Blinded Random Block Corruption
Discussion on attacking memory encryption

Rodrigo Rubira Branco (@BSDaemon)
Chief Security Researcher
STORM (STrategic Offensive Research & Mitigations) Team
Intel Corporation
rodrigo.branco *@* intel.com

Shay Gueron
Senior Principal Engineer
University of Haifa
Amazon Web Services (AWS)

DISCLAIMER

We don’t speak for our employer. All the opinions and information here are of our responsibility

- So, mistakes and bad jokes are all
  - **OUR** responsibilities

Agenda

- Background
- A modern platform
- Is DRAM really vulnerable?
- Does encryption save the day?
- **What about encryption in the cloud?**

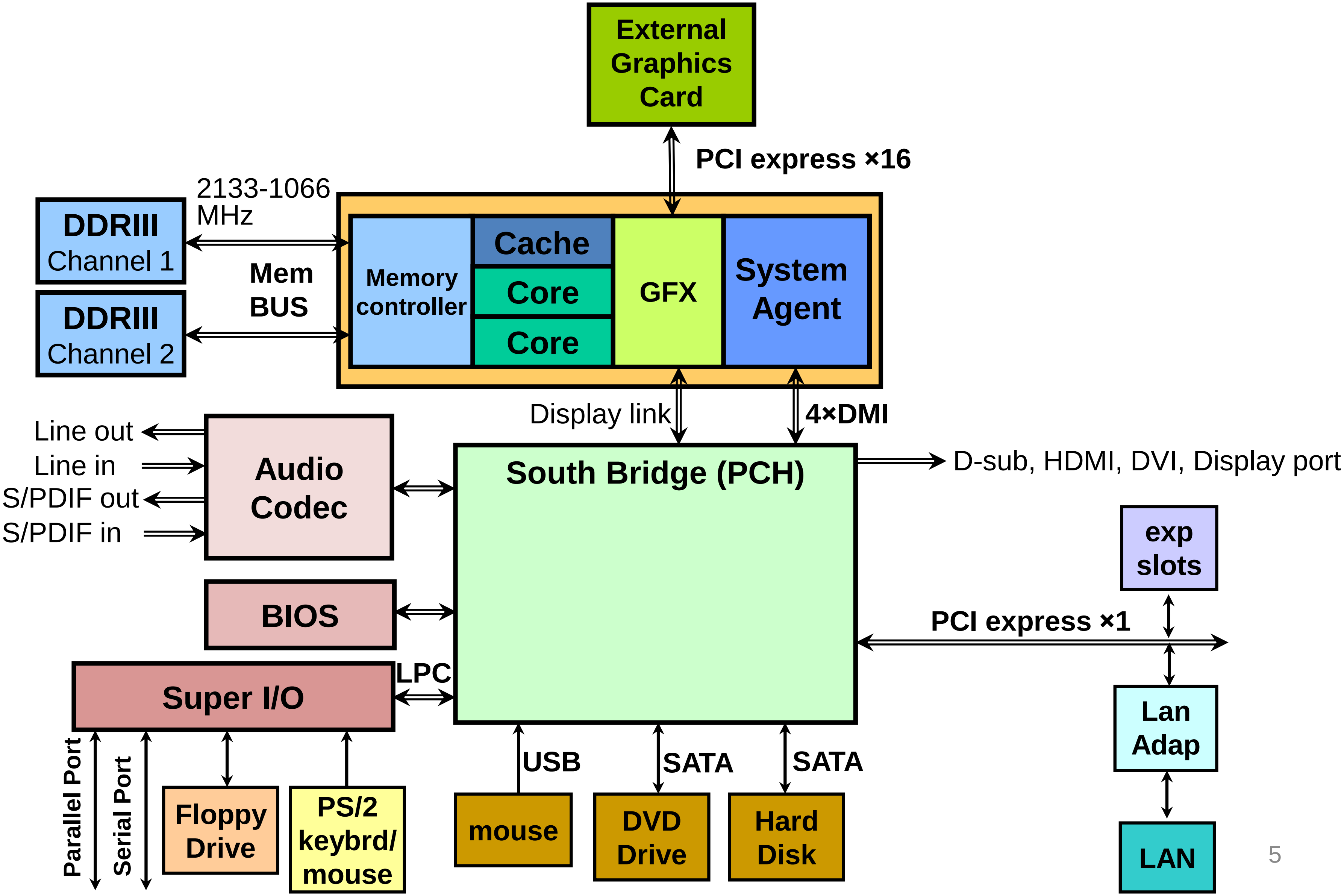
Background

Old news

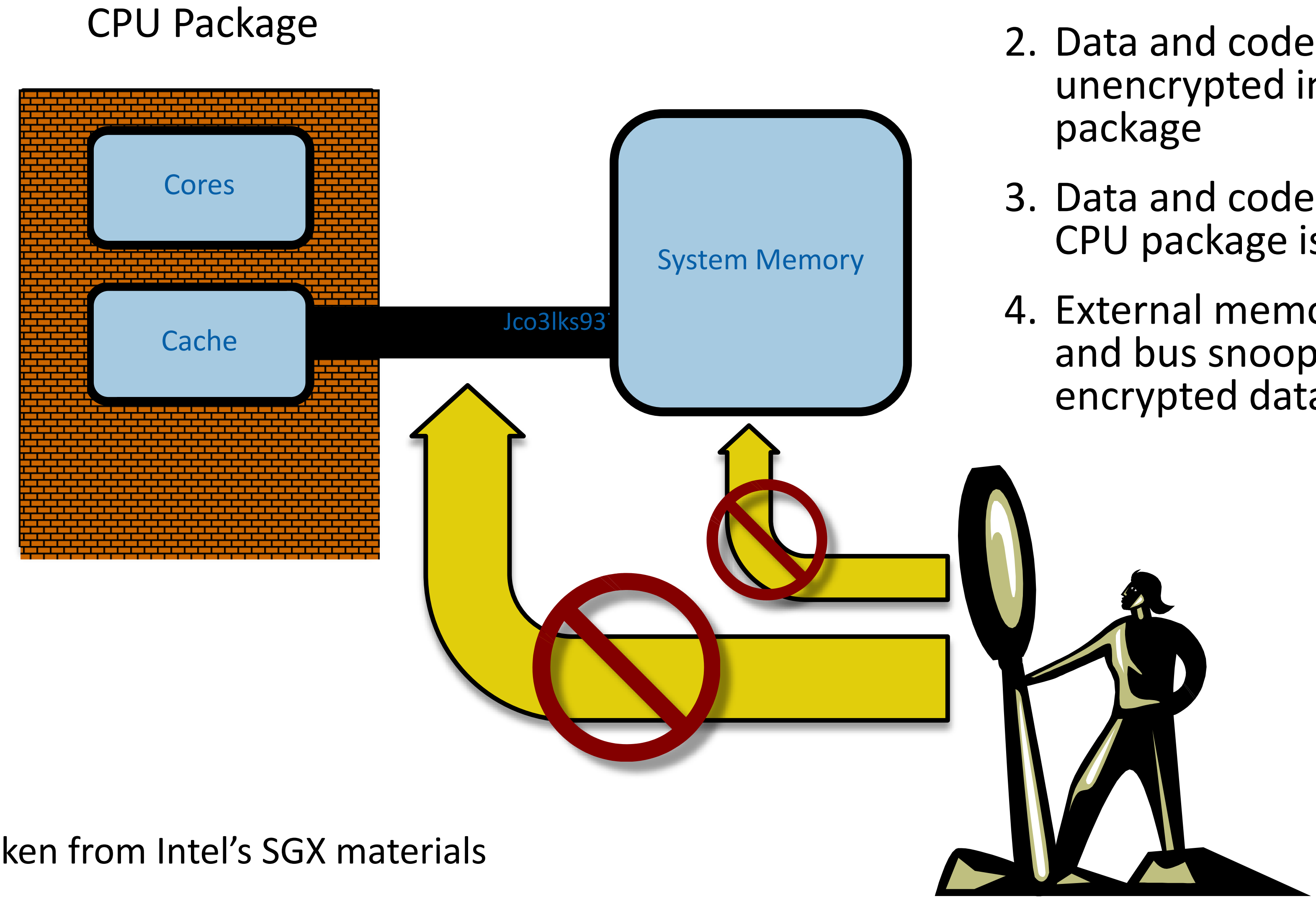
- Adversaries with physical access to attacked platform – are a concern
  - Mobile devices (stolen/lost)
  - Cloud computing (un-trusted environments)
- Read/write memory capabilities as an attack tool have been demonstrated
  - Using different physical interfaces
  - Thunderbolt, Firewire, PCIe, PCMCIA and new USB standards
- Consequences of DRAM modification capabilities
  - Active attacks on memory are possible
  - Attacker can change code / data **from any value to any chosen value**
  - **But this is too easy… right?**

Underlying attack assumption on the threat model:
The attacker has physical means to modify DRAM

A modern platform



1 image, 1000 words?



1. Security perimeter is the CPU package boundary
2. Data and code unencrypted inside CPU package
3. Data and code outside CPU package is encrypted
4. External memory reads and bus snoops see only encrypted data

Taken from Intel’s SGX materials

Technologies? - Disclaimer

- The list in the next slide are of technologies that use some kind of memory protection (with or without authentication, different encryption modes, etc)

- **We did not necessarily look into those technologies, therefore we are not claiming they are vulnerable, we are not saying they are comparable or even that they have similar purposes. The list is not comprehensive either. It is not showing in chronological order of creation or any kind of order. We just using them as examples of real cases of memory encryption technologies**

Technologies?

- Xbox
- Nintendo 3DS Security Processor (**did you read PoC || GTFO 14 already?**)
    - They discuss the possibilities of attacking encrypted code and the likelihood for the random corruption to create a valid (good for the attacker) opcode
    - Article: How likely are random bytes to be a NOP sled on ARM? By Niek Timmers and Albert Spruyt
- Apple Secure Enclave Processor
    - See the talk Demystifying the Secure Enclave Processor @ Black Hat 2016 by Tarjei Mandt and cia
- Intel MEE (Memory Encryption Engine)
- AMD SEV (Secure Encrypted Virtualization) and SME (Secure Memory Encryption)

Different attacker's tactics

- Passive attack: the attacker can only eavesdrop DRAM contents, but is not able to inject or interfere with it (in-use or not)
  - Non-existent in reality

- Active static attack: the attacker can read DRAM contents but cannot modify in-use/to-be-used (saved) DRAM
  - Example: cold boot attack
  - The attack is on the data privacy

- Active dynamic attacks: the attacker can read and modify DRAM contents that are in-use/to-be-used (saved)

The effectiveness of memory encryption without authentication is limited to active static attacks,
since the ability to modify in-use/to-be-used DRAM is assumed to be denied

Transparent memory encryption

- Some memory protection technologies against active dynamic attacks were proposed
  - Limiting the attacker’s physical ability to read/write memory
    - E.g., blocking DMA access in some scenarios
  - Memory encryption
- **Memory encryption using “transparent encryption” mode**
  - Simpler, cheaper, faster than “encryption + authentication”
  - Changes the assumptions on read/written memory capabilities of the attacker
  - Therefore, seems to be effective for limiting active dynamic attacks
- Memory encryption effects
  - Attacker has **limited control** on the result of active attacks
  - But the physical memory modification **capabilities remain available**
  - The attacker is **NOT OBLIVOUS** to DRAM changes

Underlying attack assumption: attacker has physical means to modify DRAM

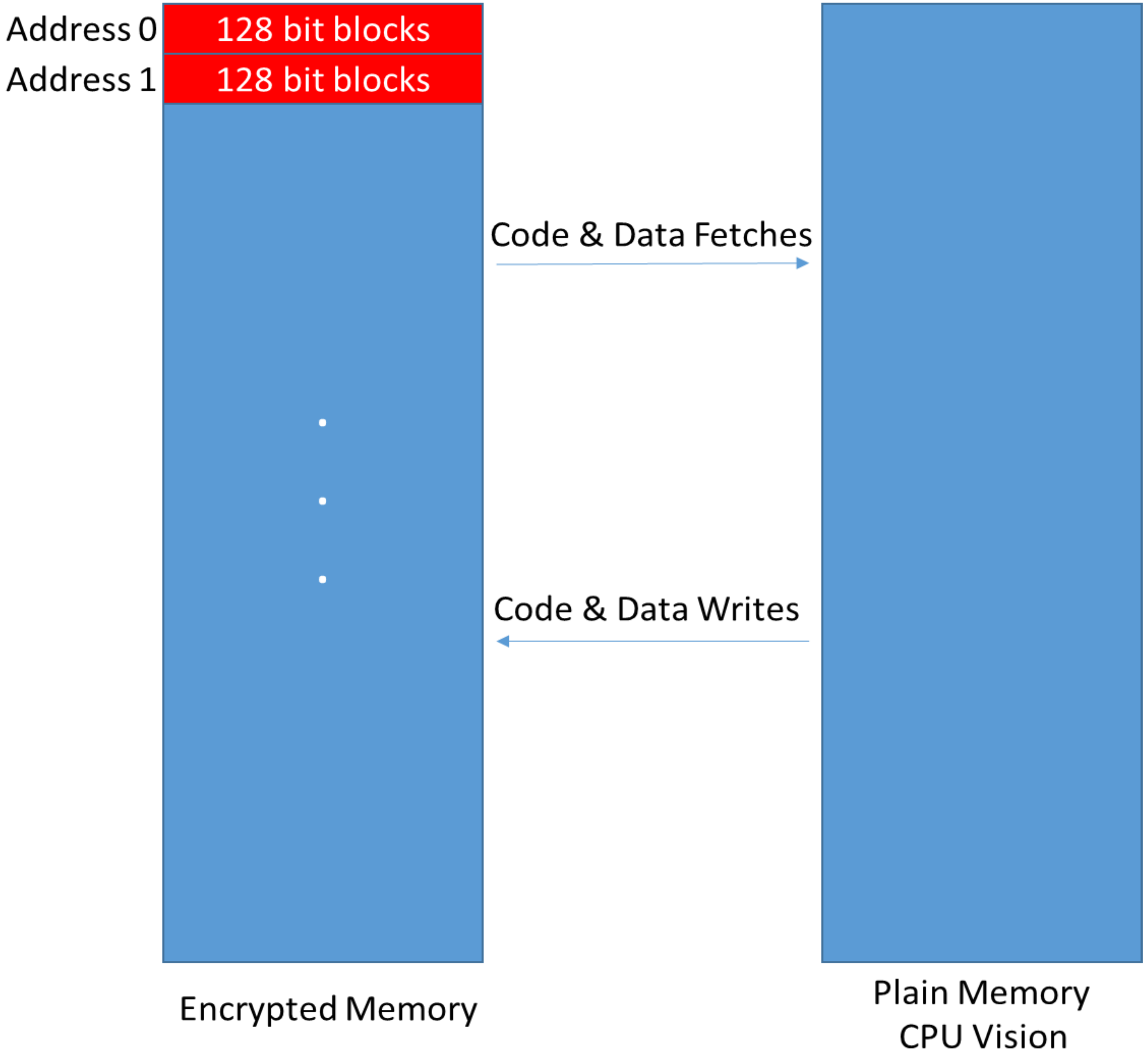
Blinded **R**andom **B**lock **C**orruption (**BRBC**)

- Under memory encryption, the attacker has limited capabilities
  - **B**linded **R**andom **B**lock **C**orruption (**BRBC**) attack
- (**Blinded**) The attacker does not know the plaintext memory values he can read from the (encrypted) memory.
- (**Random** (**Block**) **Corruption**) The attacker cannot control nor predict the plaintext value that would infiltrate the system when a modified (encrypted) DRAM value is read in and decrypted.
  - When using a block cipher (in standard mode of operation), any change in the ciphertext would **randomly corrupt** at least **one block** of the eventually decrypted plaintext

- **Question: does memory encryption (limiting the active dynamic attacker capabilities to BRBC only) provide a "good enough" mitigation in practice?**

Underlying attack assumption: attacker has physical means to modify DRAM

We will re-visit this image later



Address 0

128 bit blocks

Address 1

128 bit blocks

Code & Data Fetches

Code & Data Writes

Encrypted Memory

Plain Memory
CPU Vision

We are showing addresses in the encrypted memory as block locations only to facilitate the discussion.

Image assuming block size of 128 bits (this is obviously dependent on the cipher)

Threat Model
SW x HW initiated attacks?

- A threat model needs to be realistic (attackers do not follow written rules)

- SW initiated attacks might seem out of scope (because they are inside the encryption boundary)
  - But, HW initiated attacks do change SW behavior in unexpected ways -> that is in scope!

- So, SW needs to be considered… and it is complex, big… full of assumptions…
  - Encryption without authentication/integrity obviously do not offer integrity, but does it keep the confidentiality promises when the data is in-use?

We will show that…

  - Despite limited capabilities, dynamic active attacks are possible
  - Encryption-only does not offer a defense-in-depth mechanism against arbitrary memory overwrites without removing capabilities assumptions

- The BRBC attacker is able to create Time-of-check/Time-of-use (TOCTOU) race conditions all around the execution environment
  - Usual control-flow hijacking attacks require precise pointer control to redirect flow of execution. Usual DMA attacks perform precise code modification
  - Data-only attacks caused by a BRBC attacker can be induced after some code checks, therefore cause TOCTOU races that invalidate the results of such checks
  - Unexpected computation (and flows) can emerge (since code is driven by its input data)
    - Data-only based attacks, thus partial control flow enforcement can't prevent

Underlying attack assumption: attacker has physical means to modify DRAM

The A-B-C attacker model

- **A**ccess Seeking Attacker

This attacker is not the owner of the platform, but got it to his possession, in a locked state. He wishes to get an user access, in order to steal the data on the system.

- **B**reaching Attacker

This attacker is a legitimate user of the platform, who wishes to breach some of the system’s policies or circumvent restrictions on his privileges.

- **C**onspirator Attacker

This attacker is also a legitimate user of the platform/environment. He has administrative powers and conspires to collect other users’ data.

Underlying attack assumption: attacker has physical means to modify DRAM

Becoming "root" on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming "root" on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming "root" on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();

        auth_ok:
                return;
}
```

BRBC Attack to the preauth_flag

Becoming “root” on a locked system with a BRBC attack

```
global var1…varn
global preauth_flag
global preauth_related
code_logic() {
        if (preauth_enabled) {
                call_preauth_mechanism() -> sets preauth_flag if successful
        }
 repeat_auth:
        if (preauth_flag) goto auth_ok;

        authentication_logic();        -> THIS NEVER GETS EXECUTED!

        auth_ok:
                return;
}
```

TOCTOU (Time-of-use/Time-of-check) Race Condition

- This was caused by our arbitrary memory write (the BRBC)
- The corrupted values adjacent to the preauth_flag were not used at the corruption time (thus the block corruption is not a problem)
- The check for the preauth_flag only checks for not 0 (thus we don't need to control the exact value)

- But how do we win the race?
  - In this case, quite simple: We just cause the authentication to fail at the first time (when it does ask the password)
    - The system waits for the password prompt
    - We cause the corruption and input invalid password
    - The authentication fails and the logic is repeated, but this time with the corruption!

Previous Experiments

- Two demonstrations showed the underlying attack assumptions
  - A debugger to make it easy to step through and see the corruption effect
  - The JTAG to demonstrate the physical addresses are not a concern
  - Details released at HOST 2016 (IEEE International Symposium on Hardware Oriented Security and Trust)

- SW mitigations are not feasible because the attacker has lots of possibilities for targets (not only ! 0 comparisons). Some examples:
  - If an attacker overwrites the NULL terminator of a string, he can generate buffer overflows, memory leaks
  - If an attacker overwrites an index, he can generate out-of-bounds writes, that might lead to user-mode dereferences if in kernel-mode context
  - If an attacker overwrites a counter, he can generate REFCOUNT overflows, leading to use-after-free conditions

Underlying attack assumption: attacker has physical means to modify DRAM

Different Attack Scenarios and Targets

- Attacker with user privileges on the machine
  - Higher control/visibility of the memory space
  - Tries to bypass security policies
    - Local administrator (common on cloud-based scenarios)

- All system software/components can be seem as targets
  - We just demonstrated in a highly-limited scenario (locked machine, unknown software running, little to no information on the OS details)

- As more interactions with the system, as bigger is the scope of possible attack targets (as discussed previously)

What about the cloud scenario?
Hypervisor has management interfaces

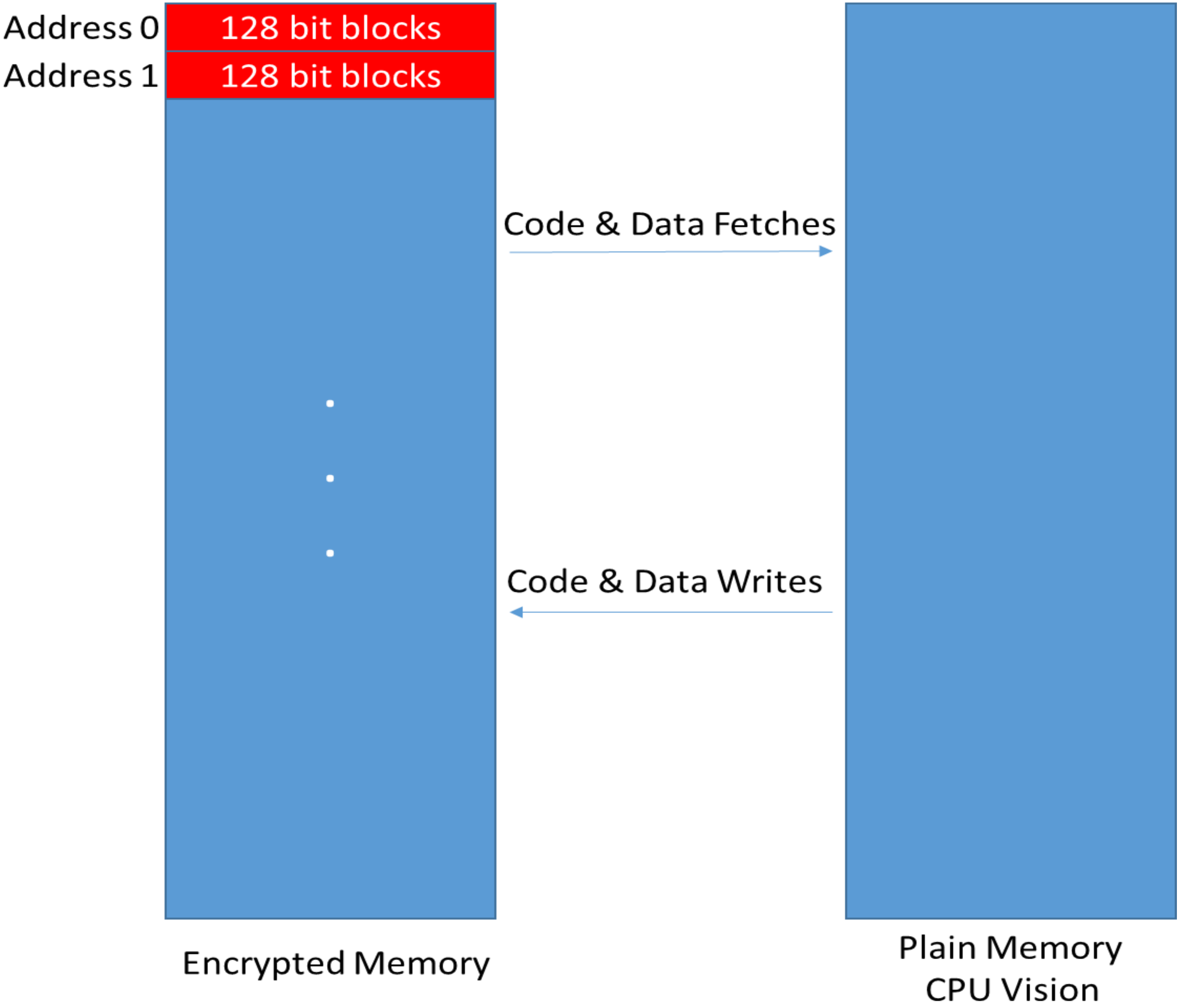
- VM Introspection capabilities exist for legitimate security reasons
  - Inspect inside guest VMs, to auto-configure network elements, to distribute resources
- The same capabilities can be “abused” by a malicious administrator (even in the presence of a trusted hypervisor)
  - Or vulnerabilities can provide similar primitives (read/write outside of the encryption context)
- Memory encryption of guest machines remove the ability of administrator to snoop into the VM’s memory
  - A different key per-VM is necessary, to avoid replay attacks with known plaintext/ciphertext in another VM fully controlled by the attacker
  - CPU control through introspection is similar to JTAG control (flow changes can be performed without a BRBC attack)
  - BRBC attack might be more reliable in scenarios where multiple connections are made to the machine (like in a server scenario)

Memory encryption with VM-unique keys
The threat model

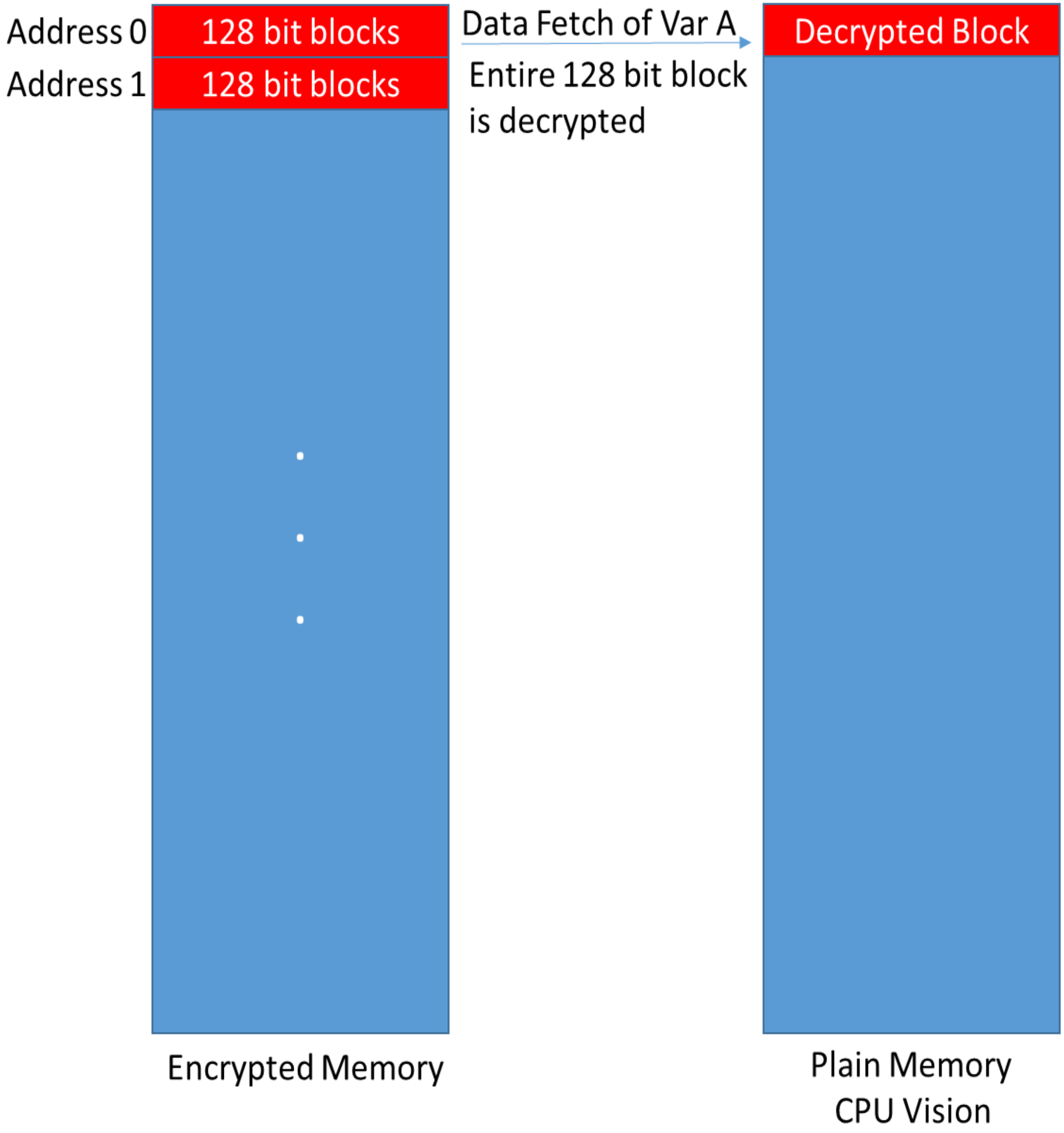
- Cloud service provider hosts multiple customers' VM's
- But users do not necessarily trust this remote environment
  - An operator at the cloud provider's facility can use the hypervisor's capabilities to read any VM's memory
- Assumption: the hypervisor is trusted (else – game over)
    - Measured hypervisor
- Memory encryption
  - Each guest VM has encrypted memory space using a unique (per-VM) key
  - Hypervisor capabilities remain, but:

    Since memory is encrypted with a VM-unique key, the user's data privacy is protected… or, is it?

But this was once again the !0 comparison case



Block corruption independently randomly corrupt each element on the block



Address 0 128 bit blocks

Address 1 128 bit blocks

Data Fetch of Var A

Entire 128 bit block is decrypted

Decrypted Block

Encrypted Memory

Plain Memory
CPU Vision

Showing addresses as block locations, to facilitate the discussion.

Var A:

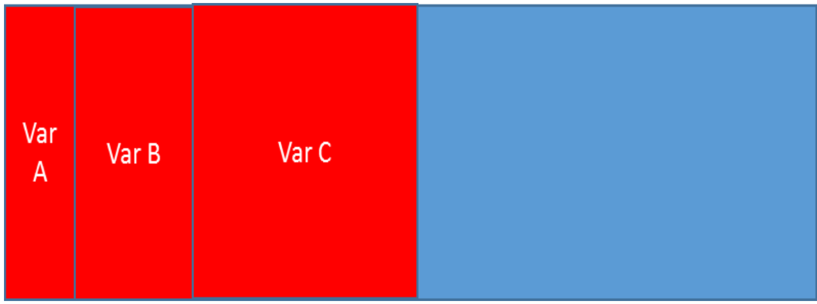
For example, an unsigned char, an 8 bits type

Var B:

For example, an unsigned short, 16 bits type

Var C:

For example, an unsigned int, 32 bits type



Decrypted Block

Corrupting var A, also corrupts Var B and C (bits corrupted randomly/independently as a property of the block cipher)

Introducing:
Feasible Brute-force-based BRBC

- If we find a way to brute-force a block that has this characteristics
  - Many different data elements, with different sizes
  - One of those elements being of interest, and small enough to be fully brute-forced (like a 32 bit integer)
    - And for which we are able to tell if we somehow have a value we want
  - In which the other elements, if changed, do not affect our interests as an attacker
  - And for which any value would not affect the system stability (meaning: we can repeat the corruption as many times as we want)

- Then we are able to
  - Have a fully controlled memory overwrite! (we just need to brute-force the element of interest til it randomly has the value of our interest!)

Can we make a pie with so many ingredients? *

- Linux Kernel manages processes using a data structure named task_struct
- Such struct has lots of elements necessary to store the process information, such as memory areas, opened files, privileges and so on
- For privileges, it uses a pointer to another data structure, which is the credentials... Having a look at it, we have something quite interesting

* Homage to a famous quote by *Noir*

Sounded like impossible?
A bit on the Linux Kernel…

Following the task_struct of a process (to find our target), we see there is a process credentials entry, which is a structure that has many elements, of interest we have:

```
kuid_t        uid;          /* real UID of the task */
kgid_t        gid;          /* real GID of the task */
kuid_t        suid;          /* saved UID of the task */
kgid_t        sgid;          /* saved GID of the task */
kuid_t        euid;          /* effective UID of the task */
kgid_t        egid;          /* effective GID of the task */
kuid_t        fsuid;         /* UID for VFS ops */
kgid_t        fsgid;         /* GID for VFS ops */
```

Notice that kuid_t and kgid_t are typedef’s to __kernel_uid32_t and __kernel_gid32_t, which in turn:

```
typedef unsigned int    __kernel_uid32_t;
typedef unsigned int    __kernel_gid32_t;
```

Is alignment inside the target block an issue?
No, since the adjacent elements do not matter for the attack

Option 1
euid is last element of a block

gid	suid	sgid	euid	egid	fsuid	fsgid	-
Block 1				**Block 2**			

Affects: gid, suid, sgid (everything before it)

Option 2
euid is middle element of a block

-	uid	gid	suid	sgid	euid	egid	fsuid
Block 1				**Block 2**			

Affects: sgid, egid, fsuid (max 2 before and two after it)

Option 3
euid is first element of a block

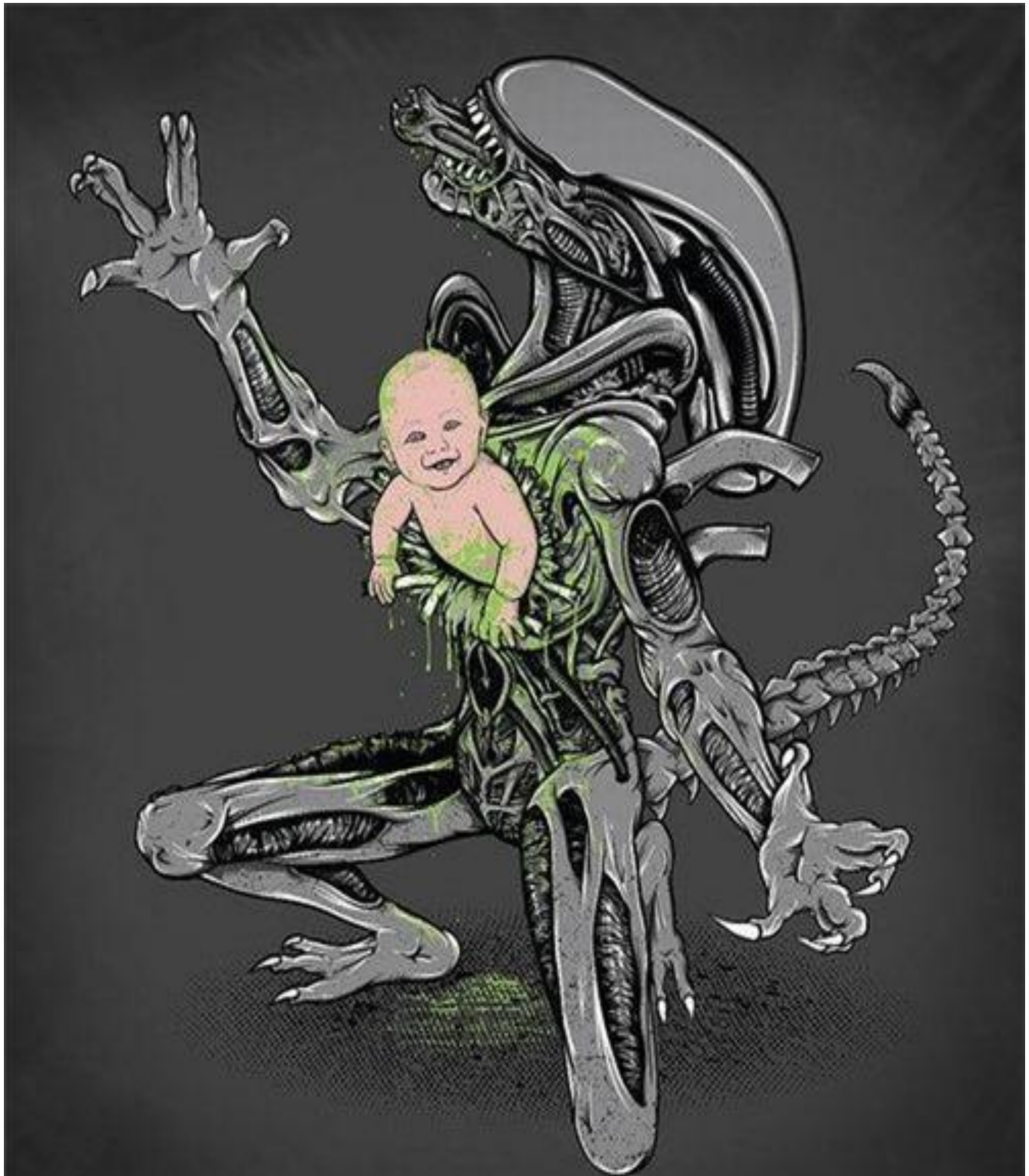
uid	gid	suid	sgid	euid	egid	fsuid	fsgid
Block 1				**Block 2**			

Affects: egid, fsuid, fsgid (everything after it)

Do we have a winner candidate?

Premisse	Does it satisfy?
Many different data elements, with different sizes	Yes
One of those elements being of interest, and small enough to be fully brute-forced (like a 32 bit integer)	Yes (euid is our target)
And for which we are able to tell if we somehow have a value we want	Yes (our target process has elevated privileges)
In which the other elements, if changed, do not affect our interests as an attacker	Yes (we can change other elements if needed with the privileges)
And for which any value would not affect the system stability (meaning: we can repeat the corruption as many times as we want)	Yes

Who is the founder of this little hacker?



What about limitations?

- We need to be able to locate such a data structure in memory (we are blind to the memory contents)
  - There are a lot of challenges to being able to do that (and system pressure might affect the ability of doing it)

- For now, our PoC requires a process running on the target (with no privileges)
  - We elevate that process' privilege
  - The requirement for such a process is exactly to avoid the limitation (given we have a process we control, we use such a process to spawn multiple child process and create a predictible memory layout and pattern that we can identify – that is how we locate the correct structure in memory)
  - We are studying other possibilities (like for server-side process that anyhow spawn child processes, and others)

Mitigation Techniques

- Hibernation when used together with proper disk encryption
- VT-d/IOMMU and PMRs
  - Limits DMA capabilities exposed
  - Might not be enough against certain attackers (that have physical access) and in some platforms (only effective if the attack requirement is fully removed)
- Software self-protection (or control flow enforcement technologies)
  - Attack uses valid flows with invalid data (data-only attack) bypassing CET
  - Different attack targets make software hardening inviable
- Memory encryption with Authentication
  - Able to detect the arbitrary change and prevent the attack
  - What about replay attacks?
- Intel SGX (Software Guard eXtensions)
  - MEE currently employ authentication and replay protection

Summary and conclusions

- Hierarchical model of the A-B-C attackers
- Formalization the notion of **BRBC** attack
- Demonstration of a BRBC attack
- **Encryption-only by itself is not necessarily a “good enough” defense-in-depth mechanism against arbitrary memory write primitive**

- Dilemma: What is easier/viable:
  - Remove ***ALL*** cases of arbitrary writes for ***ALL*** platforms the technology would support (which would depend on integration teams capabilities to guarantee that)
  - Or support encryption with authentication

The end! Is it !?

Questions?

Rodrigo Rubira Branco (@BSDaemon)
Chief Security Researcher
STORM (STrategic Offensive Research & Mitigations) Team
Intel Corporation
rodrigo.branco *@* intel.com

Shay Gueron
Senior Principal Engineer
University of Haifa
Amazon Web Services (AWS)

Summary and conclusions

- Hierarchical model of the A-B-C attackers
- Formalization the notion of **BRBC** attack
- Demonstration of a BRBC attack
- **Encryption-only by itself is not necessarily a “good enough” defense-in-depth mechanism against arbitrary memory write primitive**

- Dilemma: What is easier/viable:
  - Remove ***ALL*** cases of arbitrary writes for ***ALL*** platforms the technology would support (which would depend on integration teams capabilities to guarantee that)
  - Or support encryption with authentication

Mitigation Techniques

- Hibernation when used together with proper disk encryption

- VT-d/IOMMU and PMRs
  - Limits DMA capabilities exposed
  - Might not be enough against certain attackers (that have physical access) and in some platforms (only effective if the attack requirement is fully removed)

- Software self-protection (or control flow enforcement technologies)
  - Attack uses valid flows with invalid data (data-only attack) bypassing CET
  - Different attack targets make software hardening inviable

- Memory encryption with Authentication
  - Able to detect the arbitrary change and prevent the attack
  - What about replay attacks?

- Intel SGX (Software Guard eXtensions)
  - MEE currently employ authentication and replay protection