melchior, P. Tsankov, and M. Vechev, "zkay: Specifying and enforcing data privacy in smart contracts," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pp. 1759–1776, 2019.
[2] S. Steffen, B. Bichsel, R. Baumgartner, and M. Vechev, "Zeestar: Private smart contracts by homomorphic encryption and zero-knowledge proofs," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 179–197, IEEE, 2022.
[3] SWC-registry, "Smart contract weakness classification (swc)," 2020.
[4] N. Ivanov, Q. Yan, and A. Kompalli, "Txt: Real-time transaction encapsulation for ethereum smart contracts," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 1141–1155, 2023.

Post-Deployment Privacy Enhancement for Smart Contracts

Tongqi Wang, Jin Wu, Jian Dong
Harbin Institute of Technology
{tongqi6,wujin,dan}@hit.edu.cn

Privacy issues in the post-deployed smart contracts

Transparency vs. Confidentiality

Transparency: all data are revealed public in contracts
Confidentiality: sensitive data should not be exposed in sealed-bid auction, voting, healthcare, etc

Privacy issues of deployed contracts

Causes {
- ➢ code defects (e.g., SWC-136)
- ➢ emerging privacy risks

Fix → contract upgrading

What happens when contract upgrading encounter privacy issues?

Contract Upgrading

- √ Incorporate new features
- √ Address security vulnerabilities
- ×Address privacy vulnerabilities

ZKP-based Privacy Enhancement (ZKAPP)

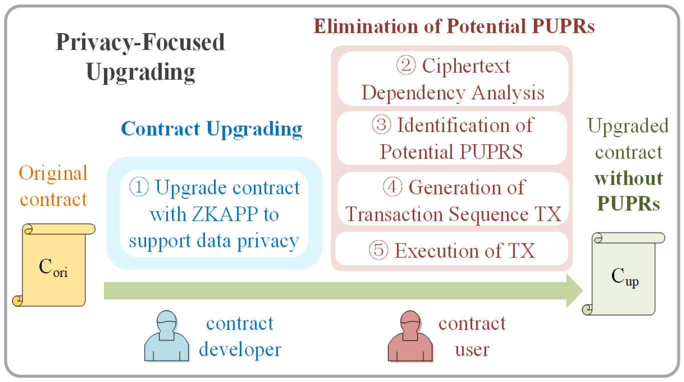
Encrypt sensitive data and validate state-correct updates via ZKP

Features {
- ➢ NOT rely on any trusted third party
- ➢ NOT require interactive participants
- ➢ Support a wider range of Ethereum contract applications

Contract Upgrading + ZKAPP: Privacy risks still persist!

Post-Upgrade Privacy Risks **(PUPRs)**
- ① Reading Risk (RR): the "victim" ciphertext variable to be revealed in the future is already public revealed
- ② Writing Risk - Leakage (WRL): a newly written "victim" ciphertext variable exposed its sensitive data to public
- ③ Writing Risk - Manipulation (WRM): a newly written "victim" ciphertext variable is maliciously manipulated by someone

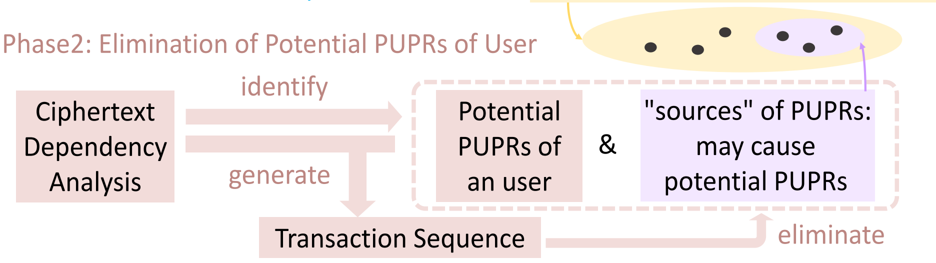
PD-Priv: Performing *privacy-focused upgrading* for privacy issues


Privacy-Focused Upgrading
Elimination of Potential PUPRs
② Ciphertext Dependency Analysis
③ Identification of Potential PUPRS
④ Generation of Transaction Sequence TX
⑤ Execution of TX
Contract Upgrading
① Upgrade contract with ZKAPP to support data privacy
Original contract
C_{ori}
Upgraded contract without PUPRs
C_{up}
contract developer
contract user

Phase1: Contract Upgrading with ZKAPP
- Transfer sensitive data into ciphertexts
- Ensure correctness of ciphertexts via ZKP

New ciphertext variables: wherein everyone can infer plaintext/ciphertext pair

Phase2: Elimination of Potential PUPRs of User


Ciphertext Dependency Analysis
identify
generate
Potential PUPRs of an user
&
"sources" of PUPRs: may cause potential PUPRs
Transaction Sequence
eliminate

Case Study for repairing a contract with SWC-136 privacy issues

Original Contract Odd_Even with SWC-136 privacy issues

```
1   contract Odd_Even{
2       address public admin;
3       uint public counter;
4       uint private sum;
5       mapping(address => uint) private balance;
6       mapping(uint => address) public players;
7       address public winner;

8       function constructor(){
9           admin = me;
10      }

11      function buy(){
12          balance[me] += uint(msg.value);
13      }

14      function transfer(uint amount, address to){
15          ...
16          balance[me] -= amount;
17          balance[to] += amount;
18      }

19      function sell(uint amount) {
20        ...
21      }
22      function start(uint number) {
23          require(me == admin);
24          ...
25          sum += number;
26      }

27      function play(uint number) {
28          ...
29          require(me != admin);
30          require(counter < 2);
31          balance[me] -= 1;
32          sum += number;
33          players[counter] = me;
34          counter += 1;
35      }

36      function select_winner() {
37          require(me == admin);
38          ...
39          if (counter == 2) {
40              uint index = sum % 2;
41              winner = players[index];
42              balance[winner] += 2;
43          }
44          counter = 0;
45      }
46  }
```

privacy-focused upgrading

1) Upgrade with ZKAPP

- Transfer sensitive data from plaintext to ciphertext

2) Potential **PUPRs** in upgraded version

Type of PUPRs	"Victim" of PUPRs	Location of PUPRs
RR	*sum*	select_winner, line 40
WRL	*balance*	buy, line 12
WRM	*sum*	play, line 32

3) Elimination of **PUPRs**

Contract User	Type of PUPRs	"Source" of PUPRs	Transaction Sequence for Contract User
admin	RR WRL WRM	*sum* *balance* *sum*	[transfer, start]
non-admin	WRL	*balance*	[transfer]

The complete process of **privacy-focused upgrading** for Odd_Even is available at https://sepolia.etherscan.io/address/0xb8664b572118716dded6ad2ea51be7c4411b5d69