15-316: Software Foundations of Security and Privacy

Lecture Notes on Side Channels

Matt Fredrikson

Carnegie Mellon University
Lecture 15

1 Introduction

A *side channel* is a means of obtaining information about secret program state that relies on observations that fall *outside* the formal model of any information flow protections that are in place. In recent years, so-called **side channel attacks** that leave otherwise well-designed and implemented systems vulnerable to serious issues, such as leaked encryption keys and sensitive user data.

What does it mean for an observation to fall outside the formal model? Think back to the way that we defined indistinguishability sets. In particular, we defined them with respect to a pair of observations ω_L, ω'_L. This pair constitutes the **observation model** of our information flow protections, and the guarantee that we obtain is contingent on the attacker's observations falling within the scope of this pair.

Oftentimes, when programs are run on real systems, there are aspects of the ensuing execution that are not considered in the formal model used to design the protection mechanism. Some examples of these are:

1. Execution time
2. Size of the program's memory footprint, memory access patterns
3. Sequence of instructions executed by the program
4. Electromagnetic radiation emitted from processor, other hardware components
5. Power usage of hardware components

If the attacker is able to make observations that are influenced by any of these aspects, then any information flow guarantees that rely on the incompatible observation model will not apply.

2 Revisiting `match`

Let's return to the **match** function that we've discussed several times before. Suppose that we chose to implement a version called `fastmatch` that compares two lists for equality in the following way.

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

Is anything wrong with this implementation? Not according to the semantics we've studied previously. We can declassify the output of this function, and the only way an attacker can misuse the result to leak a secret is by making an exponential number of calls to this code.

However, if we change the attacker's *observation model* to include the amount of time it takes the code to complete, then the story changes. For now we'll just think of timing as the number of execution steps (informally defined at the moment) that it takes to execute the program. Obviously, obtaining such exact information in practice may be difficult, but this simplification will help us see the big picture first.

Let's break this down further to see what information can be learned from this new observation.

- When the values differ on their first element, the function will return immediately. This is the least amount of time `fastmatch` can take.
- When the inputs are the same, `fastmatch` will execute the longest.

Combining these two facts, the attacker knows that the longer `fastmatch` executes, the more elements they have successfully guessed at the beginning of the list. Can we use this intuition to significantly decrease the exponential-time bound we studied last week? Consider the following attack, where N is the number of possible characters that each position in a password or PIN number can take. We'll assume that this is finite, such as 10 or 256 (i.e., lists of digits or ASCII characters) or 2^{64} (i.e., machine integers).

1. First try all one-character passwords "x_1", where x is one of the possible values taken at indices of the password. Note the amount of time taken for `fastmatch` to return in a fresh variable t_{x_1}.
2. Take the first character of the password $p(0)$ to be $\arg\max_x t_{x_1}$, i.e. the character that took the longest for `fastmatch` to terminate on.
3. Now try all two-character passwords "$p(0)x_2$" obtained by appending each possible character to the value decided for the first character.

4. As before, when done enumerating all two-character passwords that begin with the decided prefix $p(0)$, update the prefix $p(0)p(1)$ to $\arg\max_x t_{x_1} + t_{x_2}$.
5. Continue appending characters that result in the longest execution of `fastmatch` until $auth = 1$ when it finishes.

What is the complexity of this attack? Let L be the length of the high-security PIN/-password, and we'll assume that elements are coded in binary so there are $\log(N)$ bits in each element. The brute-force approach that would have been necessary without the timing information required $2^{L\log(N)} = N^L$ queries. With timing information, each element takes exactly N guesses to find, and so now the attack will finish in LN queries to `fastmatch`. In short, timing information reduced an exponential attack into a linear one. Obviously this poses a serious problem.

3 Side-channel information leaks

Thinking back to when we discussed declassification, we introduced the notion of an observation model that in turn defines an indistinguishability set for the attacker. The observation model that we used then was simply the low-security portions of the initial and final states, $(\omega_{\mathrm{L}}, \omega'_{\mathrm{L}})$, we formalized information flow security as noninterference:

$$\forall \omega_1, \omega_2 . \omega_1 \approx_{\mathrm{L}} \omega_2 \wedge \langle \omega_1, c\rangle \Downarrow \omega'_1 \wedge \langle \omega_2, c\rangle \Downarrow \omega'_2 \implies \omega'_1 \approx_{\mathrm{L}} \omega'_2 \tag{1}$$

This worked out nicely because the observations $(\omega_{\mathrm{L}}, \omega'_{\mathrm{L}})$ are accounted for directly by the semantic relation $\Downarrow$. But now that we are concerned with information leakage through timing information, the attacker's observations must also contain the number of execution steps taken until the program terminates. How do we incorporate such information in a formal definition of security?

3.1 Cost semantics

One natural approach is to enrich the semantics with precisely this information. Such a relation is called the *cost semantics*, as the idea was originally conceived in the context of formalizing the performance of programs in terms of execution time [3]. To see how this works, recall our original semantic relations for expressions and commands.

$$\langle \omega, e\rangle \Downarrow v \qquad\qquad \langle \omega, c\rangle \Downarrow \omega'$$

This notation means that executing expression e (resp. command c) in environment ω yields value v (resp. state ω'). Now we want to incorporate a notion of execution time corresponding to discrete steps into our semantics, and we will do so by annotating the relation $\Downarrow$ with a cost r.

$$\langle \omega, e\rangle \Downarrow_r v \qquad\qquad \langle \omega, c\rangle \Downarrow_r \omega'$$

This notation means that executing expression e (resp. command c) in environment ω yields value v (resp. state ω') in exactly r steps. In this case, r is a non-negative integer,

$$\frac{}{\langle \omega, c\rangle \Downarrow^1_{\mathbb{Z}} c} \quad \frac{\omega(x) = v}{\langle \omega, x\rangle \Downarrow^1_{\mathbb{Z}} v} \quad \frac{\langle \omega, e\rangle \Downarrow^{r_1}_{\mathbb{Z}} v_1 \quad \langle \omega, \tilde{e}\rangle \Downarrow^{r_2}_{\mathbb{Z}} v_2}{\langle \omega, e \odot \tilde{e}\rangle \Downarrow^{r_1+r_2+1}_{\mathbb{Z}} v_1 \odot v_2} \quad \frac{}{\langle \omega, \texttt{true}\rangle \Downarrow^1_{\mathbb{B}} \top}$$

$$\frac{}{\langle \omega, \texttt{false}\rangle \Downarrow^1_{\mathbb{B}} \bot} \quad \frac{\langle \omega, P\rangle \Downarrow^r_{\mathbb{B}} b}{\langle \omega, \odot P\rangle \Downarrow^{r+1}_{\mathbb{B}} \odot b} \quad \frac{\langle \omega, P\rangle \Downarrow^{r_1}_{\mathbb{B}} b_1 \quad \langle \omega, Q\rangle \Downarrow^{r_2}_{\mathbb{B}} b_2}{\langle \omega, P \ \odot \ Q\rangle \Downarrow^{r_1+r_2+1}_{\mathbb{B}} b_1 \odot b_2}$$

$$\frac{\langle \omega, e\rangle \Downarrow^r_{\mathbb{Z}} v}{\langle \omega, x := e\rangle \Downarrow^{r+1} \omega\{x \mapsto v\}} \quad \frac{\langle \omega, \alpha\rangle \Downarrow^{r_1} \omega_1 \quad \langle \omega_1, \beta\rangle \Downarrow^{r_2} \omega'}{\langle \omega, \alpha; \beta\rangle \Downarrow^{r_1+r_2} \omega'}$$

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

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

Figure 1: Step-execution cost semantics for the simple imperative language. The costs indicate the number of steps needed to execute the program in a given state.

but we can take r to be a value from a different domain to account for different types of cost. For example, we will see later how to define a cost semantics that accounts for memory access patterns using a different domain for r. An example cost semantics is shown in Figure 1, corresponding to the observation of the number of execution steps taken to execute an expression or command.

Question. *The cost semantics shown in Figure 1 is rather simplistic in terms of the costs that it assigns to certain operations. For example, the same cost is assigned to evaluating an integer constant as looking a variable up in memory. This model won't have a precise correspondence with real execution time, even ignoring things like the cache. How might you refine the semantics to more faithfully account for timing? Can you incorporate empirical measurements, and if so, what is the best way to go about it?*

Question. *We've talked about two distinct observation models, but these semantics only account for one. Supposing we have two cost semantics that account for each observation model, how can we combine them into a single cost semantics that lets us reason about both types of observation?*

3.2 Side-channel security

Now that the information about runtime available to the attacker is evident in our semantics, we can now go about formalizing what it means for a program to be secure with respect to leakage through this channel. We want to express a condition which says that regardless of the values contained in the secret portions of state, the attacker's observations over the side channel remain constant. We can follow the basic form of noninterference (Equation 1, and write:

$$\forall \omega_1, \omega_2 . \omega_1 \approx_{\mathrm{L}} \omega_2 \wedge \langle \omega_1, c\rangle \Downarrow_{r_1} \omega_1' \wedge \langle \omega_2, c\rangle \Downarrow_{r_2} \omega_2' \implies r_1 = r_2 \tag{2}$$

This aligns perfectly with our intuition that observing the final execution cost is no different from observing the low-security portions of the final state. In either case, we formalize security by demanding equivalence of the final observations whenever we have equivalence of the initial observations. Note that Equation 2 doesn't account for observation of the low-security final state, but we can easily add this as follows.

$$\forall \omega_1, \omega_2 . \omega_1 \approx_{\mathtt{L}} \omega_2 \wedge \langle \omega_1, c \rangle \Downarrow_{r_1} \omega_1' \wedge \langle \omega_2, c \rangle \Downarrow_{r_2} \omega_2' \Longrightarrow r_1 = r_2 \wedge \omega_1' \approx_{\mathtt{L}} \omega_2' \tag{3}$$

Given definition of side-channel security, how might we go about designing a type system which ensures that they hold? What do we need to do differently from the case of basic noninterference when we prove soundness of such a type system? These are good questions to think about when preparing for an exam.

4 Constant-time programming discipline

Let us go back to the `fastmatch` example and think about Equation 2 in hope of developing a general approach to avoiding such timing leaks. Intuitively, the fact that the runtime of the program is influenced by high-security data is the direct cause of the problem. What are the ways in which high-security data can influence runtime? Looking at the evaluation rules for expressions, we can reason that the runtime is not dependent on the values that variables take, but rather only the number of operations present in an expression.

Lemma 1 (Constant-time expressions)**.** *Given any expression e, there exists a constant c such that for all ω and some v, $\langle \omega, e \rangle \Downarrow^c v$.*

Proof. This is a straightforward structural induction on e. You are encouraged to work out several of the cases as an exercise. □

Question. *Is this true on real computing platforms? What are examples of expressions that, when compiled, might lead to exeuction times that are dependent on the value of the operands?*

So this leads us somewhat unsurprisingly to commands as the culprit for secret-dependent timing channels. But do we need to worry about all commands? Perhaps not, which we see in the case of assignments. The runtime of those is exactly the runtime of evaluating the right-hand side expression plus one (to store the result), so the constant-time exeuction of assignments follows easily from Lemma 1.

But the remaining compound expressions are problematic. Consider an assignment $\mathtt{if}(Q)\,\alpha\,\mathtt{else}\,\beta$, and assume that α takes r_α steps while β takes r_β. If $r_\alpha \neq r_\beta$, then depending on the value of Q the entire statement will take a varying number of steps to complete. Critically, if $\Gamma \vdash Q : \mathtt{H}$ then the number of steps will absolutely depend on secret data. It is not hard to see that the exact same situation holds for `while` loops guarded by condition Q typed `H`.

So we come to realize that timing channels can arise whenever the program's control flow depends on secret data. To be more precise, whenever a change in the value of a

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

$$\text{(Var)}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(UnOp)}\ \frac{\Gamma \vdash e : \ell}{\Gamma \vdash \odot\, e : \ell} \qquad \text{(BinOp)}\ \frac{\Gamma \vdash e : \ell_1 \quad \Gamma \vdash \tilde{e} : \ell_2}{\Gamma \vdash e \odot \tilde{e} : \ell_1 \sqcup \ell_2}$$

$$\text{(Asgn)}\ \frac{\Gamma \vdash e : \ell \quad \ell \sqsubseteq \Gamma(x)}{\Gamma \vdash x := e} \qquad \text{(Comp)}\ \frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha; \beta}$$

$$\text{(If)}\ \frac{\Gamma \vdash Q : \mathtt{L} \quad \Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \mathtt{if}(Q)\, \alpha\, \mathtt{else}\, \beta} \qquad \text{(While)}\ \frac{\Gamma \vdash Q : \mathtt{L} \quad \Gamma \vdash \alpha}{\Gamma \vdash \mathtt{while}(Q)\, \alpha}$$

Figure 2: Type system for constant-time programming discipline.

high-security variable can give rise to a change in the program's control flow, timing channels may exist.

4.1 A constant-time type system

We can immeidately profit from this insight to design a type system that enforces side-channel security. Figure 2 shows the rules for this type system, which prevent information from any label $\ell \not\sqsubseteq \mathtt{L}$ from flowing to the runtime of a program α. As an added bonus, these rules also prevent flows that are observable in final-state assignments, i.e. those that are prevented by the information flow type system we have previously discussed.

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

$$\langle \omega_1, c \rangle \Downarrow_{r_1} \omega_1' \wedge \langle \omega_2, c \rangle \Downarrow_{r_2} \omega_2' \implies r_1 = r_2 \wedge \omega_1' \approx_{\mathtt{L}} \omega_2'$$

So α terminates in the same number of steps and in L*-equivalent final states when initialized in either ω_1 or ω_2*

Proof. This requires induction on the big-step derivation $\langle \omega, \alpha \rangle \Downarrow^r \omega'$ and goes much like the proof for the non-interference type system that we saw before. As this is an easier proof than the previous type system, it is left as an exercise. □

It may come as a bit of a surprise that the type system in Figure 2 is actually simpler than the one that we discussed for proving non-interference. We seem to obtain a stronger an more interesting result in Theorem 2 than our former soundness theorem, but the rules Asgn, If, and While have fewer preconditions than in the previous type system. How can this be?

The rub lies in the fact that these rules impose a more strict information flow discipline on well-typed programs. Before when we typed a conditional or while command,

we allowed the system to raise the label of `pc` to the type of the condition as long as the subcommands could be typed in the resulting context. In the constant-time system, the corresponding rules refuse to type any conditional or while command with an `H`-typed condition. This in turn means that the rule for assignment can be simplified by ignoring $\Gamma(pc)$, which is no longer relevant.

So while the type system may be simpler, this undoubtedly comes at the price of deeming fewer programs as well-typed. Perhaps we could have remedied this by making the judgements more nuanced. For example, designing the type system to require that the number of steps executed by both branches be identical even if the condition is typed `H`. This is an intriguing approach, and a topic of recent (and still active) research [4].

4.2 Writing `fastmatch` in constant-time

Programs that can be well-typed in rules like those in Figure 2 are said to be written in *constant-time programming discipline*. While it may seem quite restrictive to never branch on secret values, it is often the case that functionality which is most naturally written to branch on secrets can be expressed in constant-time discipline with some extra thought [1].

Let's think about how to fix the timing channel in `fastmatch`. We can think about this task in terms of the program counter: whenever its value depends on a secret, we're in likely trouble. There are two sources of secret-dependent control flow in the program.

1. The most obvious source is the conditional expression in the last `match`, which compares `x` and `y`. This is what causes the program to terminate early whenever the two inputs don't match.

2. A more subtle source of secret-depdenent control flow stems from the fact that the execution time of `fastmatch` is not bounded by a non-secret value. This isn't a problem in `fastmatch`, because it will always terminate early unless a correct guess is supplied as the low-security input. But if this were not the case, then the number of iterations would be a function of |`h`|, which would leak the length of the secret.

Looking at the code of `fastmatch`, notice first that the number of loop iterations is now bounded by `len`, which we'll assume to be non-secret. Where does the secret-dependent control flow come into play? The conditional statement inside the loop has a guard that mentions `h`, so we see that different values of `h` could lead to different control paths. In order to fix this, we'll obviously need to remove the conditional statement, so that the same sequence of instructions is executed regardless of the value of `h`. The only subtlety is that the output of `fastmatch` must depend on `h`, so we need to find a reasonable way to ensure that the outcome is the same as before.

One way of accomplishing this is to carry the computation of the loop forward through to the greatest number of iterations the loop can take. Looking at `fastmatch`, we can think of it as nothing more than an aggregate of Boolean expressions: when all of the

$pin(i) = guess(i)$, for $0 \leq$ `i` $< len$, then `fastmatch` returns 1. If $pin(i) \neq guess(i)$ for any `i`, then `fastmatch` returns 0. In other words, `fastmatch` is nothing more than a conjunction over equality literals, which we can easily implement using straight-line code in the loop.

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

It is important to point out that any solution that leaves the conditional intact is not in constant-time discipline. For example, one may initially opt for the seemingly more natural implementation shown below.

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

In this version of the program, we still have control flow that is dependent on secret state. However, the way we've written it, the same number of statements are executed on every branch, regardless of the value taken by secret state. Clearly, an attacker who is only allowed to see the execution time as the number of steps taken will have no difference in observations, so one might argue that in this case we need not worry about the secret-dependent control flow. However, this type of code is discouraged in constant-time programming discipline for various reasons.

- Code like the last example above tends to be more complex than necessary, and can be difficult to read. In order to achieve step-time equivalence on all paths, we needed to essentially insert a noop `auth := auth` in the `else` branch, which adds to the code's complexity, and might be innocently removed by a collaborator unaware of our constant-time goal.
- Leaving conditionals that are dependent on secrets in the code forces us to reason about whether all affected paths are step-time equivalent. As the complexity of code increases, this quickly becomes difficult and error-prone.
- Optimizing compilers might remove some branches, or instructions in branches, that we needed for step-time equivalence, with no guarantee that the resulting program is still constant-time. This would almost certainly happen if we compiled the above with `gcc` configured with standard optimizations.

In short, although it may seem unnatural and difficult to write programs so that control flow never depends on secret values, if constant-time execution is needed for security then adhering to this discipline is probably the simplest and least error-prone approach compatible with conventional imperative languages.

5 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.

5.1 Example: AES block cipher

This type of side channel was famously exploited by Dan Bernstein [2] 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,

$$p_i = T_0[p_i[0] \oplus k_i[0]] \oplus T_1[p_{(i+1)\%4}[1] \oplus k_{(i+1)\%4}[1]] \oplus T_2[p_{(i+2)\%4}[2] \oplus k_{(i+2)\%4}[2]] \\ \oplus T_3[p_{(i+3)\%4}[3] \oplus k_{(i+3)\%4}[3]] \oplus k_i$$

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 straightforward 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.

5.2 Plugging the leak

Like in the case of `fastmatch`, 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.

However, the problem is more subtle than before, as even correcting for timing would not necessarily thwart attack in this case. Consider an attacker who operates as follows.

1. Vacate the entire cache.
2. Trigger a single 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.

This variant of the attack works because the cache is a shared resource that allows users to infer details of how others 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.

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.

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

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

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

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

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

Figure 3: 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.

5.3 Memory access as cost

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 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 3. For the most part these rules follow the same basic form as the execution step cost semantics shown in Figure 1. 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,

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

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

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

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

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

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

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

Figure 4: Conservative type system for mitigating cache side-channel leaks.

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.

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. 2 and 3 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.

5.4 A cache-channel type system

Figure 4 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 $\mathtt{L}$ 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 3.

Theorem 3. *The type system in Figure 4 enforces both non-interference and cache side-channel security. That is, if $\Gamma \vdash \alpha$ by the rules in Figure 4 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 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. □

References

[1] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, 2016. USENIX Association.

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

[3] J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)*, pages 357–370, 2011.

[4] V. C. Ngo, M. Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. Verifying and synthesizing constant-resource implementations with types. In *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017.