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

LangSec 2024

Shay Gueron & Rodrigo Branco (BSDaemon)
Meta/Haifa University & Meta/Oregon State University

What is **Confidential Computing?**

- A marketing term, that is loosely defined **on purpose**

- I rather discuss what does Confidential Computing imply for customers, end-users and overall non-experts?

- AWS has a nice take on Confidential Computing that is beyond jargons

PERCEPTION IS REALITY

Can we *NOT* trust our Cloud Provider?

- A big take on Confidential Computing is that the trust is divided, between the cloud provider and the HW manufacturer… **is it?**

- **Your cloud provider is a your HW provider… even if we ignore very nuanced bugs [1] [2]**
  - **For example how do you know that there is no memory interposer? [3]**
  - **Or special commands to the memory controller? (none of that is part of 'attestation')**

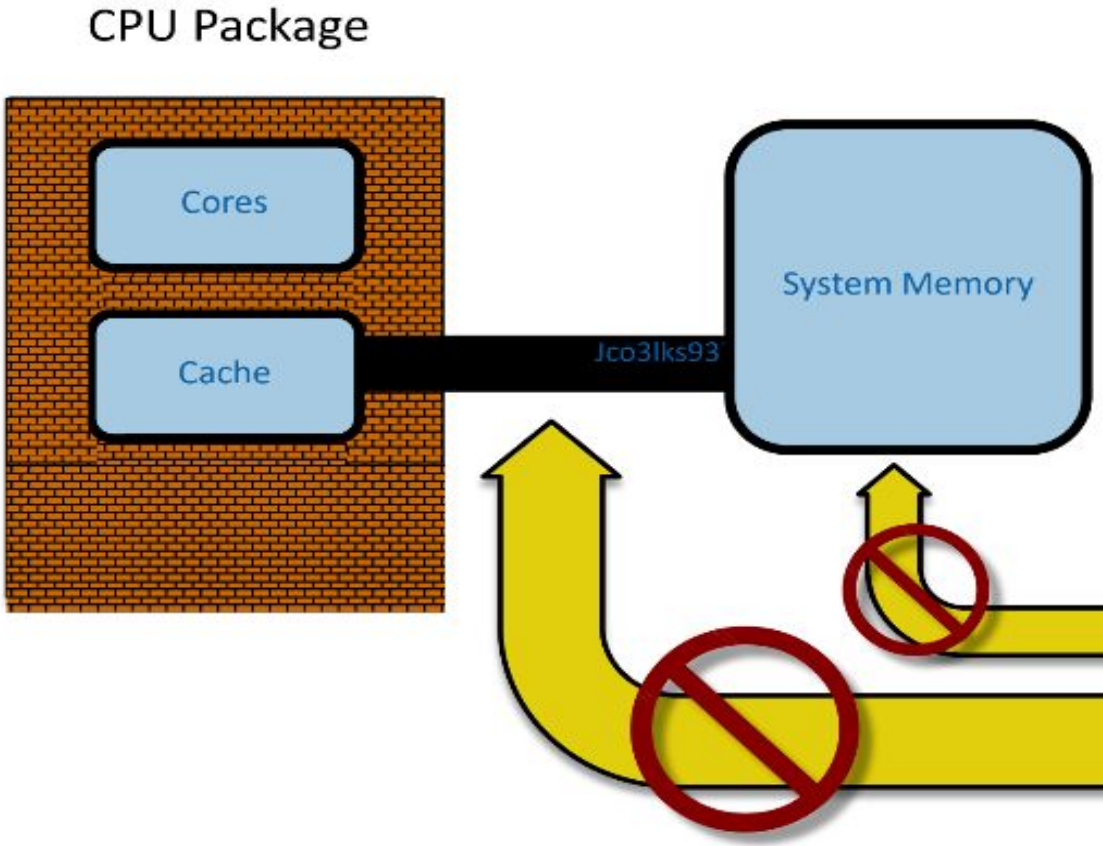
[1] Hyperbleed. Link: Current state of spectre-BTI mitigations on cloud

https://blog.gris.dcc.ufrj.br/en/blog/2023-05-29-hyperbleed-post/

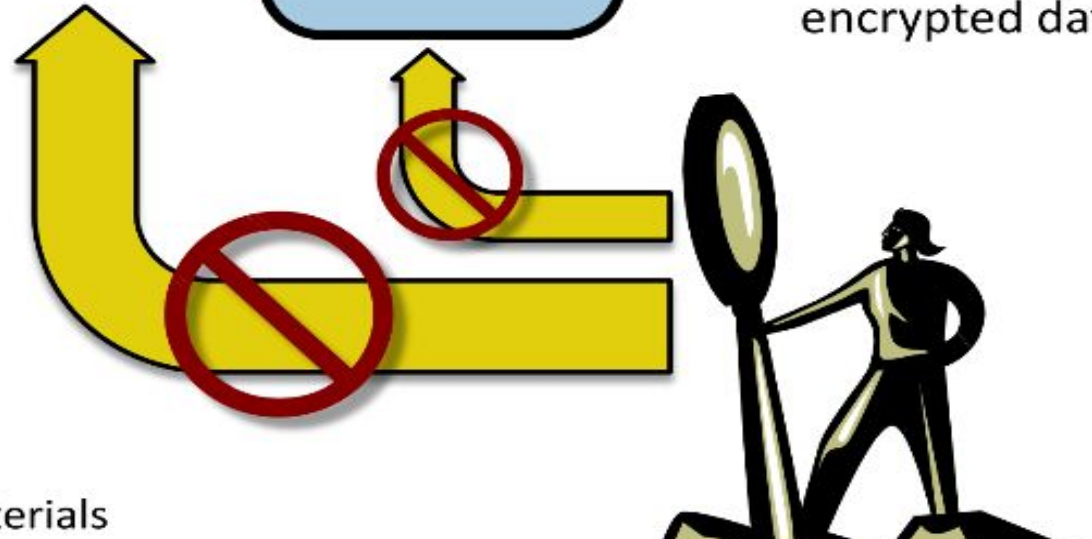
[2] CVE 2023-0045. Link: https://nvd.nist.gov/vuln/detail/CVE-2023-0045

[3] Taking DMA Attacks to the Next Level. Link:
https://www.blackhat.com/docs/us-17/wednesday/us-17-Trikalinou-Taking-DMA-Attacks-To-The-Next-Level-How-To-Do-Arbitrary-Memory-Reads-Writes-In-A-Live-And-Unmodified-System-Using-A-Rogue-Memory-Controller.pdf

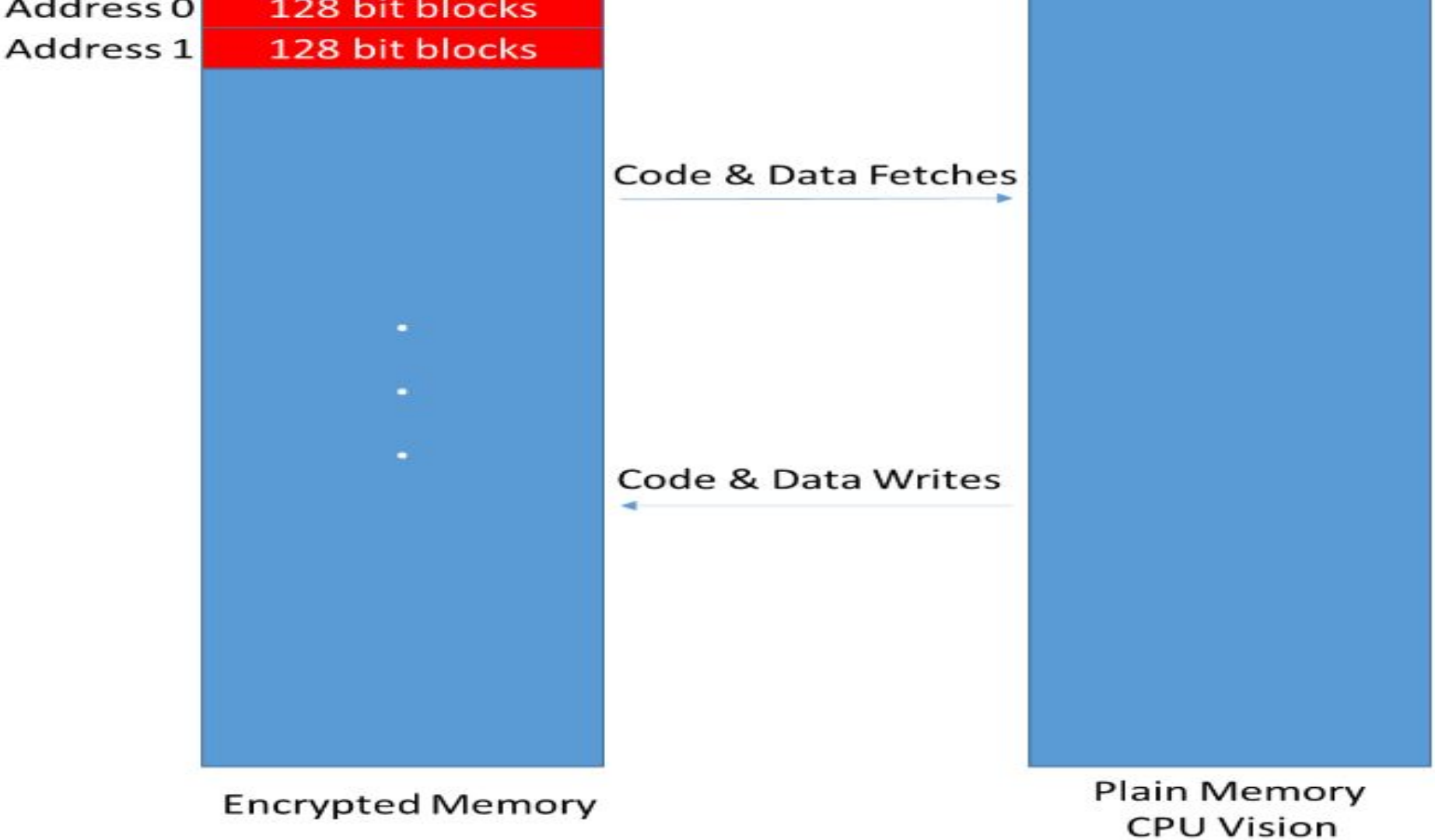
Memory Encryption



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


Address 0
128 bit blocks
Address 1
128 bit blocks
Code & Data Fetches
Code & Data Writes
Encrypted Memory
Plain Memory
CPU Vision

What does encrypting the memory changes for an attacker?

- The attacker is not able to read the memory contents directly
- Attacker changes to the memory result in ‘unpredictable’ (random) changes of entire blocks (memory encryption schemes use block ciphers)

Both benefits from the security perspective assume the attacker has the ability to read and write memory, somehow (otherwise, there is no need for encryption in the first place)

But memory is not like a stored file

- Memory is in-use, has context, imply all kinds of SW stacks
- Memory encryption also does not create obliviousness: The attacker is still able to observe that memory areas (blocks) are changing (albeit not knowing the specific contents of those changes)

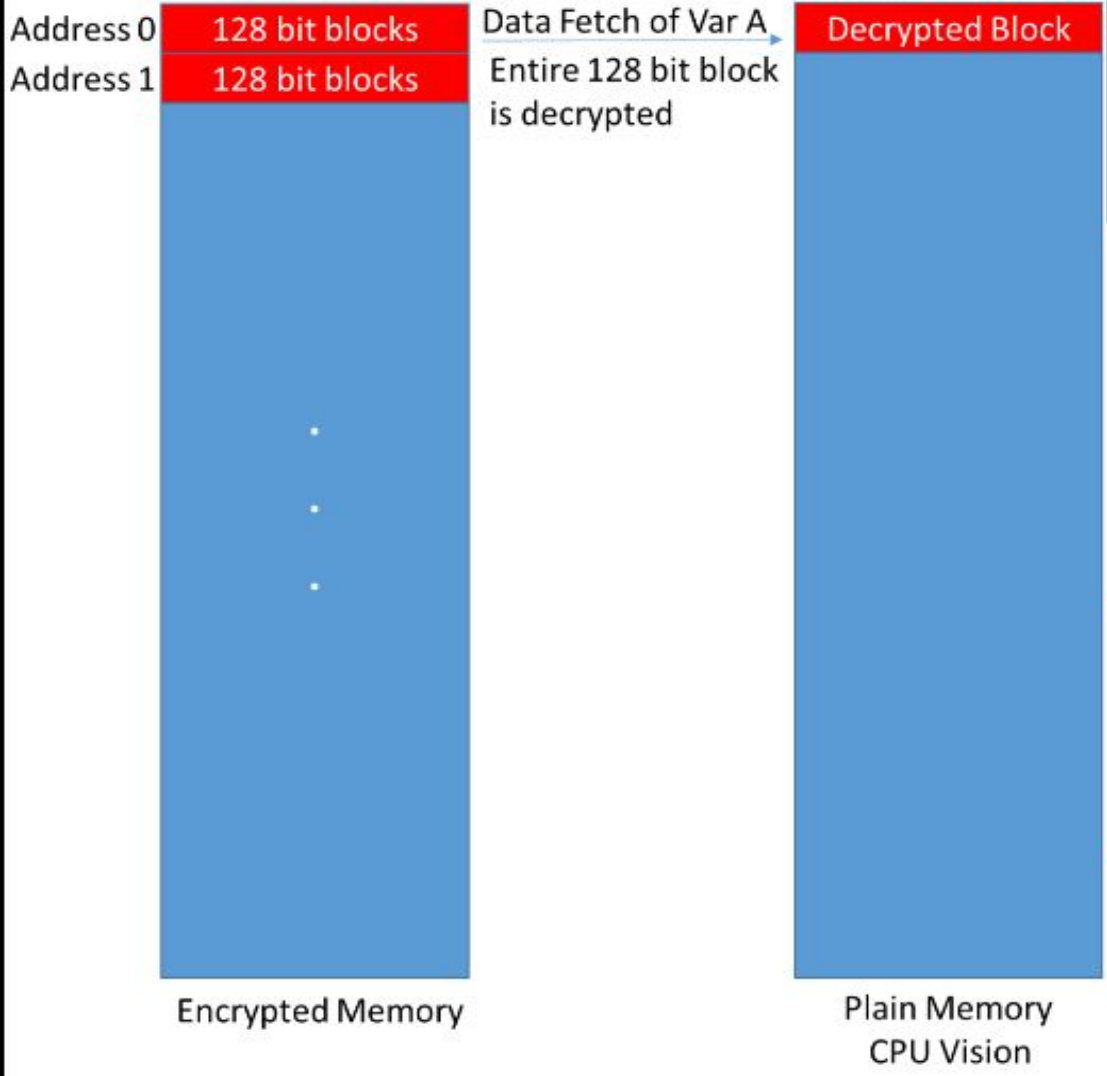
- So we can say that the attacker is:
  - **Blinded**: Does not have the plaintext value for any (**most**?) memory locations
  - And can only cause a **Random (Block) Corruption** (the attacker has no control over the plaintext that would be ‘consumed’ by the system after a modification of an encrypted memory area (and no matter the size of the modification, the impact causes an entire block to change)
  - But the change is in a **controlled** location (block-aligned)

Non-obviousness

- There is really an uncommon **primitive** to discuss (arguably it only exists because of memory encryption)
- Why it is important?
  - Let’s imagine a scenario of the malicious (cloud/system) administrator
  - They can use different means to read/write VM/guest memory (like debug stubs, HW JTAG ports, etc)
  - Memory encryption means that such individual now is prevented from accessing that VM (and its secrets - assuming all other building blocks work properly, such as the image encryption, attestation, etc)
  - That administrator though is still able to interact with the system via its valid ways (like ssh to the system, browsing a page it hosts, etc). Maybe even interacting with the login prompt of that system
  - And observe the encrypted memory changing (so while the admin is not able to know the contents, they can analyze memory changes versus interactions with the system to learn about locality!)

Example target : if (!0) data-only attacks

```
global var1…varn
global preauth_flag
global preauth_related
code_logic() {
 if (preauth_enabled) call_preauth_mechanism() // if successful, sets preauth_flag
 repeat_auth:
        if (preauth_flag) goto auth_ok
         authentication_logic(); // goes to repeat_auth in case of failure
 auth_ok:  return true;
}
```



Showing addresses as block locations, to facilitate the discussion.

Var A:

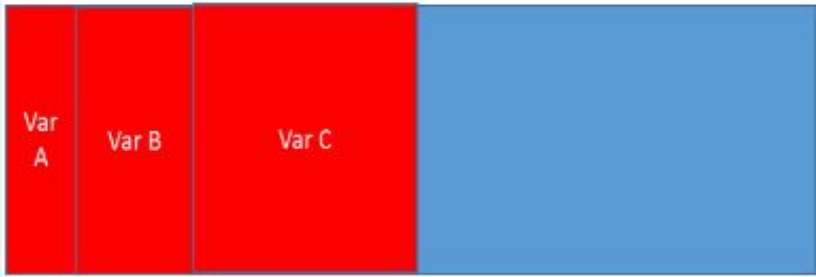
For example, an unsigned char, an 8 bits type

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)

Yeah, obviously… so what?

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!)

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;
```

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

Conclusions

- There is no confidential computing in a general purpose platform in 2024 -> you do have to trust the hardware/infrastructure provider

- Memory encryption is an interesting build block, but it brings with it other challenges that have to be considered

- We want to see the world continue developing the technology, we see more (immediate) importance for it in the desktop market than in the server market

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

LangSec 2024

Shay Gueron & Rodrigo Branco (BSDaemon)
Meta/Haifa University & Meta/Oregon State University