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

Anirban Chakraborty
Dept. of Computer Science
IIT Kharagpur, India
anirban.chakraborty
@iitkgp.ac.in

Sarani Bhattacharya
COSIC, ESAT
KU Leuven, Belgium
Sarani.Bhattacharya
@esat.kuleuven.be

Sayandeep Saha
Dept. of Computer Science
IIT Kharagpur, India
sahasayandeep
@cse.iitkgp.ac.in

Debdeep Mukhopadhyay
Dept. of Computer Science
IIT 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 were recently 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. As a case study, we induce single bit faults in the T-tables on OpenSSL (v1.1.1) AES using our proposed attack ExplFrame. We also propose an improvised fault analysis technique which can exploit any Rowhammer-induced bit-flips in the AES T-tables.**

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

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 induce exploitable faults on the T-table-based AES implementation from OpenSSL 1.1.1 [2]. It is worth mentioning that *our proposed ExplFrame does not rely on the memory 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.*

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 with experimental results. We further provide a discussion on the analysis of induced faults in Sec. V.

978-3-9819263-4-7/DATE20/©2020 EDAA

II. Background

A. Linux memory allocation subsystem

In NUMA (Non-Uniform Memory Access) based OS, each *node* [1] 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 OS maintains a *page frame cache* [2] 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.

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], [4].

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 [3]. 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.

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

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

III. The ExplFrame Approach

Continuing our discussion from Linux memory allocation subsystem, in this section we discuss about the security implication of such performance improvisation scheme and introduce our attack ExplFrame.

A. Exploiting Page Frame Cache

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 PFC; else, it will invoke the buddy allocator once more. The situation becomes interesting 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[4]. 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

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

[4]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.

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 environment running on a Linux-based OS, where 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.

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 [5] 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 banks or same bank but different rows within the DRAM [5]. 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.

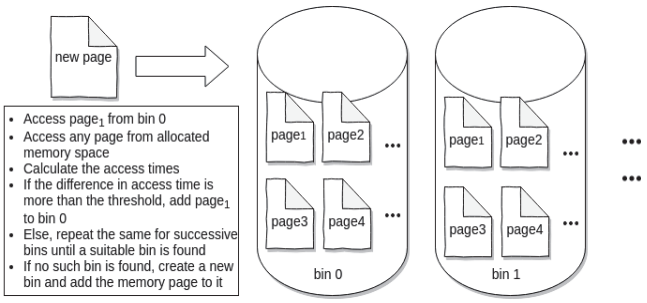

Fig. 1: An overview of the bin partitioning process

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 [6]. 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 [7] in a multi-user environment. Now, the adversary waits for the victim to load the T-tables into the

[5]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.

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

[7]Long Term Support (LTS) version, supported until 11^{th} Sep, 2023.

TABLE I: Experimental Setup

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

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. The results of inducing faults in OpenSSL AES T-tables is shown in Table II.

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

No.	Time (mins)	T-table index	Value before flip	Value after flip
1	538	Te0[25]	b3d4 d467	a3d4 d467
2	224	Te1[254]	d66d bbbb	c66d bbbb
3	12	Te3[87]	cbcb 468d	cbcb 460d
4	105	Te3[148]	2222 6644	0222 6644
5	67	Te1[193]	88f0 7878	88f0 7868

V. Discussion and Future Work

Given the use-case of ours induces faults in AES T-tables, an efficient fault attack is required to extract the secret key by analysing the faulty ciphertexts generated due to the induced faults in T-tables. There exist a large body of work addressing fault attacks (FA) on different classes of cryptographic primitives, especially on block ciphers like AES [6]. Our proposed attack ExplFrame induces single bit faults in the T-tables which persist until they are reloaded. This typical fault model matches with the one proposed in [7] (popularly referred to as Persistent Fault Attack (PFA)). 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. However, the fault model makes two strong assumptions - The adversary exactly knows the value that got corrupted inside the T-table and the corrupted entry is accessed at the last round of AES computation. It is worth mentioning that fault injection by Rowhammer provides limited control over the location of faults. More specifically, *it is hard to control which bit of the T-table gets corrupted.* Also, OpenSSL uses four T-tables T_0, T_1, T_2 and T_3 for all the rounds r_i ($1 \leq r_i \leq 10$). Therefore it is difficult to ensure that the corrupted entry is accessed only at the last round. Interestingly, our investigation revealed that the faults induced by Rowhammering are still exploitable for key extraction. In this context, we propose an improvised fault attack along the lines of PFA where we continue the partial decryption 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 can be utilized for a complete key recovery which we leave as a future direction to our work.

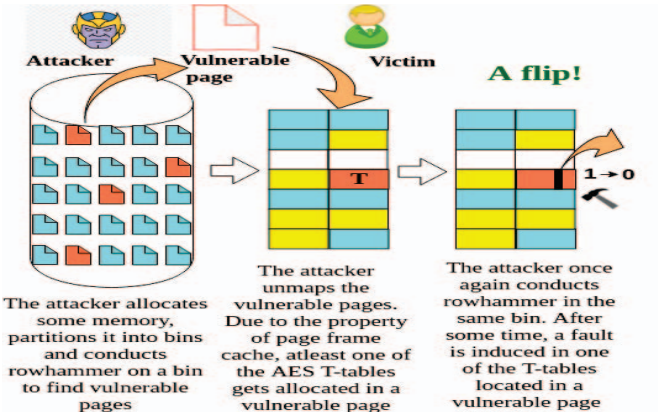

Fig. 2: An overview of ExplFrame on OpenSSL AES T-tables

VI. 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 (v1.1.1) AES to induce faults in T-tables.

Acknowledgement

This work was partially supported by Qualcomm India Pvt. Ltd. under grant 'Automating Fault Based Cryptanalysis in Secured Embedded Applications using Machine Learning' and Information Security Education and Awareness (ISEA), MeitY, Govt. of India.

References

[1] Y. Kim and et. al., "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ACM SIGARCH Computer Architecture News*. IEEE Press, 2014, pp. 361–372.
[2] OpenSSL, "Openssl cryptography and ssl/tls toolkit," 2019.
[3] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *S&P '20, in press*.
[4] S. Bhattacharya and D. Mukhopadhyay, "Curious case of rowhammer: flipping secret exponent bits using timing analysis," in *CHES '16*. Springer, 2016, pp. 602–624.
[5] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *USENIX Security '16*, 2016, pp. 565–581.
[6] 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.
[7] F. Zhang and et. al., "Persistent fault analysis on block ciphers," in *IACR TCHES*, 2018, pp. 150–172.