Fault Attacks on Encrypted General Purpose Compute Platforms

Robert Buhren[3], Shay Gueron[1,2], Jan Nordholz[3], Jean-Pierre Seifert[4], Julian Vetter[3]
[1]shay@math.haifa.ac.il, University of Haifa, Israel
[2]Intel Corporation, Intel Development Center, Israel
[3]{robert, jnordholz, julian}@sec.t-labs.tu-berlin.de, TU Berlin, Germany
[4]jean-pierre.seifert@telekom.de, TU Berlin, Germany

ABSTRACT

Adversaries with physical access to a target platform can perform cold boot or DMA attacks to extract sensitive data from the RAM. To prevent such attacks, hardware vendors announced respective processor extensions. AMD's extension SME will provide means to encrypt the RAM to protect security-relevant assets that reside there. The encryption will protect the user's content against passive eavesdropping.

However, the level of protection it provides in scenarios that involve an adversary who cannot only read from RAM but also change content in RAM is less clear.

This paper addresses the open research question whether encryption alone is a dependable protection mechanism in practice when considering an active adversary. To this end, we first build a software based memory encryption solution on a desktop system which mimics AMD's SME. Subsequently, we demonstrate a proof-of-concept fault attack on this system, by which we are able to extract the private RSA key of a GnuPG user. Our work suggests that transparent memory encryption is not enough to prevent active attacks.

Keywords

Fault injection; main memory encryption

1. INTRODUCTION

Adversaries use cold boot attacks [13, 27] to steal data from main memory. Transparent encryption of the main memory (with a secret key that is not stored in RAM), leaves the adversary with only one snapshot of ciphertext, therefore providing perfect mitigation against it. Hardware vendors also acknowledged this threat, with AMD announcing new processor extensions to mitigate such a threat. AMD's SME (Secure Memory Encryption) [19] provides ways to encrypt parts of the RAM and leave the adversary only with ciphertext.

Although the above mechanism protects the privacy of the user's data, it is less clear whether such transparent encryption is enough against an active adversary. Looking at previous work (e.g., [3, 6, 30]) reveals that a number of different hardware interfaces, e.g., Thunderbolt, Firewire, PCIe, PCMCIA and USB ports, can be used for attack purposes. Again the threat can be mitigated by adding integrity protection to the RAM. Unfortunately, adding integrity to the RAM is complex [11, 29] and requires dedicated storage on the RAM, for integrity tags, consuming an overhead of >20%.

Thus, the above discussion leads to the following question, which is the focus of this paper. Can transparent memory encryption protect the system from active attacks and obviate the need for expensive authentication? The presence of encryption limits the active adversary in a fundamental way, by blinding him in two ways. He does not know what data is being changed on the encrypted RAM, and he has no control on the resulting value when the modified RAM is decrypted. The security properties in our scenario cannot be proven, so the discussion is reduced to the question of the practicality of two-way blinded attacks. We address the problem here and show that the protection offered by transparent encryption against active adversaries is not guaranteed.

We make the following contributions: a) We designed a memory encryption scheme that replicates the functionality SME implements in hardware. Our prototype is embedded into the Linux kernel. b) We built a proof-of-concept fault attack on GnuPG [20]. With our attack we are able to reveal the private RSA key of a GnuPG user. We employed a mechanism based on LLC (Last Level Cache) probing to time our attack. We combine this information with a kernel page allocator prediction mechanism to inject a fault into the victim application's encrypted data in order to cause a predictable effect. c) We discuss the challenges that an adversary needs to overcome in order to extend our proof-of-concept attack to a real attack.

2. PRELIMINARIES & TECHNICAL BACKGROUND

This section describes some background that is crucial for understanding the rest of the paper without requiring the reader to be a priori familiar with these details.

The Boneh-DeMillo-Lipton fault attack on RSA-CRT.

We use the Boneh-DeMillo-Lipton fault attack [7], which can be applied to a device that computes RSA signatures using the CRT. The attack is based on obtaining two signatures of the same message m. The first one is correct, and denoted by s. The second one is faulty, and is obtained by injecting some corruption (to the computing apparatus),

CODASPY'17, March 22-24, 2017, Scottsdale, AZ, USA
© 2017 ACM. ISBN 978-1-4503-4523-1/17/03...$15.00
DOI: http://dx.doi.org/10.1145/3029806.3029836

that is *timed appropriately* so that the value of s_q is computed correctly, but s_p is corrupted to s'_p. The recombination yields the faulty signature s'. It satisfies (with very high likelihood) $q = gcd(s' - s, n)$, thus leading to factorizing n and hence to discovering the secret exponent d.

Cache architecture.

Modern x86 processors have multiple levels of caches. The LLC (Last Level Cache) is usually shared among all CPU cores on the chip. However Intel splits the LLC into several parts where each CPU core has a local part of the LLC and can access remote parts with an increased latency.

The cache is divided into so-called cache lines. On a x86-64 system, the typical size of a cache line is 64 Bytes. The x86-64 caches operate in a set-associative mode. All available slots are grouped into sets of a specific size. This number varies, depending on the processor and the size of the cache. Each memory chunk can be stored in all slots of one particular set.

The addressing of the cache is determined by various bits of the physical address. The lowest bits of each address denote the offset inside the cache line. The intermediate bits determine the set. The remaining high bits form the address tag, that has to be stored with each cache line for the later lookup. Additionally, when looking at the Sandy Bridge LLC, the high address bits are also taken into consideration for the calculation of the cache slice [15].

When looking at the set-associativity, it can be observed that, memory addresses with identical index bits compete on the available slots of one set. Hence, memory accesses may evict and replace other memory content from the caches.

3. ATTACK MODEL

In this paper we consider general purpose compute platforms (e.g. desktop computers or laptops) which are located in an untrusted environment. For such a scenario AMD's memory encryption provides an appealing option. SME is a general purpose mechanism to encrypt the RAM, which works on desktops, laptops and workstation systems.

We consider the following attack scenario. The adversary was able to install an unprivileged malware process on the system. He however does not have root privileges on the system. The adversary can also physically access the platform (e.g. plug in a USB stick or connect a firewire device). Of course, we suppose that the victim is aware of the valuable assets on his compute platform, and has therefore activated main memory encryption to protect specific processes. It is important to note that we do not assume any vulnerabilities in the underlying OS kernel. We also do not assume that the adversary and the victim necessarily share CPU cores.

In particular, this leads to the following assumptions: a) the memory is encrypted, i.e., the adversary does not know what values are encrypted; b) the memory accessing tools can retrieve only ciphertext, and the adversary has no access to the encryption keys; c) the adversary has the ability to modify memory locations using a physical device (as described in Section 1). Since he modifies only ciphertext, the modifications lead to some kind of unpredictable corruption of the plaintext.

4. SOFTWARE BASED MAIN MEMORY ENCRYPTION

We implemented a software based main memory encryption scheme to replicate the functionality of AMD's SME. It works transparently without any need to modify the running applications. We used AES as the block cipher and leveraged the dedicated hardware AES instructions (AES-NI).

4.1 Implementation

We wrote a kernel module that notifies the kernel on "protection worthy" applications, and extended the Linux kernel itself to perform the encryption. From the driver, we are able to enable/disable the encryption in general, and to notify the kernel about processes (identified by their process IDs) deserving encryption. The driver holds a protection list with PIDs, for which the memory pages should be encrypted when the process is currently not running.

Figure 1 illustrates the main encryption procedure. When the Linux OS decides to schedule a new process, it calls the function `schedule()`. In this function, the OS switches to the new process' memory context (`context_switch()` ①). We installed a hook in this function, to determine whether the last scheduled process belonged to our protection list. If so, we call `do_encmem` ② to encrypt all present (writable) 4 KBytes pages in place. After all the pages are encrypted, we return to the `context_switch()` method ③. Subsequently, normal execution is resumed, and the memory contents of our process has been encrypted ④. We also check whether the next process is in our protection list, and then decrypt all of its writable pages to make it ready for execution.

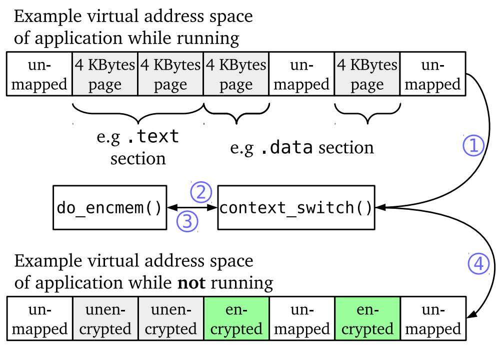

Figure 1: Main memory transparent encryption scheme. When the Linux kernel schedules a new process all present pages of our process are encrypted in place.

For simplicity, the encryption code and the necessary keys are stored in RAM, but it does not matter for our demonstration that only tests a fault injection. Other publications [9, 26] have already shown how to store cryptographic material outside of RAM and also perform cryptographic computation without leaking sensitive material to RAM. Our scheme could easily adopt such mechanisms. Still, it is important to note that our implementation does not provide a complete and secure solution by any means. Its sole purpose is to behave like a hardware scheme and provide a means to demonstrate our fault injection.

Property		Software impl.	AMD SME
1	Unit of encryption	4096 Bytes	64 Bytes
2	DMA access to enc. memory possible	yes	yes
3	Memory authentication	no	no (?)
4	Encryption enforced by	Operating system	Memory controller

Table 1: Difference between AMD SME and our software prototype.

4.2 Software Implementation vs. AMD SME

Since AMD's processor extensions SME are not available on the market yet, we implemented the software based memory encryption as close as possible to the information AMD revealed thus far [2, 19]. The requirements for the fault attack to work can be broken down into four properties. In Tab. 1 we show what these properties are and to what extent our software implementation differs with respect to AMD SME. We now discuss whether or not this impacts the fault attack (with the indexing we refer to the properties as depicted in Tab. 1):

1 The *unit of encryption* determines the required precision of a fault attack and the size of the affected area. On decryption every bit in the affected plaintext block might be faulty. Whether this is an advantage or disadvantage for an attacker depends on the situation: sometimes the target of the injection may lie close to other vital data which an attacker would like to keep unscathed; at other times locating the target may be more difficult, so a reduced precision requirement would be beneficial.
The attack used in our paper is not affected by the block size, as we can reliably determine the location of the target prime with sufficient granularity. Furthermore it is irrelevant for the Boneh-DeMillo-Lipton fault attack how many bits are changed.

2 According to AMD's documentation *DMA* read/write *access to encrypted memory* is *possible*. This is of course a core requirement for the attack to work.

3 All documentation from AMD show that *memory authentication* is not applied, because authenticating the entire encrypted memory would cause a substantial storage overhead.

4 For our software implementation the *encryption* is *enforced by* the operating system, therefore a determined adversary with kernel-level access could disable the encryption. However, in this paper we only consider fault attacks on the encrypted memory itself, as our kernel-level implementation only serves as a vehicle to demonstrate unauthenticated memory encryption. We do not claim that our pure software implementation provides an equivalent security level to a hardware implementation inside the memory controller like AMD SME. Thus, this difference does not impact the attack mechanism.

5. ATTACKING GNUPG

Our attack combines the fault injection principle with traffic analysis based on cache side channels. It introduces different ways to leverage this combination in order to attack cryptographic applications on a general purpose platform. To have a concise description, we kept our attack simple and describe only a proof-of-concept. We solve the practical problems in Section 6. To test our attack on a real application, we performed our fault attack on the RSA signing operation in GnuPG Version 1. As almost all commonly used email clients provide a way to integrate GnuPG into the application to sign and encrypt emails, we deemed this a reasonable target.

We first provide a short roadmap on the different components of the attack, because it integrates multiple mechanisms which have to be timed correctly in order for the attack to work. Fig. 2 gives an overview of each step of the attack. The details for every step of the attack are given in the following sections.

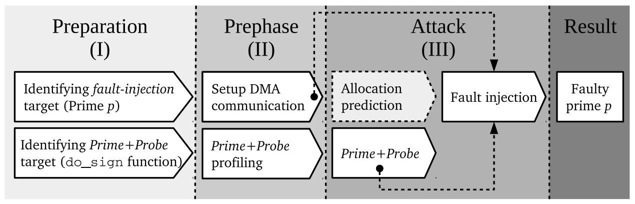

Figure 2: The three phases of our attack on GnuPG.

Preparation: fault injection target.

From Fig. 2-I can be obtained that the first step is to identify a potential fault injection target. GnuPG uses the CRT to speed up the exponentiation in RSA signatures (see Section 2). We run GnuPG 1.4.19 with the signature checking disabled[1]. An RSA key in GnuPG consists of six elements: n, e, d, p, q and q_{inv}, which are stored in a key file on disk. So for using the Boneh-DeMillo-Lipton we want to inject the fault into one of the primes p or q. The key file itself is protected with a passphrase (in this case, using 3DES). Before the signing operation commences, the user has to type in his password in order to unlock the key. Only then, the key elements are decrypted to main memory.

Preparation: Prime+Probe target.

In general *Prime+Probe* [23] monitors cache eviction. The adversary selects a number of addresses that fall into the same cache set as the one from the victim binary. In the *prime* phase the adversary fills the cache sets with his own garbage data. The adversary *idles* for a few cycles, and then *probes* in the last step. There, he measures the access time to the addresses that fall into that same cache set as the one from the victim binary. If the victim process executed at the specific address, it would have evicted some of the adversary's cache lines.

So for our attack we need to identify a feasible *Prime+Probe* target (Fig. 2-I). By inspecting the GnuPG binary we can determine an address in the executable section of the binary where the key is already present in main memory. In our case, this is the function `do_sign` in the file `g10/sign.c`. The virtual address of this function can be determined by

[1]The parameter `-no-sig-create-check` disables the signature checking.

inspecting the binary (e.g. by using `objdump`). By mapping the binary into our address space and again leveraging the `pagemap` we can determine the physical address of that function. Then, we have to determine where this physical address is stored in the LLC. Afterwards we have to determine a number of memory locations that, in terms of cache set and slice collide with the physical address of the `do_sign` function. With these information we can later carry out the *Prime+Probe* attack.

Prephase: setup DMA communication.

In the prephase of the attack we need to setup the DMA communication (Fig. 2-II) As described in Section 3 we defined that the adversary was able to spawn a process on the same host, but has also connected a remote DMA device (e.g. laptop). Now, in order to inject the fault at the right time the adversarial process has to notify the external DMA device about when to inject the fault. To do this, the adversarial process allocates a piece of memory and determines the physical address of the allocation using the `pagemap` (details are deferred to Section 6). The determined location is then sent to the external agent (who can then set his DMA device to this address location). Once negotiated, the adversarial process uses this memory location to notify the external DMA device when to inject the fault (by writing a specific pattern to this location)[2]. The adversary can of course also notify the external agent via network, but depending on the configuration of the system and the requirement for stealthiness of the attack this might be problematic.

Attack: Prime+Probe.

To successfully carry out the *Prime+Probe* attack we have to constantly calculate the mean access time over all twelve addresses. If this exceeds a certain threshold, we know that one of the addresses has been evicted from the cache, probably because the victim process has reached the *do_sign* function in its execution. The threshold value which needs to be exceeded is determined experimentally in a prephase of the attack (Fig. 2-II), details on exact values for our attacked platform are given in Section 7.

Attack: fault injection.

The actual attack starts by checking whether GnuPG was started, and then beginning to do the *Prime+Probe* (Fig. 2-III). When the attack process determines that GnuPG executes the `do_sign` function, we enforce a schedule call. GnuPG is then put in to the background, and its memory gets encrypted. This is of course just a vehicle because our software-based main memory encryption only works for processes in the background. In a real hardware implementation this would not be necessary. We then have to determine the location where to inject the fault. To do so we profiled GnuPG beforehand to determine the location of the key structure in memory (Fig. 2-III "Allocation prediction"). We predict the physical location of the key structure using a PFN leakage mechanism (again the details are described in Section 6). We then use the remote DMA to inject the fault (Fig. 2-III). For this purpose we extended the Inception framework [24] to be able to read and write memory via a FireWire cable (limitations to this approach and countermeasures are discussed in Section 8). After the fault has been injected, we resume the execution of GnuPG. The kernel will decrypt all memory pages of GnuPG to make it runnable again, among them the one with the modified memory location. Once decrypted the value of p will be faulty. GnuPG will then create the faulty signature. Finally, we calculate q offline based on the obtained faulty signature (see Section 2).

[2]When using a DMA device to inject the fault, the RAM access is still performed by the memory controller through the root complex [30], therefore ECC (Error Correcting Code) is irrelevant.

6. REAL-WORLD CHALLENGES

In Section 5 we described a fault attack against GnuPG. In order to convert this proof of concept into a real world attack, the adversary faces four challenges:

1. Determine *Prime+Probe* target addresses.
2. Physical addresses of dynamic data structures.
3. Obtain two signatures of the same message.
4. Find suitable hardware interfaces.

We address these challenges in the following sections.

Determine Prime+Probe target addresses.

In general the adversary is looking into ways to obtain the translation of virtual to physical addresses for the use in his *Prime+Probe* attack, and also for the fault injection. Which addresses the adversary wants to obtain, is of course very specific to the attacked target. In our case, this is the `do_sign` function of GnuPG. Obtaining the virtual address of this function is quite easy (as shown in Section 5). The major Linux distributions only slowly adopt position independent binaries. For now we can safely rely on the virtual addresses we can obtain when inspecting the binary with, e.g. `objdump`.

The `/proc/<pid>/pagemap` file can be used to obtain a physical address. For every user-space page, the `pagemap` provides a 64 bit value, indexed by its virtual page number, which contains information regarding the presence of the page in RAM. Bit 63 determines whether the page is present in memory and bits 0 to 54 encode its page frame number. Under the assumption that the underlying memory is shared between adversary and victim, the adversary can just use `mmap` to map the GnuPG binary into his address space. Then he uses the `pagemap` to determine the physical address of the function.

Physical addresses of dynamic data structures.

The location of the key data structure (that includes p) is not only unknown, but also differs for every execution, because it is stored on a writable page (on the heap). To obtain the physical address of this dynamic data structure an adversary can again draw on the `pagemap` mechanism.

The adversary can trace a single execution run of GnuPG with the desired commandline parameters and concurrently monitor the pagemap for new page allocations. Mere syscall tracing is insufficient, as the adversary needs to know the actual physical memory footprint of the process, not the number and size of allocations. As Linux employs a lazy allocation strategy, page ranges which have been requested by `mmap` but not touched will not have physical backing at all. Whereas other areas that do not require explicit allocation (e.g. stack growth) may indeed cause the number of consumed physical pages to increase. Once the adversary has reached the point where p is copied into memory, the trace is complete.

In our case, we determined that p will be placed on the 10th allocated page. This knowledge still does not tell the adversary the exact physical address where the key will be located in a future run of GnuPG. However he can take advantage of the fact that the Linux kernel allocator tends to give out pages that where freed shortly before. So, most recently freed memory pages are given out first to processes. Thus the adversary can allocate a number of physical memory pages (i.e. allocate them and actually perform write operations to them), consult the `pagemap` to determine their physical addresses, and free the pages again. Since the Linux kernel will most likely give out these very same pages to GnuPG the adversary can predict which physical page will contain the key. Freeing the pages has of course to be timed correctly by the adversary so that no other program gets these pages. But this can easily be combined with the already running *Prime+Probe* attack.

Our observations also showed that if our prediction fails, i.e. the 10th page allocated to GnuPG did not match the 10th from last page freed beforehand, that no newly allocated page matched any freed page at all. Also the accuracy of the prediction astonishingly depends on the overall number of pages the adversary allocated before. The exact success probability of such an attack is presented in Section 7.

Obtaining two signatures of the same message.

There are several ways an adversary could obtain a signature for the same message twice. But first there is an obstacle the adversary has to overcome. Along with the message that should be signed GnuPG puts a Unix timestamp into the hash calculation. Fortunately for the adversary, the timestamp is only in second granularity. So, when the adversary is able to motivate the victim to create two signatures from the same message shortly one after the other he gets the desired double signature.

There are a number of scenarios the adversary could leverage. Any form of signed *auto-reply* message (e.g. *out-of-office* notifications) would work for the adversary. It is possible to configure all commonly used email clients (e.g. Outlook, Thunderbird, Apple Mail, etc.) to sign *auto-reply* messages with a defined key. The GnuPG options `-passphrase-file`, `-passphrase-fd` or `-passphrase` allow a user to provide the passphrase for the signing key either directly from the commandline, some file or from a file descriptor. It is questionable, from a security perspective, to store the passphrase for the key in a file, however, such scenarios exist.

Find suitable hardware interfaces.

In our attack we rely on the FireWire interface. One could argue that modern systems might not be equipped with a FireWire controller anymore. However, our attack can not only be performed through a native FireWire port, but also via ExpressCard/PCMCIA expansion ports or a Thunderbolt to FireWire adapter. It is likely that a system has at least one of the aforementioned interfaces.

7. ATTACK RESULTS

All experiments were conducted on a Lenovo Thinkpad T520 with an Intel Core i7-2670QM CPU and 4 GBytes of RAM. This platform uses the aforementioned hash function for LLC slice determination. The device is equipped with a FireWire interface, and we verified the above described memory modification capabilities. Software-wise we used Debian 8 “Jessie” with a Linux kernel version 4.0.

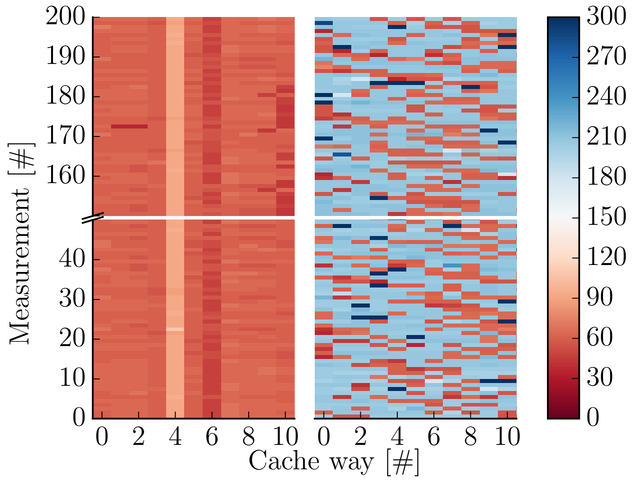

Figure 3: Cache way access times measured with `rdtscp`.

7.1 Prime+Probe results

For our attack we had to determine a threshold for the cache eviction in order to perform the *Prime+Probe* attack. All accesses were measured using the `rdtscp` instruction[3]. We divided our measurements into two sets $\{A\}$ and $\{B\}$.

- Set $\{A\}$ was our baseline: we chose twelve set-colliding memory locations, accessed all of them to allocate them into the cache, and then immediately measured the access time for all twelve locations again.
- For set $\{B\}$, we chose twelve memory locations that fell into the same cache set and slice as the address we want to monitor. We accessed all of them once, then we executed one instruction from our victim application (GnuPG) at exactly that address, and finally measured the access time for our twelve lines again.

Cache ways.

First, we had to figure out if we are able to reliably measure variations in access times to addresses from the same cache set. Fig. 3 illustrates this experiment.

Set $\{A\}$ is represented by the left column. Our assumption was that the measured access time would be very low, because all twelve locations fit into the cache. Since no other application ran in between, all access requests would be served by the cache. In [22], Levinthal shows that when an access request to an address takes $\sim$40 cycles, it indicates that it was served from the LLC. Indeed, all access times were in the range between 0 and 40 cycles.

[3]`rdtscp` is a serializing call that prevents reordering around the call, and returns the number of executed processor cycles.

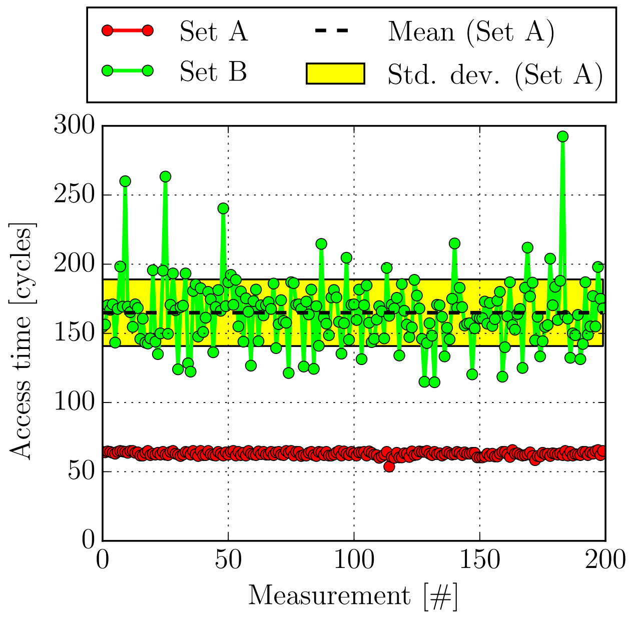

Figure 4: The mean access time over 200 measurements.

Set $\{B\}$ is represented by the right column. Our assumption was that at least one of the addresses had to be evicted from the cache, because the cache only has twelve ways. Indeed, Fig. 3 supports this assumption. The processor had to load some addresses from main memory. We assume that more than one address was evicted from the cache, because our measurements were conducted from a separate application, so not only the victim code ran in between but also other kernel scheduling code, etc. In general we confirmed that we would be able to measure when the victim application executed its binary at a certain memory location.

LLC access.

So far, we confirmed that we could recognize cache eviction in terms of cache ways. To know when a certain application hit a certain address in the executable, we had to specify a reliable threshold. To this end, we calculated the mean value over all twelve access times for set $\{A\}$ and for set $\{B\}$. Fig. 4 illustrates the result, each showing the mean access times over all twelve addresses. Clearly, the mean access time is significantly higher when one of the addresses was accessed. Therefore, we were able to set the threshold to $delta\_mean = mean(\{A\}) \pm std(\{A\})$, corresponding to the yellow bar in Fig. 4. The noise we see in the figure is due to the fact that the victim's data can be partially cached in higher-level caches (leading to faster reads), and the variation in the L1 and L2 contents affects the cache probe time and induces measurement noise.

7.2 Page allocator prediction results

As already described in Section 6 it is necessary to find the physical address of the prime factor p. As it is allocated on the heap it is necessary to somehow predict the right physical address where p is located on. To predict the correct physical address we did the following experiment. First we annotated GnuPG to print the virtual and physical address of the prime p. Then, in our adversarial process, we allocated a number of pages using `mmap` and calculated their respective physical address (using the `pagemap`). Afterwards we freed all these pages and let GnuPG run. We then compared if the physical address of the prime p was among our previously allocated and freed pages. We did this with a various number of allocations, ranging from only 8 allocations up to 1024. The result can be obtained from Fig. 5. The figure is split into two parts. The first green bars contain results in steps of 8, after 248 experiments we increased the number of allocations in every measurement run by 16. So the orange bars contain the results from 256 to 1024 in steps of 16. For every allocation step we performed 100 measurements.

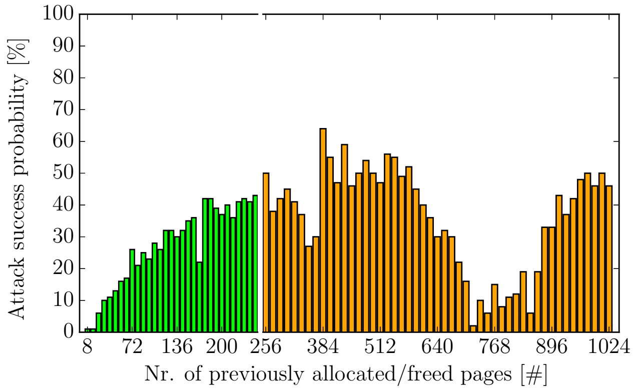

Figure 5: Attack success probability based on the previously allocated and freed memory pages.

What we observed is quite interesting. When the physical address of the prime was among the previously allocated/freed pages it was always on the same one (the 10th last page). It was entirely deterministic which of the previously allocated/freed pages would later contain the prime p. But as can be obtained from the figure only in a certain number of measurements the physical page of the prime was among the allocated/freed pages at all. Moreover, the overall success rate depended on the number of previously allocated/freed pages. We achieved a maximum success rate of ~55-60% when allocating and freeing between 380 and 500 pages before executing GnuPG. This value is already surprisingly high, given that the window of uncertainty (the time between the release of the pages and GnuPG receiving a page for p) includes the creation of a process and thus a full address space. A determined attacker could employ the techniques used in this paper to release pages much closer to the critical allocation, thus further boosting his chances.

8. RELATED WORK

In the following section we present a number of topics which are relevant for this work.

Cache-based side channels.

Cache-based side channels have a long history [28]. In recent years, the focus has been shifted toward the LLC [12, 15, 32]. But these attacks are harder to carry out, because, e.g., Intel uses an undocumented hash function to distribute addresses to different segments of the LLC. Thus, initially, a lot of effort has been put into reverse engineering this hash function for different processor generations [15,25]. Based on these new findings, a number of attacks have been proposed. Yarom et al. [32] show how to extract the private encryption keys from a victim program in a single operating system and also over VM boundaries.

Mitigating cache-based side channels remains a challenging task. In [21], Kong et al. investigate several schemes to mitigate cache attacks. Also, modern Intel server CPUs provide a technology called CAT (Cache Allocation Technology) [17] to prevent traffic analysis by other processes or VMs through eavesdropping the LLC.

procfs.

The procfs provides user applications with useful information about the current system state. However, often these information can be used as a means to spy on other processes or resources in a system. Jana and Shmatikov [18] show how to exploit information provided by procfs to create detailed profiles of applications. They use memory consumption discrepancies of a browser to determine which website the user currently looks at. Zhou et al. [33] leverage, among other sources, the procfs and sysfs to gather detailed information about the user of an Android smartphone.

procfs is an important resources in Linux' system architecture. Many applications (e.g. `ps`, `netstat`) rely on information exported through procfs. Therefore disabling this filesystem is not an option. However, as already done with many other files in procfs the access to some security critical files should be limited to admin users. For the `pagemap` this already happened. Since Linux kernel version 4.2, when reading the file only returns valid data if the user holds an admin capability. The access to other security critical information that are exported through procfs, e.g., `kallsyms` were also limited in the past.

DMA.

In order to launch our fault attack we rely on DMA. A lot of attacks have been proposed to launch a DMA based attack using various interfaces [3, 31]. In 2006 Boileau [6] showed the remote DMA capabilities of the Firewire bus. More recent attacks have been proposed by Sevinsky [30] in 2013. Sevinsky used the Thunderbolt interface to launch a DMA attack.

The obvious countermeasure to prevent DMA attacks is using an IOMMU [1, 16]. However, when running a 32Bit Linux kernel the IOMMU (if present) never gets enabled. On 64Bit systems the Linux kernel makes use of the IOMMU to block remote DMA accesses. However, it is highly processor specific whether the system contains an IOMMU or not. If no IOMMU is present users are advised to disable the bus master capabilities of specific devices.

Fault attacks.

Fault attacks are a well-known concept in computer security. Initial work on fault attacks was done by Boneh et al. [7] in 1997. In this groundbreaking work they show that various cryptographic algorithms can be attacked using hardware fault injection. Since then, several fault attacks have been proposed. In 2003 Dusart et al. [10] describe a differential fault analysis attack on AES. They are able to break an AES128 key with around 10 faulty messages.

In the early work of Boneh et al. [7], the authors already propose the use of random padding[4] to protect against their attack (as suggested by Bellare and Rogaway [5]). Also cryptographic padding (e.g. OAEP [4]) was already discussed in [5, 7] and would also prevent the fault attack to work.

Main memory encryption.

Henson et al. [14] provide a survey on memory encryption techniques. Other work on main memory encryption has been done by Chhabra et al [8]. They propose an encryption scheme for systems with NVMM (non-volatile main memory). The attack scenario is valid for mobile devices, where an adversary would be able to easily read out the RAM cells. In [26], Müller et al. propose an architecture called TRESOR. TRESOR makes cold boot attacks difficult, because instead of using RAM, it ensures that all encryption states as well as the secret key and any part of it are only stored in processor registers.

Memory authentication.

An integrity tree is the classical way to provide authentication to a larger amount of data. Elbaz et al. [11] provide an overview of several concepts for memory authentication. Among them are classical concepts like Merkle Trees. These are binary trees where each node holds a hash digest of its two children, and the lowest leaves hold digests of the protected data units. However, the overhead in storage ($\sim$20%) is infeasible when condering to encrypt the entire main memory.

Signature verification.

It is important to note that we did not exploit any vulnerability in GnuPG. In default operation mode, GnuPG verifies the generated signatures, and if an error is detected, it terminates with an error report without releasing the (faulty) signature. However, the command line parameter `-no-sig-create-check` exists, and made our attack possible. Fortunately, this parameter exists only in GnuPG version 1, and GnuPG version 2 removed the command line option completely. The OpenSSL library always verifies a signature before releasing it, and repeats the computation without using CRT in case an error is detected. There is no option to disable the check other than by modifying the source code, and this type of threat is not part of our attack model.

9. CONCLUSION

Threat scenarios that include an active adversary on a dynamic system require memory encryption for privacy, and memory authentication for integrity. The scenario became even more relevant due to the fact that hardware vendors now acknowledged this threat and therefore provide solutions. So the question addressed in this paper is the following. Is it reasonable (and to what extent), in order to save the high cost of dedicated authentication mechanisms, to rely on encryption to protect both privacy and integrity?

To answer this question, we showed the possibility of fault attacks on memory that is encrypted with no authentication. Of course, our attack is complex, and implementing it even as a proof-of-concept was a serious challenge. The complexity results from the protection that the encryption provides by itself.

[4]It is interesting that GnuPG version 1 is not using random padding.

Nevertheless, this work clearly illustrates that sophisticated attacks against the integrity of the memory cannot be dismissed, or at least cannot be ruled out. We therefore propose that memory encryption techniques should include integrity protection, despite the added complexity and performance costs.

10. ACKNOWLEDGMENTS

The second author is supported by the PQCRYPTO project, which is partially funded by the European Commission Horizon 2020 research Programme, grant #645622, by the Blavatnik Interdisciplinary Cyber Research Center (ICRC) at the Tel Aviv University, and by the ISRAEL SCIENCE FOUNDATION (grant No. 1018/16).

The fifth author is supported by the Helmholtz Research School on Security Technologies.

11. REFERENCES

[1] Advanced Micro Devices Inc. AMD I/O Virtualization Technology (IOMMU) Specification, February 2015.

[2] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming, April 2016.

[3] M. Becher, M. Dornseif, and C. N. Klein. FireWire: all your memory are belong to us. *Proceedings of CanSecWest*, 2005.

[4] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *EUROCRYPT'94*, pages 92–111. Springer, 1994.

[5] M. Bellare and P. Rogaway. The exact security of digital signatures-How to sign with RSA and Rabin. In *EUROCRYPT'96*, pages 399–416. Springer, 1996.

[6] A. Boileau. Hit by a bus: Physical access attacks with Firewire. *Presentation, Ruxcon*, page 3, 2006.

[7] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *EUROCRYPT'97*, pages 37–51. Springer, 1997.

[8] S. Chhabra and D. Solihin. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *38th ISCA*, pages 177–188, June 2011.

[9] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *ASPLOS*, pages 177–189. ACM, 2015.

[10] P. Dusart, G. Letourneux, and O. Vivolo. Differential fault analysis on aes. In *Applied Cryptography and Network Security*, pages 293–306. Springer, 2003.

[11] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In *Transactions on Computational Science IV*, pages 1–22. Springer, 2009.

[12] D. Gruss, R. Spreitzer, and S. Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security*, pages 897–912, 2015.

[13] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[14] M. Henson and S. Taylor. Memory encryption: a survey of existing techniques. *ACM Computing Surveys (CSUR)*, 46(4):53, 2014.

[15] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *IEEE Security and Privacy*, pages 191–205, 2013.

[16] Intel Corporation. Intel Virtualization Technology for Directed I/O, October 2014.

[17] Intel Corporation. Improving Real-Time Performance by Utilizing Cache Allocation Technology Enhancing Performance via Allocation of the Processor's Cache. Whitepaper, April 2015.

[18] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Security and Privacy*, pages 143–157. IEEE, 2012.

[19] D. Kaplan, J. Powell, and T. Woller. White Paper AMD Memory Encryption, April 2016.

[20] W. Koch. The GNU Privacy Guard, January 2016. https://www.gnupg.org/.

[21] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou. Architecting against software cache-based side-channel attacks. *Computers, IEEE Transactions on*, 62(7):1276–1288, 2013.

[22] D. Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30, 2009.

[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Security and Privacy*, pages 605–622, 2015.

[24] C. Maartmann-Moe. Inception. http://www.breaknenter.org/projects/inception/, April 2016. Accessed: 2016-04-26.

[25] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse engineering Intel last-level cache complex addressing using performance counters. In *Research in Attacks, Intrusions, and Defenses*, pages 48–65. Springer, 2015.

[26] T. Müller, F. C. Freiling, and A. Dewald. TRESOR Runs Encryption Securely Outside RAM. In *20th USENIX Security*, pages 17–17, 2011.

[27] T. Müller and M. Spreitzenbarth. Frost. In *Applied Cryptography and Network Security*, pages 373–388. Springer, 2013.

[28] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer, 2006.

[29] D. Owen Jr. *The feasibility of memory encryption and authentication.* 2013.

[30] R. Sevinsky. Funderbolt: Adventures in Thunderbolt DMA Attacks. *Black Hat USA*, 2013.

[31] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Xen 0wning trilogy. *Invisible Things Lab*, 2008.

[32] Y. Yarom and K. Falkner. Flush+reload: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security*, pages 719–732, 2014.

[33] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *2013 ACM CCS*, pages 1017–1028. ACM, 2013.