15-316: Software Foundations of Security and Privacy

Lecture Notes on Timing Side Channels

Matt Fredrikson

Carnegie Mellon University

Lecture 14

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 arrays, and stops as soon as it sees two entries that don't match. This implementation also returns immediately if the lengths of the two input arrays don't match—very fast!

```
bool fastmatch(byte a[], size_t a_len, byte b[], size_t b_len) {

  if(a_len != b_len) return false;

  for (size_t i = 0; i < a_len; i++)
    if (a[i] != b[i]) return false;

  return true;
}
```

Recall our discussion from a few lectures ago, where we reasoned that it is generally safe to leak the output of such a function, even in cases where one of the inputs is secret, because an attacker needs to call it in exponential number of times (in the size of its secret input) to learn the secret. However, if we change the attacker's *observation model* to include the amount of time it takes this 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 type of observation. Throughout our discussion, assume that `a` is the secret, and `b` is the attacker's guess.

- When `a_len` and `b_len` differ, then the function returns immediately (say, in 1 "step", as it needed to compare the two arguments).
- When `a_len = b_len` but `a[0] != b[0]`, then the function still returns quickly, but note that it still needed to set `i = 0`, check `i < a_len`, and compare `a[0]` with `b[0]`. Following our simplistic notion of timing, this returns in four steps.
- When the inputs are identical, `fastmatch` will execute the longest. Each iteration of the loop comprises four steps (counting the increment `i++` as one step), so including the initial comparison, this amounts to `1 + 4*len` steps.

Thus, the attacker knows that the longer `fastmatch` executes, the more elements they have successfully guessed at the beginning of the list. Rather than simply learning whether their guess differs at all from the secret value, the attacker can now learn *how much* their guess differs from the secret. There is one subtlety, which is that this is only true when the similarity follows from the low to high indices; for example, they learn quite a bit about the password `weak` when they provide the guess `weaa`, but less with the guess `aeak`, despite the comparable differences.

Let's think about how to leverage this leakage to reduce the attacker's work from $O(2^k)$ to $O(k)$ (here k is the length of the secret). Consider the following attack, where N is the number of possible characters that each position in a password or PIN number can take. For the program above, this is 256, because each character is a byte.

1. First the attacker learns the length of the secret. They provide guesses `a`, `aa`, `aaa`, ..., until they observe greater than 1 step of time from `fastmatch`.
2. Knowing the length of the secret, they try all N variants of the guess: x`aa`..., where x is one of the N possible first characters. They store the amount of time that `fastmatch` took on each guess in t_x^0, and conclude that,
$$\texttt{b}[0] = \arg\max_x t_x^0$$
3. They continue with the second element, guessing all N variants of `b[0]`x`aa`..., and again take the one that required the greatest number of steps (execution time) to return.
4. Continue appending characters that result in the longest execution of `fastmatch` until it returns `true` when it finishes.

What is the complexity of this attack? Assuming that elements are coded in binary so there are $\log(N)$ bits in each element, recall that the brute-force approach that would have been necessary without the timing information required $2^{\texttt{a\_len}\cdot\log(N)} = N^{\texttt{a\_len}}$ queries. With the timing information in our example, i.e. the exact number of steps needed to return, each element takes at most N guesses to find, and so now the attack will finish in $\texttt{a\_len} \cdot N$ queries to `fastmatch`. In short, timing information reduced an exponential attack into a linear one.

Ramifications. In practice, it may seem as though this isn't a problem. Attackers can't know actual number of instructions executed by a program, and it may seem that the actual timing differences of a few additional comparison operations will hardly be distinguishable on real systems. Also, how often is an equality comparison pivotal for truly secret data or secure operation?

Many have studied the first question, and the results have shown consistently and overwhelmingly that timing differences introduced by just a handful of instructions are in fact distinguishable—even over the Internet! Crosby et al. found that by observing multiple runs, on the order of hundreds or a few thousands, timing differences as small as 20μs could be reliably distinguished over the open Internet, whereas on a less-noisy LAN timings as small as 100 nanoseconds are distinguishable.

Regarding the second issue, equality comparisons crop up all over the place, and in some cases if you simply make use of the built-in equality operator in languages like Python, you may be using an optimized implementation like `fastmatch`. The developers of Keyczar, a recently-deprecated crypto library from Google, learned this when they made use of built-in comparison functions in Python and Java to verify hash-based message authentication codes (HMAC's) Lawson [2009]. In short, their use of a leaky comparison function enabled attackers to forge authenticated messages with a few million queries over the Internet. While this may seem like a lot, it is a modest cost in absolute terms, and not sufficient to protect real systems.

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 Hoffmann et al. [2011]. 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, 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*

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

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

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

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

$$\frac{\langle \omega, P\rangle \Downarrow^{r}_{\mathbb{B}} \mathit{false}}{\langle \omega, \texttt{while}(P)\,\alpha\rangle \Downarrow^{r} \omega} \qquad \frac{\langle \omega, P\rangle \Downarrow^{r_1}_{\mathbb{B}} \mathit{true} \quad \langle \omega, \alpha; \texttt{while}(P)\,\alpha\rangle \Downarrow^{r_2} \omega'}{\langle \omega, \texttt{while}(P)\,\alpha\rangle \Downarrow^{r_1+r_2} \omega'}$$

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.

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:

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

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.

$$\forall \omega_1, \omega_2 . \omega_1 \approx_{\mathrm{L}} \omega_2 \land \langle \omega_1, c\rangle \Downarrow^{r_1} \omega_1' \land \langle \omega_2, c\rangle \Downarrow^{r_2} \omega_2' \implies r_1 = r_2 \land \omega_1' \approx_{\mathrm{L}} \omega_2' \tag{3}$$

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 $\texttt{if}(Q)\,\alpha\,\texttt{else}\,\beta$, 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 $\Gamma \vdash Q : \texttt{H}$ 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 $\texttt{H}$.

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 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 $\ell \not\sqsubseteq \texttt{L}$ from flowing to the runtime of a program α. As an added

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

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

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

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

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

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 $\omega_1 \approx_{\mathtt{L}} \omega_2$,*

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

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 Ngo et al. [2017].

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 Almeida et al. [2016].

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 $|\texttt{h}|$, 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 elements are equal, and the arrays are the same length, then `fastmatch` returns `true`. In other words, `fastmatch` is nothing more than a conjunction over equality literals, which we can implement quickly by aggregating the exclusive-or of each element, along with the lengths, and returning true whenever the result is 0 (i.e., all of the bits in both arrays are identical).

```
bool fastmatch(byte a[], size_t a_len, byte b[], size_t b_len) {

  size_t matched = a_len ^ b_len;

  for (size_t i = 0; ((i < a_len) & (i < b_len)); i++) {
    matched |= a[i] ^ b[i];
  }

  return matched == 0;
}
```

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.

```
bool fastmatch(byte a[], size_t a_len, byte b[], size_t b_len) {

  size_t matched = a_len ^ b_len;

  for (size_t i = 0; ((i < a_len) & (i < b_len)); i++) {
    if(a[i] != b[i])
      matched = 1;
    else
      matched = matched;
  }

  return matched == 0;
}
```

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.

References

Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

Scott Crosby, Dan Wallach, and Rudolf Riedi. Opportunities and limits of remote timing attacks. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):1–29, 2009.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. In *38th Symp. on Principles of Prog. Langs. (POPL'11)*, pages 357–370, 2011.

Nate Lawson. Timing attack in google keyczar library. https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/, 2009.

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.