15-316: Software Foundations of Security and Privacy

Lecture Notes on Memory Side Channels

Matt Fredrikson

Carnegie Mellon University

Lecture 16

1 Introduction

In the previous lecture, we begin discussing *side channels* as a means of obtaining information about secret program state using on observations that fall outside the formal model of any information flow protections that are in place. In particular, we looked at timing channels that arise when an attacker can observe how long a program takes to execute, or how many instructions it executed, to infer secret state.

We informally concluded that timing channels might exist whenever the control flow of a program depends on secret state. We then formalized this by introducing *cost semantics*, which characterize some aspect of resource consumption directly as part of the big-step semantics of the language.

$$\langle \omega, c \rangle \Downarrow^{r} \omega'$$

The cost semantics associates a value r with the execution of a program, which we defined to correspond to the number of operations performed by the program. Then we introduced a notion of side-channel security related to non-interference, which requires that L-equivalent initial states lead to identical values of resource consumption when a program is executed.

$$\forall \omega_1, \omega_2 . \omega_1 \approx_{\mathrm{L}} \omega_2 \land \langle \omega_1, c \rangle \Downarrow^{r_1} \omega_1' \land \langle \omega_2, c \rangle \Downarrow^{r_2} \omega_2' \Longrightarrow r_1 = r_2 \tag{1}$$

When the number of operations executed is a suitable proxy for how long the program takes to execute, this policy is sufficient to rule out timing side channels. We then constructed a type system for enforcing this policy, and discussed the corresponding *constant-time programming discipline* that is commonly used in practice to mitigate timing channels.

2 Cache side channels

So far we've focused on timing leaks that arise due to differences in the number of steps taken along a particular control flow path. It might also be the case that even when the program executes exactly the same instructions, there are differences that crop up in execution time due to other factors. One such factor is cache behavior, i.e., the state of the processor's cache lines might cause the execution time to differ with variances in high-security state.

2.1 Example: AES block cipher

This type of side channel was famously exploited by Dan Bernstein [1] and others to attack software implementations of the AES encryption primitive. To understand how the attack works, we'll need some basic information about AES.

Basics of AES. AES is a block cipher, which encrypts a 16-byte input p using a 16-byte key k. Essential to AES's encryption are two so-called S-boxes, which are nothing more than 256 byte tables loaded with values that are constant across all implementations of AES. These tables are expanded into four 1024-byte tables T_0, T_1, T_2, T_3 by applying an expansion:

$$\begin{aligned}
T_0[i] &= (S'[i], S[i], S[i], S[i] \oplus S'[i]) \\
T_1[i] &= (S[i] \oplus S'[i], S'[i], S[i], S[i]) \\
T_2[i] &= (S[i], S[i] \oplus S'[i], S'[i], S[i]) \\
T_3[i] &= (S[i], S[i], S[i] \oplus S'[i], S'[i])
\end{aligned}$$

In most implementations, these tables are pre-computed and loaded into memory before any encryption takes places. For each 16-byte block p to be encrypted, AES first applies a transformation to k (which we won't cover in detail here), and then uses the tables T_0, T_1, T_2, T_3 to scramble p. Let $p = p_0, p_1, p_2, p_3$, so that p_i is a 4-byte fragment of p, and similarly for $k = k_0, k_1, k_2, k_3$. AES replaces each p_i as follows:

$$\begin{aligned}
p_0 &= T_0[p_0[0] \oplus k_0[0]] \oplus T_1[p_1[1] \oplus k_1[1]] \oplus T_2[p_2[2] \oplus k_2[2]] \oplus T_3[p_3[3] \oplus k_3[3]] \oplus k_0 \\
p_1 &= T_0[p_1[0] \oplus k_1[0]] \oplus T_1[p_2[1] \oplus k_2[1]] \oplus T_2[p_3[2] \oplus k_3[2]] \oplus T_3[p_0[3] \oplus k_0[3]] \oplus k_1 \\
p_2 &= T_0[p_2[0] \oplus k_2[0]] \oplus T_1[p_3[1] \oplus k_3[1]] \oplus T_2[p_0[2] \oplus k_0[2]] \oplus T_3[p_1[3] \oplus k_1[3]] \oplus k_2 \\
p_3 &= T_0[p_3[0] \oplus k_3[0]] \oplus T_1[p_0[1] \oplus k_0[1]] \oplus T_2[p_1[2] \oplus k_1[2]] \oplus T_3[p_2[3] \oplus k_2[3]] \oplus k_3
\end{aligned}$$

More concisely,

$$p_i = \left(\bigoplus_{0 \le j \le 3} T_j[p_{(i+j)\%4}[j] \oplus k_{(i+j)\%4}[j]] \right) \oplus k_i$$

It continues to modify k and p in this fashion for ten rounds, at which point the contents of p are the final ciphertext.

Leaking key bits. Notice that in each round, the value of p_i is computed using table lookup and xor. Each table lookup accesses an index that is dependent on the contents of the key, e.g., the operation $T_0[p_0[0] \oplus k_0[0]]$ will access a different element of the array holding T_0 for different values of the key k. If the amount of time necessary to look up this element varies depending on the index that is accessed, then we can reason that the total execution time of the encryption will depend on the value of k.

For this to work, we need to know what timing to expect as a function of k, or some approximation of it. This is where the cache comes into play. Because cache is a limited resource, several main memory blocks are mapped to the same cache block by way of a hash function H. Suppose that address a was previously read, causing the cache address $H(a)$ to hold its value afterwards, and subsequent accesses to a will complete more quickly. If we then read address a', such that $H(a) = H(a')$, then the corresponding cache block will no longer hold the value at a, and a subsequent read to a will need to fetch from main memory, thus taking longer to complete.

This is the crux of the attack: by selectively evicting the cache blocks corresponding to different elements of T_0, T_1, T_2, T_3, we can force encryption to take longer than it would have otherwise. Additionally, because the elements of T_i accessed by encryption depend on k, we can learn the contents of k by inspecting which evictions cause the encryption to require more time.

In short, the attack works as follows.

1. Ensure that T_0, T_1, T_2, T_3 are cached, e.g., by performing an encryption.

2. Select an element of T_0 to be evicted from the cache, and force its eviction by loading from an address that maps to the same cache block. This can be accomplished by guessing a value for $k_0[0]$, and determining which cache block the subsequent table lookup will consult.

3. Perform an encryption, and measure its time.

4. After doing this for each element of T_0, conclude that $k_0[0]$ takes the value that corresponds to the longest lookup from T_0.

5. Repeat for $k_0[1], k_0[2], \ldots, k_3[3]$.

Notice that this attack requires several capabilities of the attacker.

- The ability to time the execution of encryptions with precision sufficient to detect cache timing differences.

- The ability to selectively evict portions of the cache.

- The ability to force encryptions.

- Knowledge of the plaintext (i.e., this is a "known plaintext" attack), but not the key. Without this, it is not possible to determine in advance which T_i will be accessed, as it is indexed as $p_{(i+1)\%4}[j] \oplus k_{(i+1)\%4}[j]$.

Although these requirements may seem improbable, attacks like this have been demonstrated in practice, and preventing them requires careful constant-time programming discipline.

An alternate attack. There is an alternate way of mounting this attack known as FLUSH+RELOAD [4]. Consider an attacker who operates as follows.

1. Flush the entire cache.
2. Trigger an encryption.
3. Access memory corresponding to each cache block, and see which addresses take longer to load. Those that do must not have been accessed during the AES operation.

This attack doesn't measure the encryption routine's timing at all, but instead measures the timing of the attacker's code! In fact, this is actually a more efficient attack, as one encryption yields substantially more information about which table elements were accessed, and thus more information about the key. Yarom and Falkner [4] reported being able to recover approximately 98% of the bits in an encryption key by triggering a single encryption with this method.

This variant of the attack works because the cache is a shared resource that allows users to infer certain details of how other applications use it. Specifically, the cache allows users to determine which memory addresses were recently accessed by other processes. Thinking in general terms, we can thus abstract the attacker's abilities here as observing memory access patterns: the cache side channel attacker is able to observe which memory locations are accessed by a program, but not the contents of the access.

2.2 Plugging the leak: memory access as cost

Like in the case of `fastmatch` from the previous lecture, this vulnerability arose due to the attacker's ability to detect timing differences in an operation that depends on secret data. These timing differences were caused by the cache's state depending on this secret data, which gave the attacker the ability to degrade performance when her guess for the secret key was correct.

Armed with this insight, we can begin to reason about how to effectively mitigate cache side channels. Before, we reasoned that the number of execution steps (our proxy for timing) could be mitigated by ensuring that control flow does not depend on the contents of secret variables, as long as each step takes the same amount of time. Now, we can translate this approach to our attacker's new set of observations, and conclude that attacks like the one we just saw can be mitigated by ensuring that the set of memory addresses accessed by the program does not depend on secret state.

To mitigate timing channels in the model that treats execution steps as observations, we formalized security in terms of cost semantics using execution step costs, and then

$$\frac{}{\langle \omega, c\rangle \Downarrow^{\epsilon}_{\mathbb{Z}} c} \qquad \frac{\omega_V(x) = v}{\langle \omega, x\rangle \Downarrow^{\epsilon}_{\mathbb{Z}} v} \qquad \frac{\langle \omega, e\rangle \Downarrow^{r}_{\mathbb{Z}} v_1 \quad \omega_M(v_1) = v_2}{\langle \omega, \texttt{Mem}(e)\rangle \Downarrow^{r::v_1}_{\mathbb{Z}} v_2} \qquad \frac{\langle \omega, e\rangle \Downarrow^{r_1}_{\mathbb{Z}} v_1 \quad \langle \omega, \tilde{e}\rangle \Downarrow^{r_2}_{\mathbb{Z}} v_2}{\langle \omega, e \odot \tilde{e}\rangle \Downarrow^{r_1+r_2+1}_{\mathbb{Z}} v_1 \odot v_2}$$

$$\frac{}{\langle \omega, \texttt{true}\rangle \Downarrow^{\epsilon}_{\mathbb{B}} \top} \qquad \frac{}{\langle \omega, \texttt{false}\rangle \Downarrow^{\epsilon}_{\mathbb{B}} \bot} \qquad \frac{\langle \omega, P\rangle \Downarrow^{r}_{\mathbb{B}} b}{\langle \omega, \odot P\rangle \Downarrow^{r}_{\mathbb{B}} \odot b} \qquad \frac{\langle \omega, P\rangle \Downarrow^{r_1}_{\mathbb{B}} b_1 \quad \langle \omega, Q\rangle \Downarrow^{r_2}_{\mathbb{B}} b_2}{\langle \omega, P \ \odot \ Q\rangle \Downarrow^{r_1::r_2}_{\mathbb{B}} b_1 \odot b_2}$$

$$\frac{\langle \omega, e\rangle \Downarrow^{r}_{\mathbb{Z}} v}{\langle \omega, x := e\rangle \Downarrow^{r} \omega_V\{x \mapsto v\}} \qquad \frac{\langle \omega, \tilde{e}\rangle \Downarrow^{r_1}_{\mathbb{Z}} v_1 \quad \langle \omega, e\rangle \Downarrow^{r_2}_{\mathbb{Z}} v_2}{\langle \omega, \texttt{Mem}(e) := \tilde{e}\rangle \Downarrow^{r_1::r_2::v_2}_{\mathbb{Z}} \omega_M\{v_2 \mapsto v_1\}}$$

$$\frac{\langle \omega, \alpha\rangle \Downarrow^{r_1} \omega_1 \quad \langle \omega_1, \beta\rangle \Downarrow^{r_2} \omega'}{\langle \omega, \alpha; \beta\rangle \Downarrow^{r_1::r_2} \omega'} \qquad \frac{\langle \omega, P\rangle \Downarrow^{r_1}_{\mathbb{B}} \top \quad \langle \omega, \alpha\rangle \Downarrow^{r_2} \omega'}{\langle \omega, \texttt{if}(P)\, \alpha \, \texttt{else}\, \beta\rangle \Downarrow^{r_1::r_2} \omega'} \qquad \frac{\langle \omega, P\rangle \Downarrow^{r_1}_{\mathbb{B}} \bot \quad \langle \omega, \beta\rangle \Downarrow^{r_2} \omega'}{\langle \omega, \texttt{if}(P)\, \alpha \, \texttt{else}\, \beta\rangle \Downarrow^{r_1::r_2} \omega'}$$

$$\frac{\langle \omega, P\rangle \Downarrow^{r}_{\mathbb{B}} \bot}{\langle \omega, \texttt{while}(P)\, \alpha\rangle \Downarrow^{r} \omega} \qquad \frac{\langle \omega, P\rangle \Downarrow^{r_1}_{\mathbb{B}} \top \quad \langle \omega, \alpha; \texttt{while}(P)\, \alpha\rangle \Downarrow^{r_2} \omega'}{\langle \omega, \texttt{while}(P)\, \alpha\rangle \Downarrow^{r_1::r_2} \omega'}$$

Figure 1: Memory access cost semantics for the simple imperative language. The costs indicate the sequence of memory accesses made when executing the program in a given state.

designed a type system that prevent leakage from secret state to that cost. We can follow a similar strategy for cache side channels by defining a cost semantics that reflects memory access patterns in the cost.

Now our cost domain will consist of sequences of memory indices that are either read or written throughout the execution of a program. So for the following program that makes three memory accesses.

$$x := \texttt{Mem}(16); x := x + 1; \texttt{Mem}(32) := \texttt{Mem}(16) + x;$$

We would have the following "cost" as a sequential list of accesses: $16 :: 16 :: 32$. We will use ϵ to denote the empty list. Then the cost semantics for memory accesses are shown in Figure 1. For the most part these rules follow the same basic form as the execution step cost semantics shown in Figure **??**. Compound commands such as composition and conditional "add up" the costs r_1 and r_2 of their subcomponents using sequences concatenation $r_1 :: r_2$ rather than integer addition. Expressions and commands whose evaluation result in no memory accesses take cost ϵ to reflect the fact that they leave no observable information in the cost.

The only rules that change the cost in non-trivial ways are those for memory lookup $\texttt{Mem}(e)$ and update $\texttt{Mem}(e) := \tilde{e}$. In the former case, the index e is evaluated to v_1, and this evaluation step has its own cost r. Then the final cost for this expression is $r :: v_1$, as the location v_1 is accessed after r is incurred. In the latter case of memory update, first the right-hand side $\tilde{e}$ is evaluated at cost r_1, then the index e is evaluated to v_2 at cost r_2. Finally, the cost of executing this command is $r_1 :: r_2 :: v_2$, which reflects the order in which the subexpressions were evaluated with the final access being the one that updates memory in this command.

$$(\text{ConstL})\ \frac{}{\Gamma \vdash c : \mathtt{L}} \qquad (\text{TrueL})\ \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{L}} \qquad (\text{FalseL})\ \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{L}}$$

$$(\text{Var})\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad (\text{MemD})\ \frac{\Gamma \vdash e : \ell \quad \ell \sqcup \Gamma(\mathtt{pc}) \sqsubseteq \mathtt{L}}{\Gamma \vdash \mathtt{Mem}(e) : \mathtt{L}}$$

$$(\text{UnOp})\ \frac{\Gamma \vdash e : \ell}{\Gamma \vdash \odot\, e : \ell} \qquad (\text{BinOp})\ \frac{\Gamma \vdash e : \ell_1 \quad \Gamma \vdash \tilde{e} : \ell_2}{\Gamma \vdash e \odot \tilde{e} : \ell_1 \sqcup \ell_2} \qquad (\text{Comp})\ \frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha ; \beta}$$

$$(\text{Asgn})\ \frac{\Gamma \vdash e : \ell_1 \quad \ell_1 \sqcup \Gamma(\mathtt{pc}) \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e}$$

$$(\text{MemU})\ \frac{\Gamma \vdash e : \ell_1 \quad \Gamma \vdash \tilde{e} : \ell_2 \quad \ell_1 \sqcup \ell_2 \sqcup \Gamma(\mathtt{pc}) \sqsubseteq \mathtt{L}}{\Gamma \vdash \mathtt{Mem}(e) := \tilde{e}}$$

$$(\text{If})\ \frac{\Gamma \vdash Q : \ell \quad \ell' = \ell \sqcup \Gamma(\mathtt{pc}) \quad \Gamma, \mathtt{pc} : \ell' \vdash \alpha \quad \Gamma, \mathtt{pc} : \ell' \vdash \beta}{\Gamma \vdash \mathtt{if}(Q)\, \alpha\, \mathtt{else}\, \beta}$$

$$(\text{While})\ \frac{\Gamma \vdash Q : \ell \quad \ell' = \ell \sqcup \Gamma(\mathtt{pc}) \quad \Gamma, \mathtt{pc} : \ell' \vdash \alpha}{\Gamma \vdash \mathtt{while}(Q)\, \alpha}$$

Figure 2: Conservative type system for mitigating cache side-channel leaks.

Now that we have characterized the attacker's observation in a cost semantics, we must define what it means for a program to be secure in this model. Luckily, the definitions from before as shown in Eqs. 1 and **??** do not make any assumptions about the cost domain other than that it is equipped with some notion of equality. Certainly finite sequential lists of memory accesses have equality, so we can just re-use those notions of cost-aware non-interference here again.

2.3 A cache-channel type system

Figure 2 shows a type system that follows the same rationale as the one that we saw for timing side-channels. Namely, a program is well-typed in this system only if there is no flow of information from secret state to memory accesses. All of the rules except MemD and MemU are exactly as they were in the original non-interference type system. The rule MemD for memory lookup expressions first types the index expression e as ℓ, and then assigns the lookup expression type L as long as $\ell \sqcup \Gamma(\mathtt{pc}) \sqsubseteq \mathtt{L}$. This prevents lookups in both cases where e contains secret information, as well as when lookups happen in secret-dependent control flow.

Also important is the fact that MemD enforces the invariant that everything read from memory is of type L. The second half of this invariant comes from rule MemU for memory updates, which checks that the type ℓ_2 of the right-hand side as well as the pc are both typed L before allowing the update. If the type system did not enforce this invariant, then we would need to assign security labels to each cell of memory as part of the context Γ. It may be possible if a bit unwieldly to do so, but for the purposes of this lecture not necessary.

Finally, MemU also checks that the type of the index expression is `L`, which is again necessary to prevent secret information from leaking into the access cost. In the end this type system enforces the same security guarantee as the timing-channel type system, as stated in Theorem 1.

Theorem 1. *The type system in Figure 2 enforces both non-interference and cache side-channel security. That is, if $\Gamma \vdash \alpha$ by the rules in Figure 2 then for all $\omega_1 \approx_{\mathtt{L}} \omega_2$,*

$$\langle \omega_1, c\rangle \Downarrow^{r_1} \omega_1' \wedge \langle \omega_2, c\rangle \Downarrow^{r_2} \omega_2' \Longrightarrow r_1 = r_2 \wedge \omega_1' \approx_{\mathtt{L}} \omega_2'$$

So α terminates with the same memory access patterns and in `L`-equivalent final states when initialized in either ω_1 or ω_2

Proof. The proof of this theorem follows a very similar form to that of Theorem 2 from the previous lecture and the soundness theorem of the non-interference type system from Lecture 12. The only extra consideration that must be done is regarding MemD and MemU. It may be helpful to factor out a lemma which proves the invariant that the contents of `Mem` are never influenced by secret data. This is left as an exercise. □

3 Meltdown

The cache side-channel attacks that we have so far discussed all rely on vulnerable characteristics of a program that works with secret data. While it may be challenging in some cases to avoid secret-dependent memory access patterns, it is comforting to know that we can mitigate these attacks by following an appropriate typing discipline.

Recall from the first lecture when we discussed the Spectre and Meltdown vulnerabilities disclosed in January 2018. These vulnerabilities rely on cache side channels, but do not necessarily require secret-dependent memory access patterns in a vulnerable process. Let's take a closer look at one of these vulnerabilities to better understand how this is possible, and how such an attack could be mitigated.

3.1 Virtual address spaces

The first key to understanding how the Meltdown attack works is the way that most modern platforms set up process address spaces. To support the simultaneous execution of multiple distinct processes on a single system, processors provide *virtual address spaces* that map virtual addresses (i.e., addresses referenced by process code) to physical memory addresses. The virtual address space is divided into units called *pages*, and the processor's mapping consists of a multi-level page translation table from virtual page addresses to physical ones. The translation tables also contain information that defines which processes are able to read, write, and execute the contents of a particular virtual page.

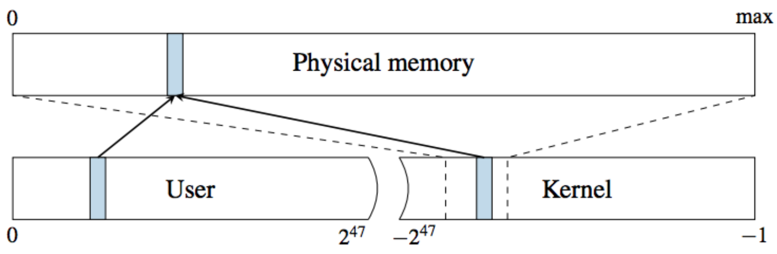

Figure 3: Physical addresses (shown in blue) for process memory are mapped both in the process' virtual address space, as well as in the kernel space. A subset of the kernel's virtual address space contains a mapping of the entire physical address space (Figure due to Lipp et al. [3]).

The processor has a register that holds a pointer to the current process' page translation table, which is updated each time the operating system performs a context switch to let the next process run. So each process has its own virtual address space defined by the translation mapping, and can only reference memory locations that are defined in its virtual address space. Each process' address space is divided into *user memory* and *kernel memory*, where the user portion contains the process-specific memory contents and the kernel portion contains the memory used by the operating system.

The operating system often needs to refer to user memory, such as when it reads arguments provided to system calls and services other requests and exceptions. To facilitate this, a portion of kernel memory is often devoted to directly mapping all of user memory (i.e., for all processes) into a range of kernel memory. This is possible because the virtual address space on modern architectures allows for referencing 2^{64} distinct addresses, which corresponds to tens of thousands of petabytes, whereas the amount of physical memory available on machines much smaller.

If the virtual address space for a process also contains mappings for the kernel's space, then what prevents a rogue process like the following from simply reading kernel memory and by extension the memory of any other process running on the system?

```
char *physmem =  0xffff880000000000;  // kernel's physical memory map
for(i = 0; i < MEMSIZE; i+=4096)
  send(sock, &physmem[i], 4096);      // send each page of memory
```

This would obviously be problematic, so the page translation tables specify that kernel pages can only be accessed by code running in *priveleged mode*, and typical platforms only grant this to the operating system code itself. So under normal circumstances untrusted processes are not allowed to interact with kernel memory, including the portion that maps to other process' memory.

3.2 Out-of-order execution

Since the mid-1990's, processors have supported optimizations that are based on executing instructions out of the order in which they appear in memory. This is a powerful technique that allows the processor to better utilize available resources in cases where the specified ordering imposes latency and underutilization of the CPU's execution units. For example, in the `fastmatch` implementation from last lecture, each index of `pin` was compared against the corresponding index of `guess`.

```
while(i < len) {
  auth := auth & (pin(i) = guess(i));
  i := i + 1;
}
```

Without out-of-order execution, the processor would need to first fetch `pin(i)`, wait for the result to come back from main memory (or the cache), then fetch `guess(i)`, wait for the result, and finally compute the equality comparison. Most processors have multiple execution lines that can operate simultaneously, so out-of-order execution allows the processor to begin fetching `guess(i)` before the result of `pin(i)` returns from main memory.

The processor can even decide to begin executing an instruction in cases where the instruction may not end up executing due to a conditional control flow transfer. For example, in the timing-vulnerable version of `fastmatch`:

```
while(i < len) {
  if(pin(i) != guess(i)) {
    auth := 0;
    i := len;
  }
  i := i + 1;
}
```

If it turns out that `pin(0) != guess(0)`, the processor may have jumped ahead and begin fetching `pin(1)` and `guess(1)` just in case the condition had evaluated to `false` and it needed those values to compute `pin(1) != guess(1)`. This is called *speculative execution*, and is widely used by modern platforms.

Importantly, speculative execution is very granular, and applies not just to distinct instructions but also the micro-operations needed to execute a single instruction. So if the following code were run by an unpriveleged process:

```
char *physmem =  0xffff880000000000;  // kernel's physical memory map
x = physmem[1];
```

Then the processor may decide to fetch the contents of `physmem[1]` concurrently while checking the access rights of the process against the protection flags for address `&physmem[1]` specified in the page translation tables. When the check fails, the processor erases any pending updates to its state resulting from the speculative fetch of `physmem[1]`, and raises an exception. However, it turns out that this erasure is incomplete, and certain parts of the arcthitectural state including the memory cache may still

3.3 Rogue memory access

The Meltdown attack exploits speculative memory reads, as well as the fact that the kernel's address space maps all of physical memory, to present a method for reading arbitrary physical memory contents from unprivileged code. The method assumes an attacker that can execute arbitrary code as a normal unprivileged user, but does not necessarily have physical access to the machine running the code. Importantly, the operating system on the machine and the other processes may all be bug-free, and the processor may implement standard memory protection mechanisms—the attack will still work.

The basic attack consists of the following three steps.

1. The attacker loads the contents of a targeted memory address into a register.
2. A speculative instruction accesses a memory cache location that is based on the contents of the register.
3. The attacker uses a cache side-channel attack such as FLUSH+RELOAD to determine which cache location was accessed by the speculative instruction.

Because the cache location depends on the contents of the targeted address, the attacker can deduce the value held in that address by observing which cache location was accessed. Note that if the attacker targets an address in kernel space, then the instructions in Steps 1 and 2 are both speculative, and will not be committed to the architectural state (but the effect in the cache will remain).

Rather, the access check in Step 1 will fail, and the processor will raise an exception for the attacker's process. To avoid crashing the process, the attacker must register an exception handler to continue executing the cache side channel. Alternatively, the attacker can allow the process to terminate, and mount the FLUSH+RELOAD attack from a separate process. For further details and example code, consult the original paper describing the attack [3].

3.4 Fixing Meltdown

This all seems rather grim. We have an attack that targets operating systems that are correctly-implemented, requires no special privileges, and allows untrusted processes to read arbitrary regions of memory that might contain passwords, credit card numbers, and our other deepest, darkest secrets. What can we possibly do to fix it?

KPTI. The approach that most platform vendors have taken so far to address Meltdown is called *Kernel Page Table Isolation* (KPTI). If we go back and think about all of the conditions that gave rise to the attack, one very good question might come to mind: why is the kernel address space mapped in unprivileged processes in the first place?

It turns out that there are solid performance reasons for doing so. To reduce the overhead of mapping virtual memory addresses to their corresponding physical addresses, processors use a *Translation Lookaside Buffer* (TLB) that caches recently-accessed virtual-physical mappings. By keeping the kernel portion of the mapping persistent across all virtual memory spaces, the corresponding TLB entries do not need to be flushed when switching between user and kernel code, or even when performing context switches between processes. Flushing the TLB is expensive because accessing page tables to repopulate it is costly, and even flushing the entries without repopulating is quite slow.

So keeping the kernel's virtual address space mapped in each process improves performance significantly, and until Meltdown was discovered was thought to be secure thanks to standard virtual memory protection checks. But now that we have seen that leaving kernel memory addressable to unpriveleged code is not secure, perhaps it makes sense to rethink this design decision and sacrifice some performance.

This is exactly what KPTI does: removes kernel address space mappings from the page tables that are loaded when unprivileged processes run. More precisely, it isn't possible to remove *all* kernel mappings because some code is needed to handle system calls, exceptions, and interrupts that might occur during unprivileged execution. These portions remain, but the rest of kernel memory, including the portion that maps the entire physical memory, are no longer present. To date, some form of KPTI is the approach that developers have taken to fix Meltdown.

Secure Hardware Description Languages. Another way to think about the vulnerability that enables Meltdown is in terms of information flow. We can view the addresses in kernel space as having a high-security label `H` and those in user space as having label `L`, and enforcing a non-interference policy over these labels. But so far we have only thought about enforcing non-interference on code, and the vulnerability in this case comes from the way the hardware operates: when speculative memory reads occur, they flow to the state of the cache, which can then flow to the `L` user-space addresses. So do we need to develop an entirely new framework for reasoning about and enforcing non-interference in hardware if we want to design processors that don't have such vulnerabilities in the future?

It turns out that hardware designers use languages, called *Hardware Description Languages* (HDLs), to design such functionality. While HDLs are specialized to the needs of efficiently designing and prototyping hardware, they are programming languages just like any other, and the same principles and techniques that we have discussed all semester can be applied to them as well. In particular, Ferraiuolo et al. [2] recently proposed adding security types to HDLs with the express goal of making it possible to design side channel-free hardware with rigorous guarantees about information flow security.

Designing new hardware using provably-secure languages is not an approach that leads to immediate fixes, and does nothing to patch the vulnerability on existing platforms that were designed using conventional tools and techniques. But if we are to learn from the unfortunate circumstances surrounding Meltdown and its sibling attack Spectre, one lesson is that hardware side channels are subtle and can lurk in obscurity even when conventional means (e.g., virtual memory protections) seem to have worked well for years. Fixing them reactively, as with KPTI, can be costly and unappealing, and leaves an unacceptable window of vulnerability. When designing the next generation of hardware, principled methods like secure HDLs are an attractive alternative to the status quo that has repeatedly failed when it comes to security until now.

References

[1] D. J. Bernstein. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2004.

[2] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

[4] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.