ExplFrame: Exploiting Page Frame Cache for Fault Analysis of Block Ciphers

Anirban Chakraborty
Indian Institute of Technology
Kharagpur, India
anirban.chakraborty@iitkgp.ac.in

Sarani Bhattacharya
Indian Institute of Technology
Kharagpur, India
tinni1989@gmail.com

Sayandeep Saha
Indian Institute of Technology
Kharagpur, India
sahasayandeep@cse.iitkgp.ac.in

Debdeep Mukhopadhyay
Indian Institute of Technology
Kharagpur, India
debdeep@cse.iitkgp.ac.in

Abstract—**Page Frame Cache (PFC) is a purely software cache, present in modern Linux based operating systems (OS), which stores the page frames that are recently being released by the processes running on a particular CPU. In this paper,** *we show that the page frame cache can be maliciously exploited by an adversary to steer the pages of a victim process to some pre-decided attacker-chosen locations in the memory*. **We practically demonstrate an end-to-end attack,** *ExplFrame*, **where an attacker having only user-level privilege is able to force a victim process's memory pages to vulnerable locations in DRAM and deterministically conduct Rowhammer to induce faults. We further show that these faults can be exploited for extracting the secret key of table-based block cipher implementations. As a case study, we perform a full-key recovery on OpenSSL AES by Rowhammer-induced single bit faults in the T-tables. We propose an improvised fault analysis technique which can exploit any Rowhammer-induced bit-flips in the AES T-tables. To the best of our knowledge, this is the first work highlighting the vulnerabilities of PFC and fault analysis of block cipher using Rowhammer completely from user-space.**

Index Terms—**Page Frame Cache, Buddy Allocator, OpenSSL, Rowhammer, DRAM, Fault Analysis, ECC**

I. Introduction

Modern operating systems (OS) are optimized to obtain the best possible performance and throughput on a given hardware architecture. The memory allocation mechanism of OS plays a crucial role in determining the overall performance of a system. The efficiency of this OS subsystem is mainly attributed to its intelligent usage of *caching*, which helps in taking advantage of the locality of reference (temporal and spacial) in the memory hierarchy.

Memory allocation subsystems in modern Linux-based OS use *Buddy Allocation scheme* to allocate memory pages to different processes. When a process requests for memory, the buddy allocator allocates the required amount of memory in the form of fixed sized *page frames*. As the process terminates, the allocated page frames are added back to the memory pool. To boost memory performance, the kernel maintains a per-CPU *page frame cache* (PFC) which is a small software cache storing recently de-allocated page frames. Upon the arrival of a new request, the page frames inside the PFC are the first to serve it, before going to the actual memory pool. Moreover, this allocation scheme is oblivious to the processes and, in practice, the pages left by one process (in PFC) can readily be re-allocated to another process.

The aim of this work is to maliciously exploit the aforementioned, seemingly benign, memory caching policy defined in the buddy allocator. We exploit the fact that the allocation of pages from PFC does not take the identity of the processes into account. Using this property of the PFC, we show that an adversary, having only user privilege, can indirectly steer the pages of a victim process to some pre-determined memory locations inside the Dynamic Random Access Memory (DRAM). Restricting a victim process to operate in some attacker-controlled memory locations may have severe security implications. Here we show that even mathematically robust cryptosystems can fall prey to this vulnerability.

One of the most prominent DRAM vulnerabilities known till date is the *Rowhammer bug* [1]. It is a phenomenon observed in most of the modern commercial DRAM modules where repeated access to a particular row induces bit flips in one of the adjacent rows. However, inducing precise faults using Rowhammer is a challenge due to the uncontrollability of flip locations which is specific to a DRAM instance. In practice, some of the rows might show higher chances of getting faulted than others. However, if the pages of a victim process get assigned to a Rowhammer vulnerable location, faults can be induced in the process in a regular manner. Quite obviously, PFC becomes a nice tool in this context as it can force the pages of a victim to some attacker-decided memory regions. Putting it differently, Rowhammer provides a concrete use case for showing the exploitability of the PFC allocation scheme. In this paper we present an end-to-end practical realization of the aforementioned idea of combining PFC with Rowhammer. The proposed attack strategy, called *ExplFrame*, has been utilized to launch a practical key recovery attack on the T-table-based AES implementation from OpenSSL 1.1.1 [2].

Previous work in [3]–[5] have exploited Linux's Buddy Allocation system to perform *memory massaging* for conducting Rowhammer. In particular, they exhaust the memory pages during templating phase, which is dependent on the allocation policy of the Buddy Allocator. *Our proposed ExplFrame does not rely on the allocation policy; rather we exploit the principle of caching in one of the components of memory allocation subsystem, the PFC. Most importantly, the exploitation of PFC, to the best of our knowledge, has never been used as an attack vector before.* It is worth mentioning that table-based AES implementations have previously been

targeted with Rowhammer-induced faults in [6]. The main idea is to corrupt an entry inside the T-Table and thereby create a statistical bias within AES state, which can be used for key recovery. However, one immediate advantage of doing the fault attack with ExplFrame strategy is that it can be done from user privilege level. In contrast, the attack proposed in [6] uses `pagemap` which requires administrative privilege on modern Linux distributions. Moreover, we observed that the strategy of fault exploitation in [6] (called Persistent Fault Attack or PFA), is limited by the fact that it can only recover the key if certain specific bits of the T-table get affected by fault injection [1] This being practically infeasible, the original PFA proposal cannot be used in the present context. Hence, as a second contribution, we propose a general fault exploitation methodology called Deep Round Persistent Fault Attack (DRPFA), which can exploit any Rowhammer-induced bit flip within the T-table. Further, we comment that Error Correcting Codes (ECC), which are often considered as effective countermeasures against Rowhammer induced faults [1], are not sufficient for preventing exploitable information leakage. We present a brief discussion on this at the end of this paper.

The rest of the paper is organized as follows. We present a brief background on the memory allocation schemes and Rowhammer in Sec. II, followed by an overview of our attack ExplFrame in Sec. III. In Sec. IV, we demonstrate an end-to-end fault induction method on the T-tables of OpenSSL AES using ExplFrame. We further provide an improved key recovery algorithm with the induced faults in Sec. V and experimental results in Sec. VI [2]. The applicability of the attack at different scenarios has been discussed in Sec. VII. Finally, we conclude in Sec. VIII.

II. BACKGROUND

A. Linux memory allocation subsystem

In NUMA (Non-Uniform Memory Access) based OS, each *node* [3] is divided into a number of blocks called *zone*. Inside each memory zone, the allocation process is handled by the core allocator for Linux, called *Buddy allocator*. In this allocation scheme, the pages are clustered into large blocks of size in power of two. When a request for certain amount of memory comes from the processor, the algorithm first searches the blocks of pages to check if the request can be met. If no blocks of pages are found to meet the demand, block of the next size is split into half and one half is allocated to the requesting process. The two smaller blocks thus produced are called buddies to each other. The process of splitting a block into half continues until a block of desired size is obtained. Likewise, when the allocated block becomes free, the buddy block is also examined. If both the blocks are free, they are merged together and returned back to their original block size.

The coalescing of blocks on de-allocation gives rise to the name of the allocation scheme.

The OS maintains a *page frame cache* [4] for each memory zone. This small software cache of recently de-allocated (released) page frames are used by the Buddy allocator if the local CPU requests a small amount of memory, typically a few pages. The presence of page frame cache can significantly boost up the system performance by taking advantage of the locality of reference. The OS kernel keeps track of two watermarks to monitor the size of the cache. Whenever the number of page frames in the cache falls below the low watermark, the kernel brings in more page frames from the buddy system. Similarly, when the number of page frame surpasses the high watermark due to release of page frames from different running and finished processes, the kernel releases some of the page frames back to the buddy system. *It is worth mentioning that our attack exploits the caching mechanism of the allocator and is independent of the allocation policy itself.*

B. The Rowhammer bug

DRAMs have been constantly scaled down to accommodate larger number of memory cells into smaller physical space, thereby reducing the cost-per-bit of memory. However, cramming a large number of DRAM cells in small space leads to electromagnetic coupling effects among themselves. Owing to its closely packed architecture, when a particular DRAM row is accessed consistently and in high frequency, the cells in the neighbouring rows tend to lose their charge, thereby inducing bit-flips. This phenomenon is termed as Rowhammer bug, which has been exploited to launch several devastating classes of attacks in recent past [3]–[5].

The driving force behind Rowhammer bug is that specific DRAM rows must be repeatedly activated fast enough such that the adjacent rows lose charge. However to achieve this, the content of the cache memory must be flushed after every access so that all the requests are served from the main memory. In x86 architecture, flushing of cache can be achieved from userspace by the `clflush` command. This fact indicates that Rowhammer on standard x86-64 machines does not require any special privilege for repeatedly activating DRAM rows. However, in practice, only certain specific regions in a DRAM chip are found to be vulnerable under Rowhammer, and it is purely driven by the device physics. In order to practically induce faults, the target data must be located on a Rowhammer-vulnerable location in DRAM [5]. Hence, from an attacker's perspective, exploiting the Rowhammer bug is not trivial and creating a deterministic exploit requires certain other vulnerabilities to be combined with this one.

III. THE EXPLFRAME APPROACH

Continuing our discussion from Linux memory subsystem, in this section we introduce the security implication of such

---

[1]The reason for this will be explained later in this paper.

[2]We have informed and shared our findings with Intel Product Security Incident Response Team.

[3]NUMA systems classifies memory into nodes. Each node has similar access characteristics and affinity to one processor.

[4]Not to be confused with *page cache* which contains files read from the disk, memory-mapped files, shared libraries, etc.

[5]Once the vulnerable locations are identified, inducing bit-flips in them is repeatable.

performance improvisation scheme in details and propose ExplFrame, which utilizes PFC for performing Rowhammer almost deterministically on a victim process.

A. Exploiting Page Frame Cache

Page Frame Cache stores the recently freed pages from all processes running on a particular processor core. The primary intention of PFC is to boost the performance of the memory allocation subsystem by keeping recently used page frames close to the processor, in case the process requests for additional memory in near future. However, the security implications of this simple scheme has never been analyzed in literature, and we explore in this work for the first time.

How can one exploit the PFC?

Let us consider the following example:

- Process A is running on a standard system and requests some amount of memory (say by using `mmap` with the `MAP_POPULATE` flag). On such instance, the buddy allocator scans the list of available page frame blocks and finds out a block that can satisfy the request.
- In course of time, the process A de-allocates a page (say by using the function `unmmap`). The 'unmapped' page resides in the page frame cache of the zone in the anticipation that it could be requested by the same process A in recent future.

Thus if the same program requests for additional memory during the course of its execution, the OS attempts to serve the page frames from the cache. If the request is small, then it is satisfied by the cache; else, it will invoke the buddy allocator once more. The situation becomes interesting to a security engineer when there is another process (Process B) running simultaneously on the system and sharing the same CPU. Consider, process A is an adversary and process B is oblivious of such adversary.

- Once again we have Process A running on the system which allocates some memory, unmaps one or two pages and waits[6].
- In this scenario, Process B sends a request for additional memory pages. The OS will first try to service the request from the page frame cache itself. Thus, there is a high probability that the page frame that was unmapped by the adversarial process gets allocated to the victim.

Therefore, PFC can be exploited by an adversary to control the memory allocation of another process. More precisely, an adversary can restrict the physical memory locations that a legitimate process can use. As a practical implementation of our claim, we present ExplFrame, which uses PFC to steer a victim process's sensitive data into Rowhammer-vulnerable locations and subsequently induce faults using Rowhammer.

B. Threat Model

We assume a multi-user server environment running on a Linux-based OS, where the adversary could introduce precise faults on victim's sensitive data. The adversary has user-level privileges. It is further assumed that the adversary cannot access any security sensitive memory possessed by the victim. This exploit requires that the victim and the adversary are operating on the same processor core [7].

C. Outline of the attack

The adversary performs the following steps in order

- She performs *bin-partitioning* to partition the allocated memory space into bins to identify the individual DRAM banks. After the partitioning is complete, she conducts Rowhammer on one of the bins to find out a vulnerable page. We provide a detailed representation of bin-partitioning process in the next section.
- She unmaps the vulnerable page and due to the presence of PFC, the unmapped page gets cached in it. As discussed in previous subsection, the unmapped page stays in the PFC until some process requests for memory.
- She waits until a victim process requests for some memory. Due to the property of PFC, the pages in the cache are the first one to get allocated to the victim. Once the vulnerable page is allocated to the victim process, the adversary starts rowhammering once again in the same bin (which also contains the vulnerable page).

We present a detailed description of all the aforementioned steps in a practical setting in the next section.

IV. ATTACKING AES T-TABLES: A CASE STUDY

ExplFrame is a generic attack which exploits the vulnerabilities of PFC to deterministically conduct Rowhammer on victim's data. In this section we present a practical end-to-end attack on OpenSSL AES using ExplFrame to induce faults in T-tables. In the next subsection, we present a novel memory partitioning algorithm in order to utilize rowhammer in a nearly deterministic way.

A. Deterministic Rowhammer from user-space

One of the major challenges for precise Rowhammering on a particular location is that the physical layout of memory is abstracted by the OS. Also, modern Linux kernel does not allow access to `pagemap` from user privilege. Therefore, in order to perform Rowhammer from userspace on a specific memory page, we need to determine the DRAM bank where the page is located and repeatedly access (hammer) the addresses on that particular bank.

Previous works [7] have shown that when a pair of addresses are accessed simultaneously, it creates a measurable timing channel depending on whether the address belong to different

---

[6]The adversarial process (Process A) must remain active rather than going into inactive state (sleep), since in that case the entire process state information including page frame cache will be swapped out of memory.

[7]Previous attack in [4] also assumes that the attacker and the victim are operating on the same core and binds the processes to the same CPU core using `taskset`.
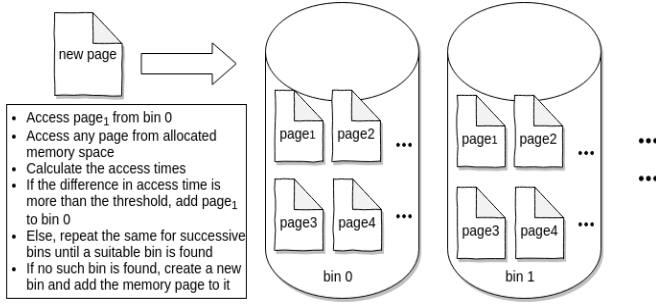
Fig. 1: An overview of the bin partitioning process

banks or same bank but different rows within the DRAM [8]. Based on the DRAM access timing side-channel, we present a novel *bin partitioning* technique to partition the entire allocated address space into the hypothetical 'bins' such that each bin corresponds to a DRAM bank.

An overview of the bin partitioning process is shown in Figure 1. We first access the first page and put it in bin_0. Next, we access the next page and the first page simultaneously and check their access times. If the access time for the second page is more than some pre-defined threshold (can be determined by initial profiling of the system), that would mean the pair of accesses have resulted in a row conflict. So, the pair of page frames must be located in the same bank but different row. In that case, we put the second page in the same bin as the first one, i.e, bin_0. Whereas, if the access time is less than the threshold, we put it in the next bin, i.e, bin_1. Similarly, we pick the next page and check its access time with respect to the pages already stored in the bins. Based on the timing value, we put the page into one of the bins. This process is repeated until all the allocated pages are exhausted. In the end, all the page frames will be partitioned into separate bins.

B. Putting it all together

We allocate a large memory (1 GB in our case) and partition the entire available memory into n bins (n = 16 for the DRAM that we targeted) using *bin partitioning* method, where each bin corresponds to a DRAM bank. After partitioning of the memory space, we start conducting Rowhammer by randomly picking addresses stored in last bin [9]. The hammering is done for a stipulated time (1 hour in our case) and if no fault is found then we move on to the preceding bin. This process continues until a flip is found. If no such flip is found in any bin, the entire process is killed and restarted once again. If a flip is found, the corresponding page is unmapped.

We target the encryption T-tables ($T0$ through $T3$) of AES of OpenSSL 1.1.1 in a standard multi-user environment. Now, the adversary waits for the victim to load the T-tables into

---

[8]If the addresses belong to the same bank but different row, then it will create a *row conflict*. The first access will bring the data into the row buffer while at the time of second access, the first row will be closed first and then the second one is fetched. Due to row conflict, the difference in access time will be much higher than the other cases.

[9]Statistically, the last bin will have the smallest number of mismatches after two pass of the algorithm.
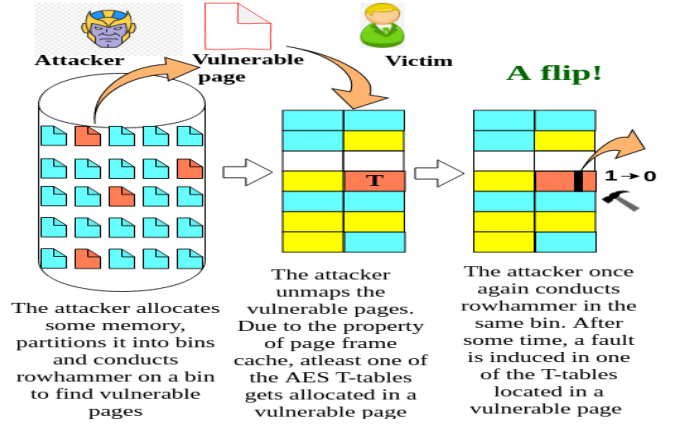

Fig. 2: An overview of ExplFrame on OpenSSL AES T-tables

Processor	Micro-architecture	DRAM type & capacity	Operating System	Kernel version
Intel i5 3330	IvyBridge	Hynix DDR3 4GB	Ubuntu 14.04	4.13.0-36 generic
Intel i7 7700	KabyLake	Micron DDR4 8GB	Ubuntu 18.04	4.15.0-50 generic

TABLE I: Experimental Setup

the memory. Due to the presence of PFC in each memory zone, the T-tables will be allocated in the same vulnerable page which was unmapped earlier. Once one of the T-tables is placed in a freed vulnerable page, the adversary again starts Rowhammering on the same bin. Due to reproducibility of the fault, the same page frame which now contains the T-table gets faulted once again, thereby corrupting the particular entry of the T-table. We validated our experiments on two standard Desktop computers having the specifications as mentioned in Table I. A pictorial representation of our ExplFrame attack on AES T-tables is depicted in Figure 2.

V. PRACTICAL EXPLOITATION OF INDUCED FAULTS

In this section, we present a generic and practical methodology for exploiting the faults induced by Rowhammering. Given that the use-case of ours induces faults in AES T-tables, we aim to perform a key recovery attack on AES by analyzing these faults. There exist a large body of work addressing fault attacks (FA) on different classes of cryptographic primitives, especially on block ciphers like AES [8]. However, the attack algorithms vary largely depending on the type of the faults that can be practically induced within a system. One should note that the corruption in the present case happens in the T-tables of the implementation. The injected fault persists in the T-table until it is reloaded. This typical fault model matches with the one proposed by Zhang et. al. in [6] (popularly referred to as Persistent Fault Model (PFM)). The main idea of PFA is to exploit the statistical bias resulting from the AES computation with a corrupted T-table. The faulty outcomes (ciphertexts) are analyzed statistically by guessing the last round key candidates. The statistical bias becomes visible only for the correct key guess eventually returning the key. The original proposal in [6] makes two important assumptions:

1) The adversary exactly knows the value that got corrupted inside the T-table.
2) The corrupted entry in T-table is accessed at the last round of AES computation. This usually happens with a reasonable probability.

Several issues may arise while realizing the PFA from a practical perspective using Rowhammer. Below we enlist the main realization issues:

1) **The Target Implementation:** The implementation under attack plays a crucial role for the success of PFA with Rowhammer induced faults. T-table-based implementations are available at state-of-the-art crypto-cores like OpenSSL, Libgcrypt etc. However, intricate implementation differences may still be there. As an example, we consider two competing implementations from OpenSSL and Libgcrypt. The first one utilizes 4 T-tables T_0, T_1, T_2 and T_3 for all the rounds r_i ($1 \leq r_i \leq 10$). In contrast, Libgcrypt utilizes 4 alternative T-tables T_0', T_1', T_2' and T_3' to realize the last round of AES.

2) **Nature of Rowhammer-induced Faults:** Rowhammer fault injection strategy provides limited control over the faults to be induced. More specifically, *it is hard to control which bit of the T-table gets corrupted.*

In [6], Zhang et. al. performed a case study on Libgcrypt implementation which uses separate T-tables for the last round. As a result, faults induced in their experiments only affect the last round. As a more generic scenario, we consider the OpenSSL implementation for the illustration which uses the same T-tables for realizing all the AES rounds. Some part of the high-level C-code for the last round in OpenSSL is depicted in Fig. 3. One should observe that *in order to undo the effect of the AES MixColumns sub-operation, some of the bytes in a T-table output are masked (i.e. ANDed with zero).*

```
s0 =
    (Te2[(t0 >> 24)       ] & 0xff000000) ^
    (Te3[(t1 >> 16) & 0xff] & 0x00ff0000) ^
    (Te0[(t2 >>  8) & 0xff] & 0x0000ff00) ^
    (Te1[(t3      ) & 0xff] & 0x000000ff) ^
    rk[0];
PUTU32(out      , s0);
:
:
s3 =
    (Te2[(t3 >> 24)       ] & 0xff000000) ^
    (Te3[(t0 >> 16) & 0xff] & 0x00ff0000) ^
    (Te0[(t1 >>  8) & 0xff] & 0x0000ff00) ^
    (Te1[(t2      ) & 0xff] & 0x000000ff) ^
    rk[3];
PUTU32(out + 12, s3);
```

Fig. 3: Code Snippet for AES Last Round in OpenSSL.

We next point out why these two above-mentioned realization issues are important from a practical perspective. Our first observation is that *the original PFA algorithm does not work in this context with Rowhammer induced faults on OpenSSL AES implementation.* To elaborate this observation, we point out that *three out of four bytes in each T-table output are masked in the final round computation of AES (in order to undo the effect of the MixColumns operation).* Now if the Rowhammer fault affects any of these masked byte locations, the fault effect will also get masked and would not propagate to the output. As the attacker cannot precisely control which bit of the T-table gets corrupted, the PFA attack targeting the last round is suppose to fail with a reasonably high probability. In nutshell, the Rowhammer induced faults cannot be utilized with the PFA algorithm described in [6] for many of the practical implementations like the one in OpenSSL.

Fortunately, our investigation revealed that the faults induced by Rowhammering are still exploitable for key extraction. In this context we propose a novel and generic attack strategy called Deep Round Persistent Fault Attack (DRPFA) which is not affected by the fact that the attacker may not have precise control over the location of the Rowhammer induced faults. One should recall that the fault in the T-table is persistent, and as a result it also affects certain intermediate rounds of the AES computation. The proposed DRPFA attack exploits these corruptions in intermediate rounds for extracting the key. The advantage of this strategy is that we need not care about the precise location of the faulty bit anymore. The details of the DRPFA attack is presented in the following subsection.

A. Deep Round Persistent Fault Attack

Let us now describe the DRPFA attack algorithm. The pseudo-code for the algorithm is depicted in Algorithm 1. The main idea is to guess a part of the secret and partially decrypt the ciphertexts up to the round where the statistical bias is being observed. More precisely, the partial decryption should continue up to the inverse MixColumns of the target round so that the bias at the S-Box outputs can be exploited. According to the well-known wrong-key assumption, the aforementioned statistical bias becomes visible for the correct key guess with a very high probability, and for the wrong guesses with negligibly small probability. This fact enables the recovery of the key with a fairly simple statistical test. We utilize the Squared-Euclidean-Imbalance (SEI) test for identifying the bias and thus the correct key. Also, in order to reduce the complexity, our attack mainly targets the 9th round of the AES computation. One important observation regarding DRPFA is that, the attack remains equally applicable even in the presence of combined Side-Channel Analysis (SCA) and Fault Attack (FA) countermeasures. From this perspective, DRPFA is equally powerful as of SIFA [9] and PFA [10].

VI. EXPERIMENTAL VALIDATION

The results of inducing faults in OpenSSL AES T-tables is shown in Table II. In order to validate the practicality of DRPFA with these faults, we encrypt 20000 plaintexts after the fault is induced. Here the fault has been induced in the table T0, and is present throughout the encryption campaign. The 9th round S-Box output is considered for attack. Consequently, the key is extracted in chunks of 32-bits (i.e. $s = 32$ according to Algorithm. 1. It is worth mentioning that the PFA attack did not work with this specific fault even with 100000 ciphertexts.

Algorithm 1: The DRPFA Algorithm

Input: Ciphertexts from Rowhammer fault injection campaign ($\mathcal{C}$), target round r.
Output: Key k_c

1 set s := size of the partial key guess based on r
2 set $K_g := \{0, 1, \cdots, 2^s - 1\}$
3 set $SEI_{dict} := \emptyset$
4 **for** $k_g \in K_g$ **do**
5   set $S_{k_g} := \emptyset$
6   **for** $c \in \mathcal{C}$ **do**
7       $S_{k_g} := S_{k_g} \cup \texttt{Partial\_Decrypt}(c, r)$
8   **end**
9     $SEI_{dict}[k_g] := \texttt{SEI}(S_{k_g})$
10 **end**
11 set $k_c := \underset{k_g \in K_g}{\operatorname{argmax}} \; SEI_{dict}[k_g]$
12 **return** k_c

However, DRPFA successfully extracts the key with roughly 8000 ciphertexts. In order to elaborate the relation of the attack

No.	Time (mins)	T-table index	Value before flip	Value after flip
1	1035	Te1[139]	477a 3d3d	47fa 3d3d
2	538	Te0[25]	b3d4 d467	a3d4 d467
3	224	Te1[254]	d66d bbbb	c66d bbbb
4	3623	Te1[38]	ae3d 9393	ae3d 9193
5	12	Te3[87]	cbcb 468d	cbcb 460d
6	105	Te3[148]	2222 6644	0222 6644
7	256	Te1[88]	bed4 6a6a	bed4 2a6a
8	67	Te1[193]	88f0 7878	88f0 7868

TABLE II: Faults induced by ExplFrame on AES T-tables

with the number of ciphertexts, we present a convergence plot in Fig. 4. The blue region in this plot presents the SEI values
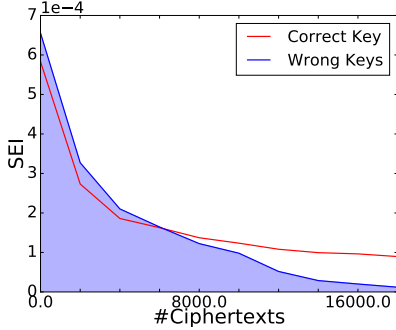

Fig. 4: SEI convergence plot for the DRPFA attack.

for wrong key guesses. The red line, on the other hand, refers to the SEI value corresponding to the correct key. For the attack to be successful, the red line should remain over the blue region. One may observe that in our experiment, the red line crosses the blue region near 7000 ciphertexts and the separation becomes prominent with 8000 ciphertexts. This clearly establishes the efficacy of the DRPFA.

VII. DISCUSSION

The attack use-case presented in the last section assumes that the AES is implemented by means of T-tables. Certain modern AES implementations use dedicated hardware accelerators, such as AES-NI [11] instead of T-tables. However, this fact does not rule out the efficacy of the proposed attack

strategy as it is not specific to AES. In contrast, the hardware accelerators available in modern processors provide support for AES only. Moreover, there exist cryptographic primitives such as pseudorandom number generators which still utilize the T-tables in OpenSSL implementation [12]. The proposed attack thus remains completely relevant for modern systems.

Error Correcting Codes (ECC) are often considered as effective countermeasures against Rowhammer faults [1]. However, we claim that a straightforward application of ECC as in [1] cannot prevent information leakage by means of faults. This claim is based on the observation [13] that the error correction operation takes considerably large number of clock cycles with respect to the normal operation [4]. It is thus feasible to detect and pinpoint when error correction takes place during the encryption (with a timing channel) by using some of the strategies described in [14], [15]. Interestingly, if the attacker possess the knowledge of the exact corrupted entry within the faulted T-table [10], she can figure out the input to the table. Figuring out the table input is equivalent to the key recovery as the secret intermediate states of a block cipher get exposed by this mean. Moreover, this *timing-assisted fault analysis* strategy does not require any access to the ciphertexts of the encryption block which enhances its scope for certain other classes of crypto-primitives. Future work will present a practical realization of this attack on state-of-the-art systems.

VIII. CONCLUSION

In this paper, we presented ExplFrame, a novel attack technique that combines the vulnerability of page frame cache with Rowhammer to induce faults in victim process's data, entirely from user space. We highlighted how PFC can be used as an attack vector to restrict a process to attacker-chosen locations in DRAM. To validate the practicality of our claims, we demonstrated an end-to-end attack on OpenSSL AES to induce faults in T-tables. We also presented an improvised Fault Analysis technique to extract the key from the faulty ciphertexts, created due to faulted T-tables.

REFERENCES

[1] Y. Kim and et. al., "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ISCA '14*. IEEE Press, 2014, pp. 361–372.
[2] OpenSSL, "Openssl cryptography and ssl/tls toolkit," 2019, [Online; accessed 25-June-2019].
[3] V. van der Veen and et. al., "Drammer: Deterministic rowhammer attacks on mobile platforms," in *CCS '16*. ACM, 2016, pp. 1675–1689.
[4] A. Kwong and et. al., "Rambleed: Reading bits in memory without accessing them," in *S&P '20*. IEEE Computer Society, 2020.
[5] V. van der Veen and et. al., "Guardion: Practical mitigation of dma-based rowhammer attacks on arm," in *DIMVA 18*. Springer International Publishing, 2018, pp. 92–113.
[6] F. Zhang and et. al., "Persistent fault analysis on block ciphers," in *IACR TCHES*, 2018, pp. 150–172.
[7] P. Pessl and et. al., "Drama: Exploiting dram addressing for cross-cpu attacks," in *USENIX Security 16*. USENIX Association, 2016, pp. 565–581.
[8] M. Tunstall, D. Mukhopadhyay, and S. Ali, "Differential fault analysis of the advanced encryption standard using a single fault," in *WISTP 2011*. Springer Berlin Heidelberg, 2011, pp. 224–233.

[10] which is reasonable if the T-table is kept in a shared memory, and attacker have a read access to it.

[9] C. Dobraunig and et. al., "Statistical ineffective fault attacks on masked aes with fault countermeasures," in *ASIACRYPT 2018.* Springer International Publishing, 2018, pp. 315–342.

[10] J. Pan and et. al., "One fault is all it needs: Breaking higher-order masking with persistent fault analysis," in *DATE 19*, 2019, pp. 1–6.

[11] K. Akdemir and et. al., "Breakthrough AES performance with Intel AES New Instruction," Intel, Tech. Rep., April 2004.

[12] S. Cohney and et. al., "Pseudorandom Black Swans: Cache attacks on CTR_DRBG," 2019.

[13] L. Cojocara and et. al., "Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks," in *S&P 19.* IEEE Computer Society, 2019.

[14] D. Gullasch and et. al., "Cache games – bringing access-based cache attacks on aes to practice," in *SP '11.* IEEE Computer Society, 2011, pp. 490–505.

[15] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security 14).* USENIX Association, 2014, pp. 719–732.