24. Hardware Security (Meltdown, Spectre, TEEs)

Blase Ur and David Cash

March 4th, 2020

CMSC 23200 / 33250

THE UNIVERSITY OF CHICAGO

Hardware Security: A Broad View

What do we trust?

How do we know we have the right code?

Recall software checksums, Subresource Integrity (SRI)

What is our root of trust? Can we establish a smaller one?

Can we minimize the Trusted Computing Base (TCB)?

Can processors' designs for efficiency lead to insecurity?

Yes!

Trusted Platform Module (TPM)

Standardization of cryptoprocessors, or microcontrollers dedicated to crypto functions w/ built-in keys

Core functionality:

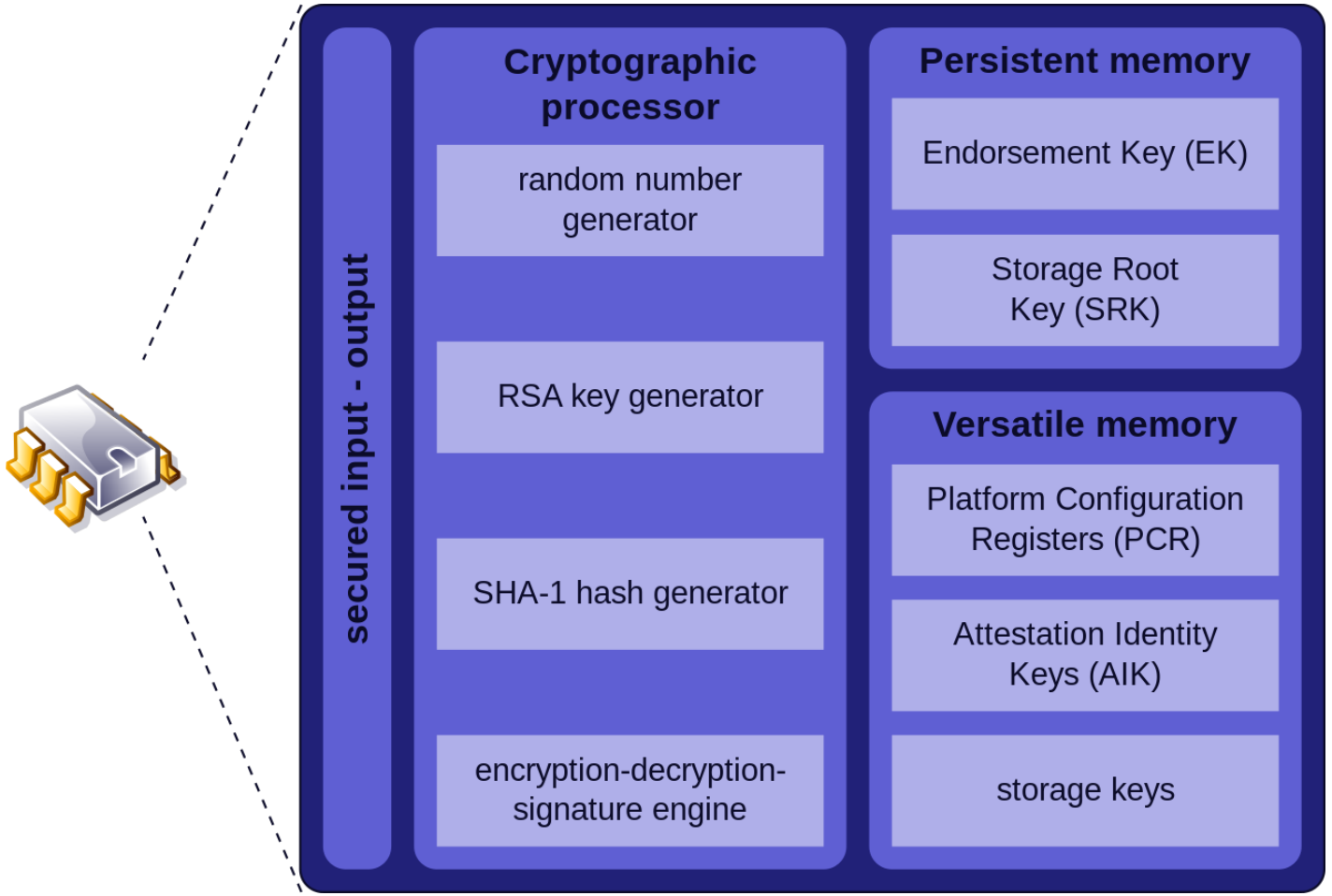
1) Random number generation, crypto key creation

2) **Remote attestation** (hash hardware and software config and send it to a verifier)

3) **Bind/seal** data: encrypted using a TPM key and, for sealing, also the required TPM state for decryption

Uses: DRM, disk encryption (BitLocker), authentication

Trusted Platform Module (TPM)



Trusted Execution Environment (TEE)

TPMs are standalone companion chips, while TEEs are a secure area of a main processor

Guarantees confidentiality and integrity for code in TEE

Key example: Intel Software Guard Extensions (SGX)

***Enclaves* =** Private regions of memory that can't be read by any process outside the enclave, even with root access

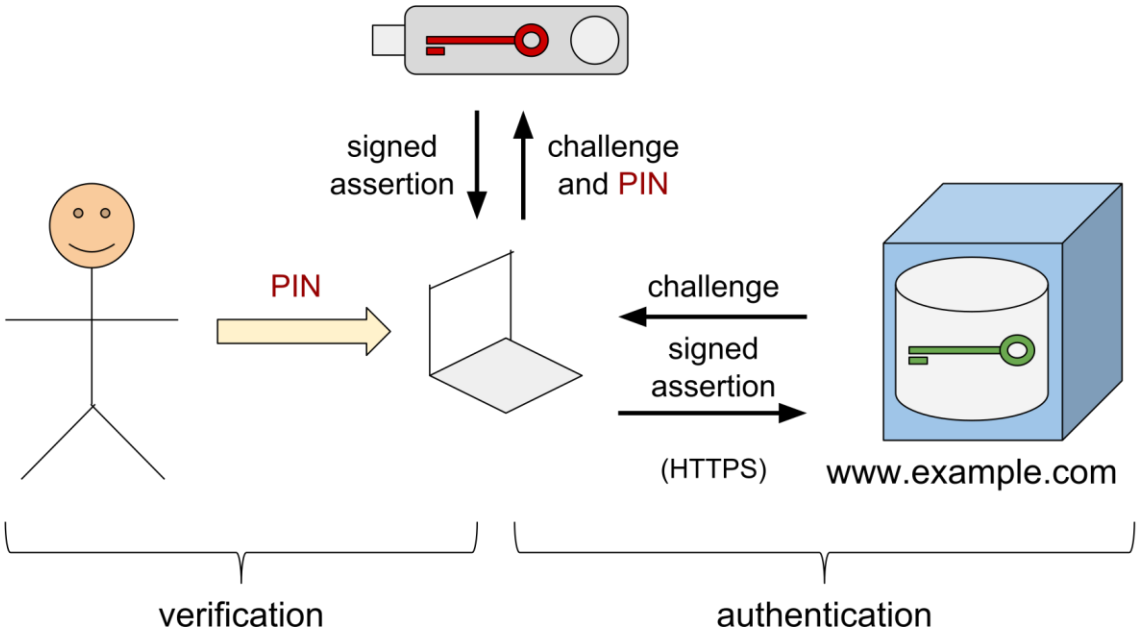
Uses: DRM, mobile wallets, authentication

Case Study: WebAuthn

Created under the FIDO2 project, now a W3C standard

Goal: Authenticate to web apps using public-key crypto

Implemented in specialized hardware OR in software using a TPM/TEE

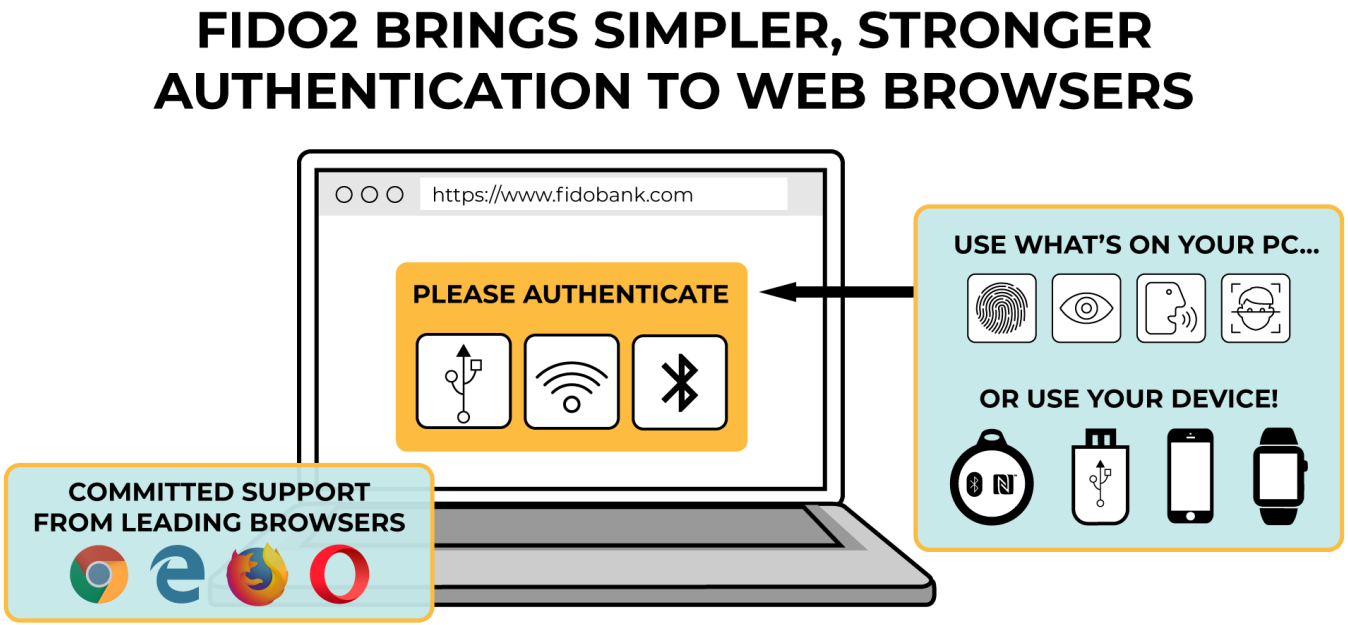


Case Study: WebAuthn

User interaction: Push a button on a key, type a PIN into the device, present biometric (fingerprint) to hardware reader

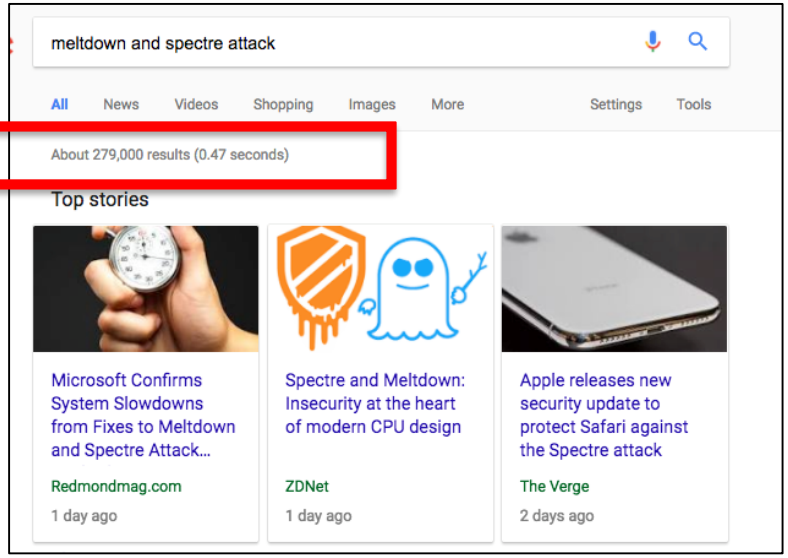
Case Study: WebAuthn

ds



Attacks that exploit processor vulnerabilities

- Can leak sensitive data
- Relatively hard to mitigate
- Lots of media attention



Relevant Ideas in CPUs

Memory isolation: Processes should only be able to read their own memory

- Virtual (paged) memory
- Protected memory / Protection domains

CPUs have a (relatively small) very fast cache

- Loading uncached data can take >100 CPU cycles

Out-of-order execution: Order of processing in CPU can differ from the order in code

- Instructions are much faster than memory access; you might be waiting for operands to be read from memory
- Instructions **retire** (return to the system) in order even if they executed out of order

Relevant Ideas in CPUs

There might be a conditional branch in the instructions

Speculative execution: Rather than waiting to determine which branch of a conditional to take, go ahead anyway

- **Predictive execution**: Guess which branch to take
- **Eager execution**: Take both branches

When the CPU realizes that the branch was mis-speculatively executed, it tries to eliminate the effects

A core idea underlying Spectre/Meltdown: The results of the instruction(s) that were mis-speculatively executed will be cached in the CPU [yikes!]

Example (Not bad)

Consider the code sample below. If `arr1->length` is uncached, the processor can speculatively load data from `arr1->data[untrusted_offset_from_caller]`. This is an out-of-bounds read. That should not matter because the processor will effectively roll back the execution state when the branch has executed; none of the speculatively executed instructions will retire (e.g. cause registers etc. to be affected).

```
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...;
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset_from_caller];
 ...
}
```

https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

Example (Bad!!!)

- load `value = arr1->data[untrusted_offset_from_caller]`
- start a load from a data-dependent offset in `arr2->data`, loading the corresponding cache line into the L1 cache

```
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
/* >0x400 (OUT OF BOUNDS!) */
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset_from_caller];
 unsigned long index2 = ((value&1)*0x100)+0x200;
 if (index2 < arr2->length) {
   unsigned char value2 = arr2->data[index2];
 }
}
```

After the execution has been returned to the non-speculative path because the processor has noticed that `untrusted_offset_from_caller` is bigger than `arr1->length`, the cache line containing `arr2->data[index2]` stays in the L1 cache. By measuring the time required to load `arr2->data[0x200]` and `arr2->data[0x300]`, an attacker can then determine whether the value of `index2` during speculative execution was 0x200 or 0x300 - which discloses whether `arr1->data[untrusted_offset_from_caller]&1` is 0 or 1.

https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

Spectre: Key idea

Use branch prediction as on the previous slide

Conducting a timing side-channel attack on the cache

Determine the value of interest based on the speed with which it returns

Spectre allows you to read any memory from your process **for just about every CPU**

Spectre: Exploitation scenarios

Leaking browser memory

- A JavaScript (perhaps in an advertisement) can run Spectre
- Can leak browser cache, session key, other site data

Meltdown: Key idea

1. Attempt instruction with memory operand (Base+A), where A is a value forbidden to the process
2. The CPU schedules a privilege check and the actual access
3. The privilege check fails, but due to speculative executive, the access has already run and the result has been cached
4. Conduct a timing attack reading memory at the address (Base+A) for all possible values of A. The one that ran will return faster

Meltdown allows you to read **any memory in the address space (even from other processes)** but only on some Intel/ARM CPUs

Meltdown Attack (timing)

Now the attacker read each page of probe array

255 of them will be slow

The X^{th} page will be faster (it is cached!)

We get the value of X using cache-timing side channel

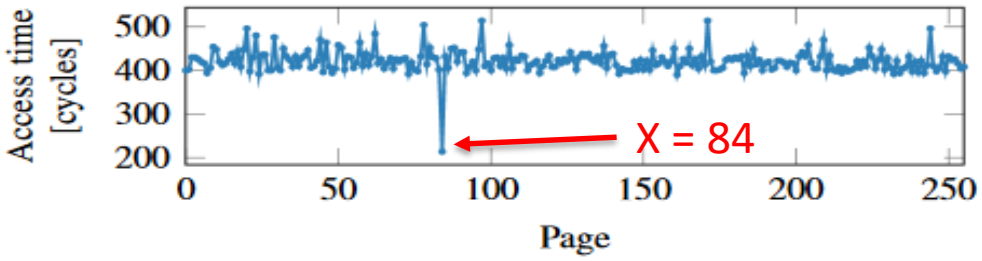


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

Meltdown: Mitigation

KAISER/KPTI (kernel page table isolation) patch

- Remove kernel memory mapping in user space processes
- Have non-negligible performance impact
- Some kernel memory still needs to be mapped