ZK-Hammer: Leaking Secrets from Zero-Knowledge Proofs via Rowhammer

Junkai Liang[1,3,4], Xin Zhang[2,3,4], Daqi Hu[2,3,4], Qingni Shen[2,3,4,*], Yuejian Fang[2,3,4], Zhonghai Wu[1,2,3,4,*]
[1]School of Computer Science, Peking University
[2]National Engineering Research Center for Software Engineering, Peking University
[3]School of Software and Microelectronics, Peking University
[4]PKU-OCTA Laboratory for Blockchain and Privacy Computing, Peking University
Email: ljknjupku@gmail.com, zhangxin00@stu.pku.edu.cn, hudaqi0507@gmail.com
qingnishen@pku.edu.cn,fangyj@ss.pku.edu.cn, wuzh@pku.edu.cn

***Abstract*—Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARK) schemes have been a promising technique in verified computation. Zk-SNARK schemes were designed to be mathematically secure against cryptographic attacks and it remains unclear whether they are vulnerable to fault injection attacks. In this work, we provide a positive answer by presenting ZK-Hammer, which leaks secrets from zk-SNARK schemes via Rowhammer. We incur faults in the exponentiate variables in the Quadratic Arithmetic Program (QAP) problem. Then we analyze the faulty proof using the bilinear pairing technique and manage to recover the secret. We employ a Rowhammer fault evaluation in `libsnark` and identify 3 CVEs.**

I. Introduction

Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARK) schemes have significantly evolved from their conceptual origins in complexity theory and cryptography [1] to their current role as fundamental components, enabling a wide array of practical applications including blockchain payments [2], smart contract [3], and other academic areas like machine learning [4], multiparty computation [5] and post-quantum cryptography [6]. As a mechanism for an untrusted prover to convince a verifier of the correctness of a computation without disclosing any other information (zero-knowledge), zk-SNARKs have transitioned from theoretical constructs to widely used practical implementations over the past decade [7]–[9].

Existing zk-SNARK schemes have been designed to have cryptographic guarantees such as soundness, completeness and zero knowledge. However, those nice properties only hold in ideal conditions, e.g., circumstances that do not consider side-channel attacks, correct implementations of upper-level applications, etc. There are no adequate works that recognise the severity of the hardware fault attacks. A powerful technique in this category is a fault injection attack through Rowhammer [10].

Rowhammer can inject a software-induced fault in the main memory of a commodity system through hardware bit flips in DRAM [11]–[13]. Since the Rowhammer vulnerability was discovered in hardware in 2014 [11], it has been exploited to break two categories of practical cryptographic schemes, signature schemes and encryption schemes (a.k.a key exchange mechanisms). For the first category, there have been numerous successful attacks against ECDSA [14]–[17], EdDSA [18] and post-quantum signatures such as LUOV [19] and Dilithium [20]. For the second category, successful attacks against RSA [21]–[23], Frodo [24] and Kyber [25], [26] have also been reported. Those works expand the impact and highlight the severity of Rowhammer attacks.

Despite the growing threat of Rowhammer and all the progress made in zk-SNARK field, we observe that the vulnerability of zk-SNARK schemes against Rowhammer-based attacks has not been adequately evaluated. Thus, we are interested in the following questions:

1) *If a fault occurs in zk-SNARK schemes, is it meaningful to an attacker?*
2) *If so, can an attacker exploit the vulnerability and deduce some secrets?*
3) *Further, is the vulnerability general to other zk-SNARK schemes and can we give some countermeasures?*

Our work: In this paper, we offer affirmative responses by presenting ZK-Hammer: a thorough fault analysis and evaluation of Quadratic Arithmetic Program (QAP) based zk-SNARK schemes. Due to the complexity of zk-SNARKs (see Section II), most of the faults will simply result in a failure in the proof generation which cannot be exploitable. A general challenge in our attack is to find the meaningful position of the fault which may result in the leakage of secrets. We managed to solve this challenge by designing a *Fault + Recover* process. In the faulting phase, we introduce faults to the secret we aim to leak in the proof generation. The faulty secret will be multiplied by its selector polynomial, subjected to a group exponentiation and then published as part of the proof. In the recovery phase, with the faulty proof, we do a modified verification. Unlike the original verification algorithm, we multiply a correction term in the pairing operation. Note that the term is correlated to the fault and if this modified verification passes, we can deduce the fault as bits of the secret leakage. We have designed an algorithm to do this phase automatically and we further emphasize our analysis applies

This work was supported by the National Natural Science Foundation of China (Grant No. 61672062). * Corresponding authors.

to all the zk-SNARK schemes in the QAP category.

In our evaluation, we have implemented range proof as an example application of `libsnark`. Utilizing Rowhammer to inject a one-bit flip, we got 160 faulty proofs. Utilizing our proposed analysis algorithm, we can leak more than 80% information of the secret (personal balance in the range-proof case) and thus break the zero-knowledge property of zk-SNARK schemes. To mitigate our attack, we provide a few algorithmic defences specific to zk-SNARK and some general hardware defences which we have reported to `libsnark`.

Contributions: The main contributions of this paper are as follows:

- To the best of our knowledge, we introduce the first fault injection analysis and evaluate ZK-Hammer against zk-SNARK schemes for general circuits. ZK-Hammer identifies vulnerable parameters and contains a necessary algorithm to recover the secrets.
- We employ ZK-Hammer to other QAP-based zk-SNARK schemes (i.e., Groth and PGHR) and demonstrate our attack applies to other schemes in the QAP category.
- We evaluate Rowhammer attacks against the GGPR scheme using the recent popular cryptographic library, i.e., `libsnark`. When faulting the secret in our test example, we achieve 80% leakage of information with 160 faulty proofs.
- We have provided countermeasures to mitigate our attack. Our countermeasures receive positive responses from developers of `libsnark`.

Responsible Disclosure: We have disclosed our findings to the developers of `libsnark`. We have been assigned 3 CVE numbers (available at https://github.com/liang-junkai/ZK-hammer), which track the fault-injection vulnerability in three zk-SNARK schemes.

II. Background

A. Rowhammer Attacks Against Cryptography

Rowhammer is a software-induced fault in DRAM that can cause bit flips in the main memory of commodity systems [11]. Specifically, when a DRAM row is accessed frequently (or hammered), it may lead to permanent charge leakage, resulting in bit flips in adjacent rows. The rows that are vulnerable are termed *victim rows*, whereas the frequently accessed rows are referred to as *aggressor rows*. A double-sided Rowhammer is shown in Figure 1.

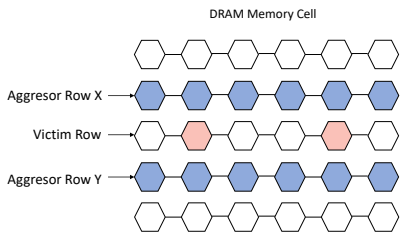


Fig. 1: Demonstration of double-sided Rowhammer

Existing Rowhammer attacks in the context of cryptography target signature schemes [14]–[16], [18]–[21] and key-exchange mechanisms (KEMs) [22]–[26]. For the first category, the attacks against signature schemes use a *Fault+Recover* pattern to leak the secret key. When faulting public keys, a single faulty signature and a valid one are compared to compute the secret key. This comparison process is called differential fault analysis (DFA). When faulting secrets, the attacker leaks one bit of secret key by one faulty signature, and a full recovery requires hundreds of unique faulty signatures. For the second category, the attacker faults the control flow or secret key in the decryption query and recovers usable information from the victim's response. We notice that there is no fault analysis against zk-SNARK schemes and all the analyses above are ad-hoc which cannot apply to other cryptographic schemes. Our work utilizes the *Fault+Recover* idea in the attacks against signature schemes but the derivation is different and more complicated in zk-SNARK schemes than in traditional signature schemes.

B. Zk-SNARK Scheme

Zk-SNARKs are cryptographic proof systems that enable the verification of NP computations with significantly reduced complexity compared to classical NP verification methods. Zk-SNARKs have been instrumental in advancing the field of cryptography by providing efficient and secure methods for proving knowledge without revealing the underlying information. The definition of zk-SNARK is as follows:

Definition II.1 (Zk-SNARK)**.** A zk-SNARK scheme consists of three algorithms: $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ defined as follows:

- $\mathsf{Setup}(\mathsf{pp}) \rightarrow (\mathsf{pk}, \mathsf{vk})$: Given public parameters pp as input, the algorithm outputs proving and verification keys pk and vk.
- $\mathsf{Prove}(\mathsf{pk}, x, w, R) \rightarrow \pi$: The prover generates the proof π related to the key pk, instance and witness pair (x, w), and the relation R.
- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \rightarrow \{0, 1\}$. The verifier uses a verification key vk, the public instance x, and proof π to verify if the proof π is correct.

In addition, the definition of zk-SNARK requires 4 properties:

- **Perfect completeness:** Given $(x, w) \in R$, an honest prover can convince the verifier outputting 1.
- **Knowledge soundness:** Given $x \notin R$, a malicious prover interacting with the verifier can only make it output 1 with negligible probability.
- **Zero knowledge:** Given $x \in R$, a simulator without w can generate a transcript of an interaction between an honest prover and a potentially malicious verifier. This simulated transcript is computationally indistinguishable from a true proof which means the proof reveals no information about w.
- **Succinctness:** The proof size and verification time are sublinear in the statement to be proven.

C. QAP Technique

Our work targets QAP-based zk-SNARKs, which is one of the most fundamental and critical categories. The scheme GGPR [27] is the first practical zk-SNARK for general computation. A main strength of GGPR is that the proof has a constant size (9 group elements). Further, refinements have been made to shrink the proof size from 9 to 3 [28]–[32]. The basic idea of these works is to represent the computation the prover wants to prove as a QAP problem, a.k.a rank-1-constraint-system (R1CS). A formal representation of QAP (R1CS) is as follows:

Definition II.2 (QAP (R1CS))**.** A QAP problem Q over field $\mathbb{F}$ contains three sets of $m+1$ polynomials, $L = \{l_k(x)\}$, $R = \{r_k(x)\}$, $O = \{o_k(x)\}$, for $k = \{0, ..., m\}$, and a target polynomial $q(x)$. Suppose F is the computation we want to prove that has n inputs and $n^{'}$ outputs, for a total $N = n + n^{'}$ IO elements. Then we say Q computes F if there exists $c_1, ..., c_N$ as a valid assignment of inputs and outputs, and coefficients $c_{N+1}, ..., c_m$ such that $q(x)$ divides $p(x)$, where:

$$p(x) = (l_0(x) + \sum_{k=1}^{m}(c_k \cdot l_k(x))) \cdot (r_0(x) + \sum_{k=1}^{m}(c_k \cdot r_k(x))) - (o_0(x) + \sum_{k=1}^{m}(c_k \cdot o_k(x))) \tag{1}$$

III. Threat Model

Aligned with existing Rowhammer attacks against cryptographic schemes [14], [19], [20], our threat model is described as follows:

- An attacker can initiate an arbitrary, unprivileged user process without root privileges, co-located with a victim process in the same DRAM module. The victim process executes a specific zk-SNARK scheme, which the attacker can query for proof. Crucially, the attacker possesses knowledge of the scheme's parameters, including the secret key bit length and the public key description.
- The operating system functions correctly, isolating the attacker from the victim without any software vulnerabilities. The kernel hosting the victim process is considered secure, ensuring effective separation between the attacker and victim. Crucially, the attacker and victim processes do not need to run on the same kernel. As long as they share DRAM modules, a remote attack can be executed.
- The shared DRAM modules are susceptible to Rowhammer bit flips induced by the attacker. The location of a bit flip is specific to a particular DRAM module.

IV. ZK-Hammer: Fault Attack against zk-SNARK

In this section, we present zk-SNARK by analysing the details of QAP-based zk-SNARK schemes and explaining how to inject a fault and make the recovery. Based on the analysis, ZK-Hammer uses 3 components to do a successful attack: pre-processing, online faulting, and post-processing. Generally, ZK-Hammer contains 4 main components (which are shown in Figure 2):

- A guideline: providing the necessary formula derivation about where to inject the faults and how to leak secrets theoretically.
- Pre-processing phase: setups for Rowhammer attack. We use a malicious, unprivileged process to collect system memory information.
- Online faulting phase: mapping the secret to vulnerable locations and starting the row refresh.
- Post-processing phase: deducing the secret leakage according to the faulty proofs.

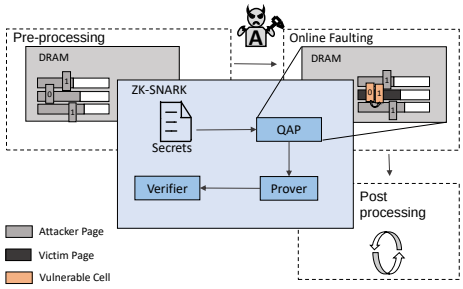


Fig. 2: An overview of ZK-Hammer.

A. Fault Analysis of QAP-based Zk-SNARK

We first briefly describe a classical scheme in this category named GGPR [27], based on which we show how to leak secrets when we incur faults on the parameters.

Gen$(F, 1^{\lambda})$**:** Let F be a function with N input and output values from field $\mathbb{F}$. Use QAP instance $Q = \{q(x), L, R, O\}$ of size m. Let $I_{mid} = N+1, ..., m$ denote the non-IO values. Let e be a bilinear map i.e., $e : G \times G \rightarrow G_T$ and g be a generator of G. Choose random values $s, \alpha, \beta_l, \beta_r, \beta_o, \gamma \leftarrow \mathbb{F}$. Compute public key PK_F for prover:

$$\begin{aligned} PK_F = \{&\{g^{l_k(s)}\}_{k\in I_{mid}}, \{g^{r_k(s)}\}_{k\in[m]}, \{g^{o_k(s)}\}_{k\in[m]}, \\ &\{g^{\alpha l_k(s)}\}_{k\in I_{mid}}, \{g^{\alpha r_k(s)}\}_{k\in[m]}, \{g^{\alpha o_k(s)}\}_{k\in[m]} \\ &\{g^{\beta_l l_k(s)}\}_{k\in I_{mid}}, \{g^{\beta_r r_k(s)}\}_{k\in[m]}, \{g^{\beta_o o_k(s)}\}_{k\in[m]} \\ &\{g^{s^i}\}_{i\in[m]}, \{g_i^{\alpha s^i}\}_{i\in[m]}\} \end{aligned} \tag{2}$$

Compute public key VK_F for verifier:

$$VK_F = \{g, g^{\alpha}, g^{\gamma}, g^{\beta_l\gamma}, g^{\beta_r\gamma}, g^{\beta_o\gamma}, g^{q(s)}, \{g^{l_k(s)}\}_{k\in[N]}\} \tag{3}$$

Here we do not inject faults to PK_F or VK_F, as they are not related to the witness. If we inject faults to the secret parameter s, from the attacker's view, the polynomials are evaluated at another point, which is also valid and we cannot deduce any information. Instead, we choose to inject faults in the next ***Prove*** procedure.

Prove(PK_F, u)**:** On input u, the prover first construct the QAP instance Q for $y \leftarrow F(u)$. As a result, the prover knows every value of $\{c_i\}_{i\in[m]}$ as a solution for the QAP instance

Q. Then she solves $h(x) = (L(x)R(x) - O(x))/q(x)$, and computes the proof π as:

$$\begin{aligned}\{&g^{L_{mid}(s)}, g^{R(s)}, g^{O(s)}, g^{h(s)},\\ &g^{\alpha L_{mid}(s)}, g^{\alpha R(s)}, g^{\alpha O(s)}, g^{\alpha h(s)},\\ &g^{\beta_l L(s)+\beta_r R(s)+\beta_o O(s)}\}\end{aligned} \tag{4}$$

where $L_{mid}(x) = \sum_{k\in I_{mid}} c_k l_k(x)$, $L(x) = \sum_{k\in[m]} c_k l_k(x)$, $R(x) = \sum_{k\in[m]} c_k r_k(x)$ and $O(x) = \sum_{k\in[m]} c_k o_k(x)$

Here $\{c_i\}_{i\in[m]}$ are the witnesses that the prover wants to keep secret. The guarantee of the zero-knowledge property is about the witnesses such that the prover convinces the verifier she knows $\{c_i\}_{i\in[m]}$ which satisfies the QAP problem without revealing those witnesses. When we inject fault on a term in the witness (e.g., c_i) when computing $L_{mid}(s)$, a part of the proof becomes (denote the faulty term as $c_i^{'}$ and the fault as Δc_i):

$$\begin{aligned}g^{L^{'}_{mid}(s)} &= g^{\sum_{k\in I_{mid}, k\neq i} c_k l_k(x) + c_i^{'} l_i(x)}\\ &= g^{\sum_{k\in I_{mid}, k\neq i} c_k l_k(x) + c_i l_i(x) - \Delta c_i l_i(x)}\\ &= g^{L_{mid}(s) - \Delta c_i l_i(x)}\end{aligned} \tag{5}$$

The derivation for $g^{\alpha L^{'}_{mid}(s)}$ is similar, we present as below:

$$g^{\alpha L^{'}_{mid}(s)} = g^{\alpha L_{mid}(s) - \alpha\Delta c_i l_i(x)} \tag{6}$$

The faulty part of the proof can be considered a valid one plus a correction term related to the fault. For $g^{L^{'}_{mid}(s)}$, the correction term is the fault Δc_i multiplies a selector polynomial $l_i(x)$. The selector polynomials are determined by the public function F and thus are public to the attacker. For $g^{\alpha L^{'}_{mid}(s)}$, α is secret but g^{α} is public and the attacker can do linear combinations by group operations. Below we use bilinear pairings in ***Verify*** procedure to eliminate the correction term in the equation and deduce the fault Δc_i. Because the bit flip is either from 0 to 1 or 1 to 0, the fault Δc_i denotes the leakage of the witness c_i.

Verify(VK_F, u, π)**:** First the verifier uses pairing function e to check α and β terms are correct to ensure the consistency of the variables and selector polynomials (e.g., $e(g^{L_{mid}(s)}, g^{\alpha}) = e(g^{\alpha L_{mid}(s)}, g)$). This requires 8 pairings and 3 for the β terms. Then the verifier computes her terms for inputs as $g^{L_{io}(s)} = \prod_{k\in[N]} (g^{l_k(s)})^{c_k}$ and do the final check for QAP divisibility requirement:

$$e(g^{h(s)}, g^{q(s)}) = e(g^{L_{mid}(s)} \cdot g^{L_{io}(s)}, g^{R(s)})/e(g^{O(s)}, g) \tag{7}$$

When a fault (as shown in Equation 5, 6) occurs, the check shown in Equation 7 will fail because of the correction computed above. As the length of c_i is measurable, we can enumerate all the possible Δc_i and compute a related correction term to pass the following equation:

$$\begin{aligned}&e(g^{L^{'}_{mid}(s)} \cdot g^{L_{io}(s)}, g^{R(s)})/e(g^{O(s)}, g)\\ &= e(g^{L_{mid}(s) - \Delta c_i l_i(x)} \cdot g^{L_{io}(s)}, g^{R(s)})/e(g^{O(s)}, g)\\ &= e(g^{-\Delta c_i l_i(s)}, g^{R(s)}) \cdot e(g^{L_{mid}(s)} \cdot g^{L_{io}(s)}, g^{R(s)})/e(g^{O(s)}, g)\\ &= e(g^{h(s)}, g^{q(s)}) \cdot e(g^{-\Delta c_i l_i(s)}, g^{R(s)})\end{aligned} \tag{8}$$

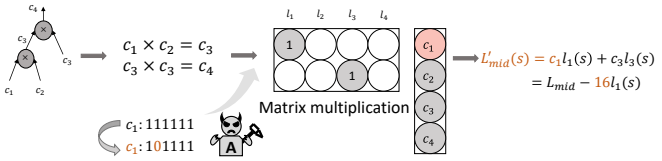


Fig. 3: An example of faulting QAP-based zk-SNARKs. The orange parts illustrate the faulting parameters and values.

when Equation 8 passes, we find the correct fault injected to the witness c_i and we can deduce the original bit value on the same position.

Generality: We apply the fault analysis to other QAP-based zk-SNARKs, such as PGHR [33] and Groth [29] to shown the generality of our attack. As long as the proof is generated by assigning variables to the QAP selector polynomials, our attack applies.

For the PGHR scheme, the only difference is that it uses 3 different group generators g_L, g_R, g_O for $L(x), R(x), O(x)$ respectively. The divisibility check is thus different but we can still extract the faulty term following a similar technique. The derivation is shown as follows:

$$\begin{aligned}&e(g_L^{L^{'}_{mid}(s)} \cdot g_L^{L_{io}(s)}, g_R^{R(s)})/e(g_O^{O(s)}, g)\\ &= e(g_L^{-\Delta c_i l_i(x)}, g_R^{R(s)}) \cdot e(g_L^{L_{mid}(s)} \cdot g_L^{L_{io}(s)}, g_R^{R(s)})/e(g_O^{O(s)}, g)\\ &= e(g^{h(s)}, g^{q(s)}) \cdot e(g_L^{-\Delta c_i l_i(x)}, g_R^{R(s)})\end{aligned} \tag{9}$$

The Groth scheme uses more random values to ensure the zero-knowledge property in only one pairing, e.g., a part of the proof is $L = g_L^{\alpha + L_{mid}(s) + r\delta}, R = g_R^{\beta + R(s) + s\delta}$, when a fault occurs on c_i, we can also extract the fault using a pairing operation:

$$e(L^{'}, R) = e(L, R) \cdot e(g^{\Delta c_i l_i(s)}, R) \tag{10}$$

Figure 3 shows an example of our fault on a simple computation with QAP representation. The computation shown in circuit form is first transformed to 2 constraints represented by multiplications. Then the prover constructs the corresponding selector polynomial matrix and multiplies it with the witnesses. Our fault takes place here right before the polynomial L_{mid} is computed. In our example, we fault the second bit of c_1 from 1 to 0, decreasing the absolute value by 16. The value $L^{'}_{mid}(s)$ used in the proof can be considered as a valid value $L_{mid}(s)$ subtracts a correction term related to the fault. If we find the value of the term, we know the position and direction of the fault and can deduce the original bit of c_1.

B. Offline Preparing Phase

To induce Rowhammer faults in c_i, we identify flippable locations in memory and gather addresses in adjacent hammering rows through *memory profiling*, which occurs before initializing the victim. We organize virtual pages in consecutive rows of the same DRAM bank and decode the DRAM addressing mechanism in two steps. First, do the virtual-to-physical address translation. This can be done through the

leverage of the consistent behaviour of the buddy allocator to encourage the kernel to allocate physically contiguous memory [34]. Second, do the physical-to-DRAM address mapping. This can be done through the leverage of a DRAM row buffer timing side channel [35]. The principle is that when two physical addresses (e.g., A and B) are located in the same bank, the time we alternatively access them is longer than the situation where they are in different banks due to row conflicts.

C. Online Faulting Phase

While Rowhammer-based faults can be injected into arbitrary parameters in the zk-SNARK scheme (e.g., the witness c_i, the evaluation of the selector polynomial $l_i(s)$), two requirements must be met for a successful Rowhammer fault attack. First, the fault must be traceable, meaning it can be identified by an exhaustive number of possible fault patterns. We conduct a cryptographic analysis and conclude that a single-bit fault of c_i is theoretically traceable. Second, the traceable parameter must be located in flippable memory. We employ the memory waylaying technique [36] to continuously evict pages within the page cache, ultimately positioning the targeted parameter at a flippable physical address.

Rowhammer Exploitation: Once the vulnerable bit is propagated to the flippable bits in DRAM, we begin by initializing two aggressor rows to hammer the victim row located in the middle. We employ a random pattern to initialize the aggressor rows. Specifically, we use random data to fill the adjacent aggressor rows, and then perform `Flush+Reload` on these rows repeatedly. This methodology allows us to trigger bit flips without relying on the knowledge of the victim row.

D. Post-processing Phase

In this phase, we recover the flipped witness bits by the faulty proofs. The error in the faulty proof is some multiples of Δc_i. Therefore, if we correct the faulty proof and make it satisfy Equation 8, we are able to find the position of the bit-flip. The main idea of this procedure is to correct the faulty proof by checking it using the proof verification algorithm. We explain it specifically in our bit tracing algorithm.

In Algorithm 1, we give public parameters in the scheme and the proof as inputs. The output is the position and value of the secret bit in the witness. After parsing the proof, we start enumerating the witness from c_0 to c_{m-1} and enumerating the position using bit_index. In lines 7-11, we compute Δ as a correction term the same way defined in Equation 8 and multiply it with the faulty part in the original proof. If the verification passes, the algorithm returns the desired output. Note that if the fault direction is from 0 to 1, we need to do a division and then do the verification as in lines 15-17.

Algorithm 1: The bit tracing algorithm

```
1  Input: Prover key PK_F; Verifier key VK_F and input
     u for the verifier; Faulty proof π' ; Max number of
     witness m
2  Output: (witness_index, bit_index, value) -
     Recovered secret bit
3  Parse the faulty proof as Equation 4 (Use π'[L] to
     denote the first term g^{L_mid(s)} in π')
4  for witness_index from 0 to m − 1 do
5      label witness_index as i
6      for bit_index from 1 to len(c_i) do
7          label bit_index as j
8          multiplier ← 2^{j−1}
9          Get g^{R(s)} from π, g^{l_i(s)} from PK_F
10         Compute Δ = e(g^{l_i(s)·multiplier}, g^{R(s)})
11         π'[L] = π'[L] · Δ
12         if Verify(VK_F, u, π) = true then
13             return (witness_index, bit_index, 1)
14         else
15             π'[L] = π'[L]/Δ
16             if Verify(VK_F, u, π) = true then
17                 return (witness_index, bit_index, 0)
18             end
19         end
20     end
21 end
```

V. Evaluation

A. Experimental setup

We perform the experiments using a machine with an Intel Core i3-10100 CPU (IceLake) and two Apacer DDR4-2666 8G DIMMs. The machine runs default Ubuntu 22.04 with Linux kernel 6.1.66. The zk-SNARK codes belong to the latest version of `libsnark`.

Victim implementation: We use the code of zk-SNARK scheme in `libsnark` and our example is the range proof, a popular application in blockchain. It proves that a value x is in a certain range $[0, 2^{32})$. In the blockchain scenario the value x may be the user's balance in the wallet. In the confidential setting, the value x must be kept secret to protect the user's privacy. Our attack goal is to leak x by ZK-Hammer.

B. Evaluation of Offline Preparing

To find nearby rows in the DRAM bank, we first obtain 2 MB of contiguous memory blocks by leveraging the deterministic behaviour of the buddy allocator. Using the contiguous memory chunk, we find the pages that are mapped into the same DRAM bank using a row buffer timing side channel. We repeatedly access different pages in the blocks and document access count and access time shown in Figure 4. The pages that are physically located on the same DRAM bank take longer to access due to row conflict.

Next, we use the double-sided Rowhammer technique for memory profiling, where each victim row is surrounded by an aggressor row from the top and bottom. Each hammering attempt consists of a finite loop targeting 4 pairs of virtual pages, with each pair being hammered for 250,000 rounds. After each hammering attempt, we scan the remaining unhammered memory for bit flips. This process allows us to collect

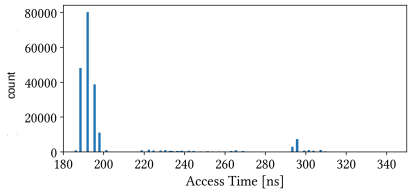
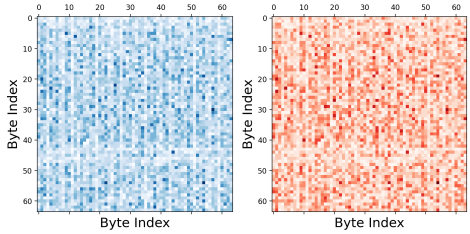


Fig. 4: Histogram of access times to the pages in the buffer.

a set of victim pages along with their corresponding aggressor pages. The page offset of a single-bit flip may vary among the collected victim pages. Figure 5 illustrates the distribution of bit-flip offsets across 4 KB-aligned pages.



(a) (b)

Fig. 5: Results of memory profiling. Bit flips from 1 to 0 (blue) and bit flips from 0 to 1 (red) accumulated over 4 KB pages.

C. Evaluation of Online Faulting

In the range-proof example, the secret value x is separated as a sequence of bits $x_0, ..., x_{31}$ because a proof of $x = \sum_{i=0}^{31} x_i \cdot 2^i$ and $x_i = 1$ or 0 yields the range of x. In this case, the vulnerable parameters $\{c_i\}_{i\in[0,31]}$ in the QAP instance are binary and we only need to fault one bit for each parameter. In the library, the prover first generates the QAP instance (a.k.a constraint system) and assigns values to all the c_i. The prover then calls a built-in function `r1cs_gg_ppzksnark_generator` to generate a key pair, prover key pk and verifier key vk. With pk, the prover can generate a proof. The time for our fault injection is between the value assignment of c_i and the finish of the proof. We measure the available faulting time which is 4245k CPU cycles. The time is not enough for a high-probability bit flip, and we compromise on the time to get 160 simulated faulty proofs. We argue that this will not decrease the severity of our attack as a small number of bits will significantly leak the privacy of users and break the zero-knowledge property of zk-SNARK.

D. Evaluation of Post-processing

We totally get 160 unique faulty proofs for 40 randomly chosen x in the domain. On average there are 4 faulty proofs for each x. Different from the fault attack of signature secret key [14] where only full key recovery is meaningful, we show that the 1 to 4-bit leakage of x reveals much information. In Figure 6, we compute the average of the percentage of the shrunk range with the 40 x's for each bit leak. We find that a 3 or 4-bit leak can shrink the guessing range of x by more than 70% for the attacker. A single-bit leak can also reveal nearly 20% information.

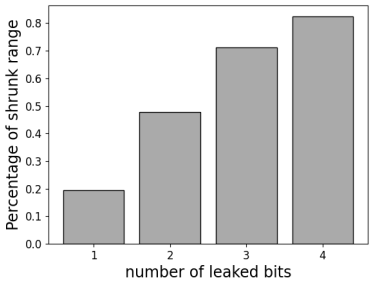


Fig. 6: Histogram of the attacking results. The ylabel represents the shrunk range percentage when an attacker guesses the value x after faulting.

VI. Countermeasures

In this section, we first discuss specific solutions to mitigate fault-injection attacks against zk-SNARK schemes. We then talk about general strategies to counteract Rowhammer.

A. Algorithmic Defenses

Verifying after proving: When a fault occurs in c_i, a faulty proof is generated. Given that the verifying algorithm can detect the fault by rejecting the faulty proof, the cryptographic library developer can append the verification step immediately after proving. However, this may incur significant overhead. In our example, this includes nine full pairing computations, slowing down the process by a factor of ten.

Redundancy check: The redundancy check includes both temporal and spatial checks [20]. The temporal redundancy check re-executes proofs and compares them. The spatial redundancy check involves storing multiple copies of secret parameters in random DRAM locations and comparing them after proving. Different results indicate a potential Rowhammer fault. These methods are generally more efficient but can be compromised.

B. General defenses

There are two categories of general countermeasures: software-only defenses [37], [38], which are compatible with existing commodity systems, and hardware-based defenses [39], which require hardware modifications. However, most of these defences have not been adopted by the industry and are not applicable to commodity systems and hardware.

VII. Conclusion and Future Work

In this paper, we presented the first novel fault attack against zk-SNARKs, coined as ZK-Hammer, that finds exploitable and traceable fault to a general category of zk-SNARK schemes and perform a post-Rowhammer analysis for secret recovery. To validate the vulnerability, we perform Rowhammer evaluations against the zk-SNARK codes in `libsnark`. Future work involves fault analysis of other categories of zk-SNARKs and more effective algorithmic countermeasures.

REFERENCES

[1] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, 2019, pp. 203–225.

[2] S. Bowe, A. Chiesa, M. Green, I. Miers, P. Mishra, and H. Wu, "Zexe: Enabling decentralized private computation," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 947–964.

[3] Z. Wan, Y. Zhou, and K. Ren, "Zk-authfeed: Protecting data feed to smart contracts with authenticated zero knowledge proof," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1335–1347, 2022.

[4] T. Liu, X. Xie, and Y. Zhang, "Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2968–2985.

[5] D. Beaver, "Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority," *Journal of Cryptology*, vol. 4, pp. 75–122, 1991.

[6] M. Chase, D. Derler, S. Goldfeder, J. Katz, V. Kolesnikov, C. Orlandi, S. Ramacher, C. Rechberger, D. Slamanig, X. Wang *et al.*, "The picnic signature scheme," *Submission to NIST Post-Quantum Cryptography project*, 2020.

[7] "libsnark," Github https://github.com/scipr-lab/libsnark, 2014.

[8] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 6, pp. 4733–4751, 2022.

[9] "Zokrates/zokrates," Github https://github.com/Zokrates/ZoKrates, 2021.

[10] Z. Zhang, D. Chen, J. Qi, Y. Cheng, S. Jiang, Y. Lin, Y. Gao, S. Nepal, Y. Zou, J. Zhang *et al.*, "Sok: Rowhammer on commodity operating systems," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 436–452.

[11] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors," in *International Symposium on Computer Architecture*, 2014, pp. 361–372.

[12] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss, "Half-double: Hammering from the next row over," in *USENIX Security Symposium*, 2022.

[13] H. Hassan, Y. C. Tugrul, J. S. Kim, V. Van der Veen, K. Razavi, and O. Mutlu, "Uncovering in-dram rowhammer protection mechanisms: A new methodology, custom rowhammer patterns, and implications," in *IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1198–1213.

[14] K. Mus, Y. Doröz, M. C. Tol, K. Rahman, and B. Sunar, "Jolt: Recovering tls signing keys via rowhammer faults," in *IEEE Symposium on Security and Privacy*. IEEE, 2023, pp. 1719–1736.

[15] K. Ryan, "Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 181–194.

[16] S. Weiser, D. Schrammel, L. Bodner, and R. Spreitzer, "Big numbers-big troubles: Systematically analyzing nonce leakage in ({EC) DSA} implementations," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1767–1784.

[17] K. Ryan, "Return of the hidden number problem.: A widespread and novel key extraction attack on ecdsa and dsa," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 146–168, 2019.

[18] D. Poddebniak, J. Somorovsky, S. Schinzel, M. Lochter, and P. Rösler, "Attacking deterministic signature schemes using fault attacks," in *IEEE European Symposium on Security and Privacy*, 2018, pp. 338–352.

[19] K. Mus, S. Islam, and B. Sunar, "Quantumhammer: a practical hybrid attack on the luov signature scheme," in *ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1071–1084.

[20] S. Islam, K. Mus, R. Singh, P. Schaumont, and B. Sunar, "Signature correction attack on dilithium signature scheme," in *IEEE European Symposium on Security and Privacy*, 2022, pp. 647–663.

[21] S. Bhattacharya and D. Mukhopadhyay, "Curious case of rowhammer: flipping secret exponent bits using timing analysis," in *Cryptographic Hardware and Embedded Systems*, 2016, pp. 602–624.

[22] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020.

[23] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Hammering a needle in the software stack," in *USENIX Security Symposium*, 2016, pp. 1–18.

[24] M. Fahr Jr, H. Kippen, A. Kwong, T. Dang, J. Lichtinger, D. Dachman-Soled, D. Genkin, A. Nelson, R. Perlner, A. Yerukhimovich *et al.*, "When frodo flips: End-to-end key recovery on frodokem via rowhammer," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 979–993.

[25] P. Mondal, S. Kundu, S. Bhattacharya, A. Karmakar, and I. Verbauwhede, "A practical key-recovery attack on lwe-based key-encapsulation mechanism schemes using rowhammer," in *International Conference on Applied Cryptography and Network Security*. Springer, 2024, pp. 271–300.

[26] S. Amer, Y. Wang, H. Kippen, T. Dang, D. Genkin, A. Kwong, A. Nelson, and A. Yerukhimovich, "Pq-hammer: End-to-end key recovery attacks on post-quantum cryptography using rowhammer," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 48–48.

[27] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, 2013, pp. 626–645.

[28] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno, "Pinocchio coin: building zerocoin from a succinct pairing-based proof system," in *Proceedings of the First ACM workshop on Language support for privacy-enhancing technologies*, 2013, pp. 27–30.

[29] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*, 2016, pp. 305–326.

[30] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct {Non-Interactive} zero knowledge for a von neumann architecture," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 781–796.

[31] A. Chiesa, E. Tromer, and M. Virza, "Cluster computing in zero knowledge," in *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II 34*, 2015, pp. 371–403.

[32] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 253–270.

[33] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," *Communications of the ACM*, vol. 59, no. 2, pp. 103–112, 2016.

[34] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *IEEE Symposium on Security and Privacy*, 2020.

[35] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "{DRAMA}: Exploiting {DRAM} addressing for {Cross-CPU} attacks," in *25th USENIX security symposium (USENIX security 16)*, 2016, pp. 565–581.

[36] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another flip in the wall of rowhammer defenses," in *IEEE Symposium on Security and Privacy*, 2018, pp. 245–261.

[37] C. Gongye, Y. Luo, X. Xu, and Y. Fei, "Hammerdodger: a lightweight defense framework against rowhammer attack on dnns," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[38] J. Jung, H. So, W. Ko, S. S. Joshi, Y. Kim, Y. Ko, A. Shrivastava, and K. Lee, "Maintaining sanity: Algorithm-based comprehensive fault tolerance for cnns," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.

[39] N. Mishra, R. A. Mool, A. Chakraborty, and D. Mukhopadhyay, "Plug your volt: Protecting intel processors against dynamic voltage frequency scaling based fault attacks," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.