Real-World Threats From In-Use “Blind Random” Block Corruption

Rodrigo Branco [1,2]
[1] Oregon State University, USA
rodrigo@kernelhacking.com
Shay Gueron [2,3]
[2] Meta, USA
[3] Department of Mathematics, University of Haifa, Israel
shay@math.haifa.ac.il

Abstract—We present a novel attack on encrypted data in use. Our threat model is an attacker who has compromised a cloud provider and has gained physical access to platform hardware. A realistic example is an insider who delivers an exploit and implant in stages.

We demonstrate how this attacker subverts a running Linux guest OS with only 2^{32} calls to a targeted corruption primitive. We compare this to a naive 2^{128} work estimate obtained using a standard cryptanalytic model for the encryption of data at rest.

Beyond our novel attack surface, we follow [1] and illustrate how choosing an appropriate cryptanalytic model is essential for assessing, and then protecting, the real world security of memory encryption, especially for data in use.

1 Introduction

In [2], Microsoft defines a security threshold whereby an attacker ’with casual physical access cannot modify data or code on the device’ [2]. Given an insider threat, we further differentiate between a casual access vector and one that is developed by time, effort, and the deployment of special purpose tools, including staged exploits and implants. A taxonomy of attack capabilities, relative to system access, is given in [3]. Here, there are three levels of attacker:

- *Access Seeking Attacker*: An attacker that does has access to a platform but does not own it. The platform is effectively in a locked state, where both disk and memory are encrypted. The objective is to steal information on the system.
- *Breaching Attacker*: An attacker that is a legitimate user of the platform, and wishes to escalate privilege. An example is a bypass of system policies or restrictions. A classical scenario for this is a corporate employee who has access to his own computer as a legitimate user, even as a local administrator , but is still limited by corporate policy, DLP and other technologies. Credential theft is part of this threat model, because domain login often permits cross-machine usage.
- *Conspirator Attacker*: An attacker that is also a legitimate user of the platform, with full administrative powers. The intention of this attacker is to collect other users’ data. The common example is in a cloud environment where the administrator of the host machine is not expected to have exposure to the VM contents. The memory encryption in-use for those cases usually offers per-VM keys and is intended to prevent the host from attacking the guest. This scenario also assumes the hypervisor is trusted and not compromised, and some kind of attestation exists to guarantee the administrator cannot modify either the host OS or the hypervisor.

Active static attacks try to perform memory analysis, but while the memory is not in-use. A well-known example is the cold boot attack [4]. For this class of attacks, countermeasures such as VT-d are insufficient because DIMM is connected to an unprotected system. However, memory encryption can be a legitimate countermeasure if deployed appropriately. Specifically, the attacker must face authenticated encryption, and that all artifacts of secret key material well protected in hardware. A common failure happens when artifacts of cryptographic key material reside in the same memory space as the data, such as an errant AES round key. We refer to the following hardware-based proposals of memory encryption for appropriate boundaries: [5] [6].

When physical access is part of the threat model, even through an intermediate device such as a DMA controller which can read and write on system memory [7] [8] [9], an *active dynamic attacker* is able to change memory contents while in-use. New standards such as the one for USB 3 expand on the relevant DMA support for devices. Mitigating such attacks is non-trivial: disabling bus mastering is not enough, as a purposefully implanted device is able to transact with runtime memory. Countermeasures such as VT-d [10] are not available in all platforms, despite ongoing efforts from Microsoft.

One way to protect against dynamic memory modifications is to use full-disk encryption for data at rest, where solutions are widely available; examples include Bitlocker on Microsoft Windows, FileVault on Apple MacOS and eCryptfs on Linux Distributions. Here, the security relies on a mechanism for authentication, either as part of the chaining mode for the block cipher, or as a companion hash function or MAC.

While full disk encryption provides assurance for data-at-rest, the main memory (DRAM) itself can be targeted by an active dynamic attacker [9] [7] [8]. On cloud settings, a cloud provider’s policy and intentions could be trusted and correctly implemented, but employees and system administrators are still a potential insider threat.

Full system memory encryption, with per-VM encryption

keys in the virtualized case, could thwart this active dynamic attacker. However, system-wide memory encryption has an additional challenge: DRAM size makes authentication prohibitively expensive. A list of systems that use memory encryption without authentication is provided in [11] as follows:

- Microsoft's Xbox (for a part of the memory space);
- Nintendo's 3DS Security Processor;
- Apple's Secure Enclave Processor (for the data stored in external DRAM in iPads [8]);
- AMD's Secure Encrypted Virtualization (SEV) and Secure Memory Encryption (SME), using encryption with a per-virtual machine key (SEV), and/or over the whole system memory;
- Intel's Multi-Key Total Memory Encryption (MKTME) and Total Memory Encryption (TME), using encryption with a per-virtual machine key (MKTME), and/or over the whole system memory.

Consequently, any such deployment of encryption for data in use that does not include an explicit mechanism for authentication requires a totally different cryptanalytic model for accurate costing [1]. The standard costing model used in authenticated encryption for data-at-rest simply does not apply, as we will demonstrate in the next section.

A common INFOSEC assumption for the encryption of data in use is that since the data in memory is encrypted, any modification on the ciphertext will result in a random corruption of the plaintext when the CPU reads the modified ciphertext block. The claim in [5] is that ciphertext modifications inevitably lead to a system crash, and therefore there is an implicit mechanism for authentication.

Towards a pressure test of this INFOSEC assumption in the real world, we sketch a plausible threat model as follows:

- The attacker has read and write access to the DRAM, outside the encryption boundary, while all CPU-DRAM traffic still goes through the encryption mechanism. Here, encryption guarantees the confidentiality of the data on the DRAM, so the attacker's read primitive is exposed to cipher only. Additionally, DRAM changes lead to arbitrary and unpredictable values that eventually are consumed by the CPU when it reads the modified ciphertext. Hence, the attacker has a workable write primitive, but is 'blinded' and can only cause 'random' corruptions of memory consumed downstream.
- Without loss of generality, we assume that AES is the underlying block cipher, hence the underlying block size is 128 bits. We also assume that an appropriate chaining mode for AES is used, so that any modification of encrypted memory would corrupt at least one block of cipher.
- The active dynamic attacker can verify that corruptions of encrypted data in use are consumed by the hypervisor, and so the attack has a nontrivial measure of control in the downstream consumption.

In HOST 2016 [3] researchers demonstrated a new class of *active dynamic* attacks that are applicable to this threat model, with so-called *Blinded Random Block Corruption (BRBC)* attacks. Later that year, researchers demonstrated similar attacks against a varied set of randomization approaches [12]. The [3] work exposed a main weakness of using encryption without integrity for system memory: data-only attacks (attacks that do not corrupt code areas) create undefined and unexpected software behavior. This breaks the hardware guarantee that data is not modified outside of the scope of the running software stack, unless explicitly desired/configured by software itself. The attacker's ability to modify data at any point in time introduced a TOCTOU (time-of-check, time-of-use) race condition, even when the attacker has limited control of their write primitive. However, there is no TOCTOU race condition issue using at rest; it only manifests with data in use.

An improved *BRBC*-style attack was presented in H2HC (Hackers to Hackers Conference) [13] using a less restrictive scenario but with bigger impact: compromised virtualization at a cloud provider. By leveraging the VMWare Debug Feature in the hypervisor, the researchers demonstrated a *BRBC*-style, data-only attack that subverted the per-VM encryption mechanism. In [11], this improved attack was applied to other mechanisms for memory encryption. For example, having different keys per virtual machine, as in AMD SVE [5] or Intel MKTME [6].

Currently, major hardware platforms (ARM Newmore [14], AMD SNP [15] and Intel TDX [16]) support transparent memory encryption, mostly without authentication, or with mechanisms whose authentication is so weak that it can not defeat a hardware-based implant as part of the threat model.

We now present a novel, *active dynamic* attack against encrypted memory that is far less dependent on the specific code construct in [3]. In doing so, we hope to demonstrate that a real-world solution needs to be in hardware and not software, and that explicit authentication is a vital component of the encryption of data-in-use.

2 Our Attack

Overview Our attack acts against sparse data structures with small targets, relative to the 128-bit block size of the underlying encryption. For example, in a structure with four 32-bit integers, we focus on one of four for a write primitive, and ensure that the corruption of the remaining three do not cause a crash. In a later section we demonstrate an *an entire family* of such targets in the Linux kernel, so this is not a real-world limitation.

A naive cryptanalytic costing borrowed from a model for encrypted data at rest would assure 128 bits of security for any given block, including the block that houses the targeted data structure [1]. And so, in order to achieve the attacker's goals, brute-force would have to be applied, costing 2^{128} calls (i.e. 128 bits of work) to the corruption primitive.

However, an active dynamic attacker who is effectively operating from platform hardware works against encrypted data in use, which we assume is otherwise unauthenticated. The costing of their work to achieve a successful surface is then measured by the size of their software target; in the case above, a single 32-bit integer. As a consequence, 2^{32} calls to a blockwise 'blind' 'random' corruption primitive (i.e. 32-bits of work) is sufficient for compromise.

Notice also how the attack does not assume a software vulnerability, nor a defeat of the block cipher or its chaining mode. The success of the attack relies in no small part upon misplaced assurances from an inappropriate cryptanalytic model [1].

Framework [3] demonstrates how blind, data-only attacks are possible against memory encryption without authentication. While a huge part of the memory is unknown to an attacker, the attack is not oblivious to memory changes. In fact, many attack-relevant properties are easily identified, including the layout of running programs on the target machine as well as the location of relevant data structures. A determined attacker will have no trouble finding ample decision points in the program logic, nor

will they lack mechanisms to influence and eventually control those decision points.

For example, a condition like in Listing 1:

Listing 1. Non-zero comparison:
```
if ( data != 0 )
        user_is_authenticated();
```

can be forced to enter the `if` by modifying encrypted memory. The fact that the modification is not bitwise controlled (i.e. the modification is 'blind' 'random') does not matter, since any nonzero target value works for the attacker. However, while the researchers demonstrated a real-world vulnerability, their technique is highly limited and the scope is very narrow.

Our attack has a much wider scope. Our first observation toward this goal is that code and data fetches happen in 128-bit block boundaries. Given cache sizes of 512 bits, code and data fetches access multiple blocks. Figure 1 shows the encrypted memory and the plain memory, with the encryption/decryption steps.

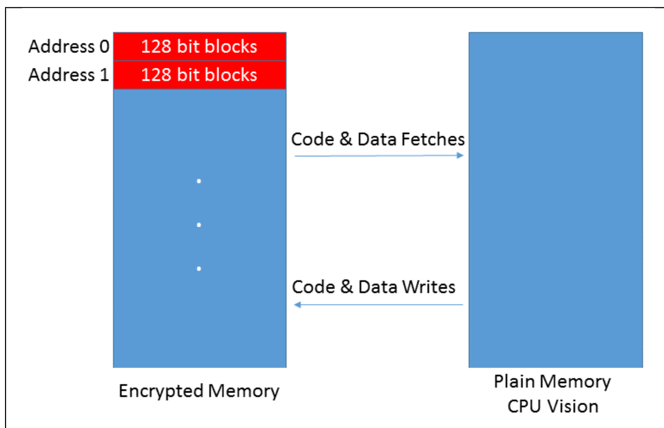

Fig. 1. Showing address in the encrypted memory as block locations

The standard cryptanalytic model for costing data-at-rest encryption assumes that all bits in a given block of cipher at rest are secure, and they often are with appropriate authentication. For AES-128, the standard model indicates that 2^{128} calls to a 'blind' 'random' corruption primitive are required for the attacker's desired outcome. Counterexamples in the real world do exist and invariably exploit a mistake in the implementation of the key derivation or the chaining mode of AES. However, there is nothing wrong with the cryptanalytic costing in these cases; all faults lie in the implementation space.

However, for data-in-use encryption, the underlying data structure sizes are different than the block size of the encryption, and are often much smaller. Figure 2 shows an "exploded" vision of the encrypt/decryption process, adding some underlying data types in memory.

As shown, access to a variable incurred in a fetch for the adjacent values (128 bit blocks) are decrypted *together*! Var A, B and C (of different types) are all part of the same 128 bit block. Any corruptions in the encrypted block forces 'blind' 'random' corruption in the decrypted block. Given the fact that Var A is typed as an `unsigned char`, an attacker interested in having a specific value for Var A only needs 2^8 calls to the corruption primitive, or 8 bits of work.

The prerequisites for success are as follows:

- The attacker has a posture in platform hardware that identifies the location of targeted data structures in memory.

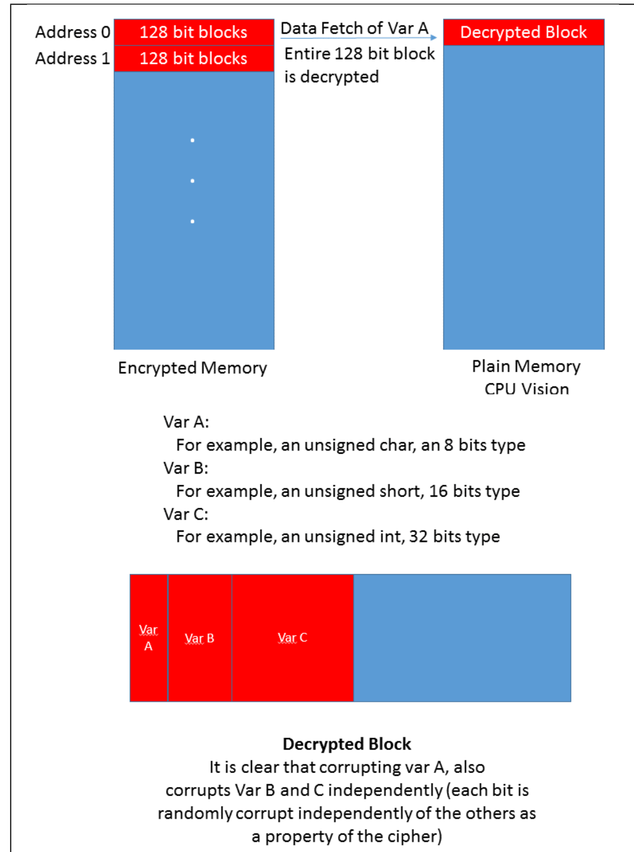

Fig. 2. Decrypted Block Vision

- Incidental data targets (i.e. collateral damage in the write primitive) do not force a crash.
- The code logic can be exercised as many times as needed for the work requirement mandated by the appropriate cryptanalytic framework.

Our experiments suggest that a real-world attacker with these prerequisites in hand can achieve 32 bits of work in less than 15 minutes on a wall clock, assuming an ordinary hypervisor, an ordinary guest Linux OS, and ordinary mechanisms for hardware-based memory encryption such as the ones illustrated in the visions above.

2.1 Elevating Privilege

Our attack target is a data structure in the Linux kernel responsible for controlling process permissions: `task_struct`. We assume the attacker has unprivileged software running on the target machine with user privileges and our objective is to make this process supervisory (i.e. root). We assume the target system has no known vulnerabilities and the interaction with the system is either through the process or external to the encryption layer (for example, the corruption of memory outside of encryption scope). The attacker process essentially creates the layout required for the external discovery of the target process's structure location. This essentially consists of spawning multiple processes and terminating them to create a visible - and predictable - pattern change in the memory layout.

For the targeted data structure for the corruption, auxiliary information is in Listing 3.

Listing 2. Process Credentials in task_struct - sched.h:
```
http://lxr.free-electrons.com/source/include/
    linux/sched.h
```

```
/* process credentials */
1659            const struct cred __rcu *
    real_cred; /* objective and real
    subjective task
1660
    * credentials (COW) */
1661            const struct cred __rcu *cred;
    /* effective (overridable) subjective task
1662
    * credentials (COW) */
```

The `task_struct`, as can be seen in Listing 2 has a `struct cred` element that contains the target we are interested in, as listed in Listing 3:

Listing 3. Cred Structure in cred.h:

```
http://lxr.free-electrons.com/source/include/
    linux/cred.h#L23
109 struct cred {
110         atomic_t        usage;
111 #ifdef CONFIG_DEBUG_CREDENTIALS
112         atomic_t        subscribers;    /*
     number of processes subscribed
*/
113         void            *put_addr;
114         unsigned        magic;
115 #define CRED_MAGIC      0x43736564
116 #define CRED_MAGIC_DEAD 0x44656144
117 #endif
118         kuid_t          uid;            /*
     real UID of the task */
119         kgid_t          gid;            /*
     real GID of the task */
120         kuid_t          suid;           /*
     saved UID of the task */
121         kgid_t          sgid;           /*
     saved GID of the task */
122         kuid_t          euid;           /*
     effective UID of the task */
123         kgid_t          egid;           /*
     effective GID of the task */
124         kuid_t          fsuid;          /*
     UID for VFS ops */
125         kgid_t          fsgid;          /*
     GID for VFS ops */
```

The sequence of elements: `uid`, `gid`, `suid`, `sgid`, `euid`, `egid`, `fsuid`, and `fsgid` contain the information related to the user and group ID for the task. If we change the `euid` to 0, we are able to transform an unprivileged task into a privileged one. If we follow the `kuid_t` type definition, we find that it is unsigned int, as shown in Listings 4, 5, 6.

Listing 4. kuid_t typedef in uidgid.h:

```
http://lxr.free-electrons.com/source/include/
    linux/uidgid.h#L22
20 typedef struct {
21         uid_t val;
22 } kuid_t;
```

Listing 5. uid_t typedef in types.h:

```
http://lxr.free-electrons.com/source/include/
    linux/types.h#L31
typedef __kernel_uid32_t        uid_t;
```

Listing 6. typedef's in posix_types.h:

```
http://lxr.free-electrons.com/source/include/
    uapi/asm-generic/posix_types.h#L48
47 #ifndef __kernel_uid32_t
48 typedef unsigned int    __kernel_uid32_t;
49 typedef unsigned int    __kernel_gid32_t;
50 #endif
```

Notice that if we target the effective user id (`euid`) of the process, no matter the alignment of the block related to our target, the values that are corrupted all pertain to the task of interest and do not affect its execution permissions in negative ways; the 'blind' 'random' approach does not crash the system.

A visual representation of the `euid` and bordering elements inside a block can be seeing in Figure 3. That essentially means we can elevate privileges of *ANY* process running in the system upon the diagnosis of its location, which in a collaborating process is very possible.

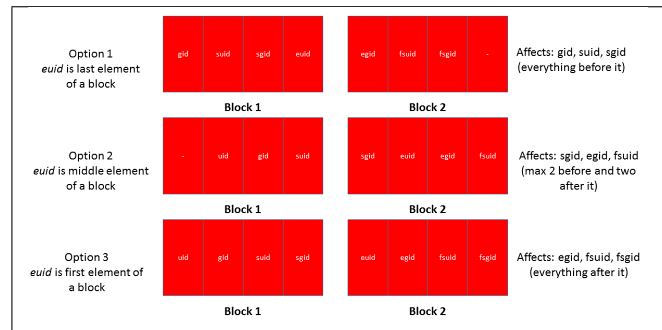

Fig. 3. Alignment inside the target block does not matter, since the 'blind' 'random' corruption of adjacent elements does not cause a crash.

2.2 Locating And Diagnosing User-mode Targets

While our attack clearly focuses on elevating privileges and uses a kernel target, the amount of valid targets is highly dependent on system specifics, and as such, depends on the willpower of the attacker. For example, the attacker might just flip special bits as demonstrated by [3]), or even do something more library-specific like probe an older OpenSSL's certificate verification.

User-mode targets are also more amenable to attack, given that they rarely brick the entire system if haphazardly chosen [17]. A determined attacker will not have trouble differentiating between user-space and kernel-space memory targets.

Isolated periodic interrupt handling or interrupt handling routines also represent a viable attack surface. With that, an attacker is able to generate network traffic against the target VM and inspect corresponding memory changes in the small data structures propagating into the DMA buffers. Interestingly, the attacker is able to repeat it as frequently as necessary to develop the confident of hitting an optimal target in running memory, similarly to what we have discussed using *fork()*.

This is specially true if the attacker has an authoritative position in the cloud provider, and hence is an insider threat. Such an actor can deploy a suite of custom tools to control the platform hardware, and thus control devices that generate specific interrupts to better analyze system behavior.

Furthermore, modifications via simple memory row side-channels ([18]) offer an attacker the ability to identify code logic activated by specific interrupts and therefore clearly differentiate between kernel and user mode areas. Traditional memory protection (like pages marked as RO) are also not a barrier, since the attacker has access to physical memory in the first place.

Impact

Our approach amounts to exploiting a TOCTOU race-condition. The primitives used therein are 'blind' 'random', and thus constrained, but still extremely effective.

As explained in [19], the `type` of the primitive identifies the type of an exploit primitive: read, write or execute. The `property` of the primitive further describes the attack abilities associated with an exploit primitive type, such as location, timing, repeatability, etc. The five major primitive properties proposed by [20] are: arbitrary addresses (AA), arbitrary content (AC), arbitrary operation (AO), arbitrary number of times (AN) and at arbitrary time (AT).

In our case, a determined attacker still has arbitrary addresses (AA), arbitrary number of times (AN) and at arbitrary time (AT) capabilities. Here, arbitrary operation (AO) is not important when the arbitrary content (AC) primitive exists. The only added limitation is in the arbitrary content (AC); but as demonstrated above, when the arbitrary content primitive is removed, the attacker can use the arbitrary addresses (AA) and arbitrary number of times (AN) primitives together with the non-obliviousness (NO) (the ability to notice memory changes, even without knowing what the changes are) to achieve the arbitrary content (AC). Thus, a determined attacker achieves full control.

3 CONCLUSIONS

We presented a novel attack on encrypted data in use. We demonstrated how memory encryption for data in use, without authentication and replay protection, is insufficient against an attacker with a certain type of plausible, real-world control over a machine.

Our threat model is also based in the real-world; there are many potential insiders in a cloud provider who can ultimately deliver an exploit and implant chain in stages.

We showed how a determined attacker subverts a running Linux guest OS with only 2^{32} calls to a 'blind' 'random' corruption primitive. We compare this to a naive, "brute force" 2^{128} work estimate inappropriately borrowed from a standard cryptanalytic model for the authenticated encryption of data at rest.

Beyond the novel attack surface, we followed [1] and illustrated how choosing an appropriate cryptanalytic model for unauthenticated data in use is essential for assessing, and then protecting, the real world security of memory encryption.

Indeed, a real-world solution to this real-world hardware problem needs to be in hardware.

REFERENCES

[1] C. Bouillaguet, "Nice attacks — but what is the cost? computational models for cryptanalysis,"

[2] D. Weston, "Hardening with Hardware." Microsoft Blue Hat IL. Online; accessed 13-January-2018.

[3] R. R. Branco and S. Gueron, "Blinded Random Block Corruption Attacks," *HOST*, 2016.

[4] 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," *17th USENIX Security Symposium*, 2008.

[5] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," 2016.

[6] Intel Corporation, "Intel Architecture Memory Encryption Technologies Specification," 2017.

[7] R. Sevinsky, "Adventures in Thunderbolt DMA Attacks." Black Hat USA, 2013.

[8] X. K. T. Hudson, "Thunderstrike 2: Sith Strike." Black Hat USA, 2015. Online; accessed 12-January-2018.

[9] . Maartmann-Moe, "Inception Project." Online; accessed 12-January-2018.

[10] Intel Corporation, "Intel Virtualization Technology for Directed I/O," 2014. Online; accessed 31-October-2015.

[11] R. R. Branco and S. Gueron, "Memory protection challenges: Attacks on memory encryption," *Cyber Security: A Peer-Reviewed Journal*, 2016. Volume 1 / Number 3 / Winter 2017–18.

[12] B. Davis, P. Larsen, S. Volckadert, S. Winwood, D. Melski, M. Franz, and S. Magill, "Composition challenges for automated software diversity," tech. rep., Galois, Inc., 2018. Available at https://galois.com/reports/composition-challenges-for-automated-software-diversity/.

[13] S. Gueron, "Is your memory protected? attacks on encrypted memory and constructions for memory protection," 2016. Keynote. Hackers 2 Hackers Conference (H2HC).

[14] Arm Limited, "ARM Confidential Computing Architecture." Online; accessed 13-January-2018.

[15] Advanced Micro Devices, "AMD SEV-SNP: Strengthening VM Isolationwith Integrity Protection and More," 2020. Online; accessed 06-July-2021.

[16] Intel Corporation, "Intel trust domain extensions." Online; accessed 06-July-2021.

[17] S. Hong, "A Sound Mind in A Vulnerable Body:Practical Hardware Attacks on Deep Learning," *ENIGMA*, 2021.

[18] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 565–581, USENIX Association, Aug. 2016.

[19] R. Branco, K. Hu, H. Kawakami, and K. Sun, "A mathematical modeling of Exploitations and Mitigation Techniques using Set Theory." IEEE S&P Langsec Workshop 2018, 2018. Online; accessed 13-January-2019.

[20] PaX Team, "RAP: RIP ROP." Online; accessed 06-July-2021.