Lecture Notes on
Timing Attacks

15-316: Software Foundations of Security & Privacy
Frank Pfenning

Lecture 14
October 24, 2024

1 Introduction

Deducing secrets (like passwords or private keys) by observing how long a program runs is a commonly used attack on computer security. This is true despite the uncertainties about how long a program will actually run (which is only stochastically connected to the source code). For example, Brumley and Boneh [2005] demonstrate that remote timing attacks are quite feasible and can be launched against security-critical code such as an implementation of SSL. They had to run a program many times, but in the end they were successful in exploiting data-dependent optimizations in the cryptographic primitives underlying SSL. Due to the serious nature of such security threats, defenses have been devised. The primary ones are (1) introduce randomness into the computation so that the time variations don't give away the secret information, and (2) sharpen the information flow type system to ensure "constant-time" computation. We explore the second and briefly touch on the first. Timing attacks against cryptographic primitives are still being discovered even today.

There are other so-called *side channels* for information. Here are some:

- Power consumption
- Memory access patterns
- Sequence of instructions executed
- Electromagnetic radiation emitted from the hardware

Some of them require direct access to some hardware, other can be launched remotely. What they have in common in that they use physical properties of what takes place in a computer when code is executed that is outside of the usual abstract model of computation we work with when reasoning about our code. This is a significant difference between functional correctness (e.g., “this code will sort a list”) and security. In the latter case we can only prove security against a policy and a particular threat model—we can’t make any blanket statements about attacks that might fall outside the model.

As an example of the complexity of side-channel attacks that occur in real life, we briefly consider Spectre [Kocher et al., 2019]. It exploits a some features of a modern processor, namely speculative execution, together with timing attack on the memory cache. They use the following sample code to illustrate their technique:

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

This code tries to guard access to `array1` to prevent out-of-bounds access. In the first phase of the attack, it is given many valid values for `x`, training the processor’s branch prediction algorithm to believe that this branch will usually be taken. In the second phase, it is given and input that is out of bounds, that is, greater than `array1_size`. Branch prediction will assume the test turns out to be true and starts executing `y = array2[array1[x] * 4096]`. When the processor discovers that the condition is actually false, it will abandon all the actions taken, such as retrieving `array1[x]`, multiplying it to obtain some address `addr`, retrieving a value from `array2[addr]`, and any other registers, condition, codes, etc. However, one thing it does **not** undo is storing some fetched data in the memory cache. Note that the “secret” data at `array1[x]` are used as an address, so the data at this address might still be in the cache. Now in the third phase the attacker launches a timing attack against the cache to determine which memory region is now in the cache, which can reveal the underlying data at `array1[x]`.

2 Some Simple Timing Attacks

Real timing attacks [Brumley and Boneh, 2005] are complex, among other things due to timing variations in processor execution. We abstract away from these details and it turns out that the main defenses, like “constant-time programming”, are ultimately the same. Here are some timing attacks, intentionally somewhat hypothetical (slightly outside our language) to give you the opportunity to devise your own in Lab 2.

With $\Sigma_0 = (pin : \mathsf{H})$, we write

$$\textbf{while}\ (pin > 0)\ pin := pin - 1$$

Assuming the original value of pin (say c) is positive, this will take something like $4{*}c{+}2$ steps to terminate. Here we count every arithmetic operation or comparison as 1 step, the assignment as 1 step, and processing of **while** as another step. While details may vary with the cost model, it should be clear that it is easy to determine the secret value of pin from the running time.

This code, however, does not represent a very useful attack, because it is too slow, for example, if the pin has 256 bits. But we can modify it into a standard strawman example (omitting any details regarding possibly necessary declassification):

```
$auth := 1$ ;
$i := 0$ ;
while $(i < len)$
  if $pin[i] = guess[i]$
  then $i := i + 1$
  else $(auth := 0 ; i = len)$
```

Here $str[i]$ could either access the ith character of a string str or the ith bit of the number. len would represent either the length of the string or the width of the pin in bits.

Let's assume there are nonnegative integers, and len is the number of bits. If the lowest bit of the guess is incorrect, the program will terminate with $auth$ almost immediately. If it is correct, it will go around the loop at least one more time, and therefore take longer. That will allow us to infer the lowest bit of the secret pin. Our next guess will have that bit correct and now find the next bit, etc. With at most $2 * len$ experiments we should be able to determine all bits.

3 Time-Sensitive Noninterference

As in the last lecture, our first job will be to define the correct semantic notion of noninterference and measure it against the examples. We would like that if both programs terminate from prestates that are indistinguishable for a low-security observer, they take the same number of steps and terminate in low-security equivalent states. In order to express the number of steps, we change our evaluation function to return notcc only a poststate, but also the number of steps taken. The latter will be defined shortly.

We define $\Sigma \models \alpha$ secure^t (program α satisfies *time-sensitive noninterference with respect to policy* Σ)
iff

for all ℓ, ω_1, ω_2, ν_1, ν_2, n_1, and n_2,

whenever $\Sigma \vdash \omega_1 \approx_\ell \omega_2$,

and $\mathsf{eval}\ \omega_1\ \alpha = (n_1, \nu_1)$,

and $\mathsf{eval}\ \omega_2\ \alpha = (n_2, \nu_2)$, then $\Sigma \vdash \nu_1 \approx_\ell \nu_2$ and $n_1 = n_2$.

Note that we have disregarded nontermination in this definition. It is easy to combine it with the requirement to be termination-sensitive, but we prefer to separate these concerns for now.

Now our first example (with $\Sigma_0 = (pin : \mathsf{H})$) clearly does not satisfy this definition, which is what we would hope.

$$\alpha_0 = (\textbf{while}\ (pin > 0)\ pin := pin - 1)$$

A simple counterexample is

$$\begin{array}{ll} \omega_1 = (pin \mapsto 0) & \mathsf{eval}\ \omega_1\ \alpha_0 = (2, pin \mapsto 0) \\ \omega_2 = (pin \mapsto 1) & \mathsf{eval}\ \omega_2\ \alpha_0 = (6, pin \mapsto 0) \end{array}$$

We have $\Sigma_0 \vdash \omega_1 \approx_\mathsf{L} \omega_2$ (since $pin : \mathsf{H}$), and the poststates are indistinguishable (not only for a low-security observer, but in general), but the steps $n_1 = 2 \neq 6 = n_2$.

The second example is subject to a similar analysis, but we'd need to be careful about declassification which we would like to avoid.

4 Evaluation with Time

Next we should define evaluation with step-counting. This is quite straightforward, just a bit tedious. You can find the summary of this exercise in Figure 1. We have to recognize that this form of timing isn't realistic and hope that to some extent the design of the information flow type system can make up for this lack of realism.

One point about the Boolean operations: we do not want their running times to be data-dependent. This means that evaluating $\top \wedge P$ and $\bot \wedge P$ should take the same amount of time. In other words, the Boolean operations should not be "short-circuiting". Recall that in a previous lecture we determined that it didn't matter whether they were short-circuiting or not since all expressions and formulas are safe and terminating. In this context it does matter, and steps may need to be taken to ensure data independence.

Timing considerations raise another point about abstractions. So far we have said that variables can hold arbitrary integers. However, this means that the running times of many operations like addition cannot be data-independent. So we assume instead that integers are implemented with a fixed range, like 64 bits or 256 bits and that arithmetic is modular. This creates a number of complications when reasoning about the correctness of code, but it actually makes reasoning about certain security properties (especially those that are timing-sensitive) both more realistic and simpler.

5 A Type System for Constant-Time Computation

Next comes the task to develop a type system that enforces time-sensitive noninterference, which is usually called "*constant-time*". However, it isn't actually constant time, it is just that the running time can only depend on low-security inputs. In order to formalize that, we once again proceed construct by construct. We write $\Sigma \vdash \alpha\ \mathsf{secure}^t$ for time-sensitive security with respect to signature Σ.

Assignment. For now, there doesn't seem to be any reason to change our rule for assignment. The point is that the time for the evaluation of an expression e is some n, regardless of the state ω. So it just once again comes down to the basic information flow properties.

$$\frac{\Sigma \vdash e : \ell \quad \ell' = \Sigma(pc) \sqcup \ell \quad \ell' \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e\ \mathsf{secure}^t} :=F^t, \text{preliminary version}$$

See the end of this section for further thoughts on assignment.

Sequential Composition and skip. If α and β are indistinguishable even via a timing channel, then their composition should not be, either. The proof is straightforward and follows familiar patterns, so we omit it here. **skip** is trivial, as usual.

$$\frac{\Sigma \vdash \alpha\ \mathsf{secure}^t \quad \Sigma \vdash \beta\ \mathsf{secure}^t}{\Sigma \vdash \alpha\ ;\ \beta\ \mathsf{secure}^t} ;F^t \qquad \frac{}{\Sigma \vdash \mathbf{skip}\ \mathsf{secure}^t}\ \mathbf{skip}F^t$$

Conditionals. The first instinct might be that in a conditional, both branches need to take the same amount of time. Under such a discipline, we could rewrite our motivating example as follows:

```
$auth := 1$ ;
$i := 0$ ;
**while** $(i < len)$
  **if** $pin[i] = guess[i]$
  **then** $auth := auth$
  **else** $auth := 0$ ;
  $i := i + 1$
```

The good part here is that we go around the loop the same number of times, regardless whether the guess is correct or not. The bad part is that even though the two branches, abstractly, take the same amount of time that is unlikely to be case in practice. Any reasonable compiler would optimize $auth := auth$ into a no-op and the loop would leak some timing information.

The problem actually goes deeper: in order to have a *type system* for time-sensitive information flow that allows such a program, the type system would have to capture exactly how long an expression or command takes to evaluate. Not only would this be inaccurate with respect to the actually running code, it would also require our type-checker to perform arithmetic reasoning. This would just be too brittle a design.

Instead, we take a more drastic step: for any conditional, we require that the formula P depends only on low-security variables. This rules out our sample program above, because the comparison $pin[i] = guess[i]$ depends on pin which is of high security.

Intuitively, we are repeating the solution for termination-sensitive information flow from the last lecture, but with conditionals instead of loops. Here is the first formulation: we just require ℓ' to be $\bot$, the least element of the security lattice.

$$\frac{\Sigma \vdash P : \ell \quad \bot = \ell' = \Sigma(pc) \sqcup \ell \quad \Sigma' = \Sigma[pc \mapsto \ell'] \quad \Sigma' \vdash \alpha \text{ secure}^t \quad \Sigma' \vdash \beta \text{ secure}^t}{\Sigma \vdash \textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta \text{ secure}^t}\ \mathbf{if}F^t$$

We notice that this requires $\ell = \Sigma(pc) = \bot$. And if $\ell' = \bot = \ell$, there is no need to update the pc, so we can use $\Sigma' = \Sigma$.

$$\frac{\Sigma \vdash P : \bot \quad \Sigma \vdash \alpha \text{ secure}^t \quad \Sigma \vdash \beta \text{ secure}^t}{\Sigma \vdash \textbf{if } P \textbf{ then } \alpha \textbf{ else } \beta \text{ secure}^t}\ \mathbf{if}F^t$$

At this point we realize we don't need pc at all, since we only introduced it to track the implicit flow from the condition into the branches of an if-then-else.

With this simplification, we have drastically reduced the range of programs that are considered secure. So how do we rewrite our example? We have to "inline" the conditional as an ordinary operation. Note that the number of bit (or maximal length of the string) would have to be a low-security variable.

```
$auth := 1$ ;
$i := 0$ ;
$\mathbf{while}\ (i < len)$
  $auth := auth \wedge (pin[i] = guess[i])$ ;
  $i := i + 1$
```

This is just outside the range of what our language permits in that equality produces a Boolean, and conjunction takes two Booleans, but here they are integers. This could be solved any number of ways. Perhaps the simplest is that the Boolean operators are overloaded to also work on integers, for example, with $\bot$ represented by 0 and $\top$ by 1, or by anything nonzero.

Also, we go back and think of pin and $guess$ as strings rather than nonnegative integers, this code still makes sense and is not subject to timing attacks.

Let's prove the soundness of this rule. We set up:

$\Sigma \models P : \bot$ (1, first premise)
$\Sigma \models \alpha\ \mathsf{secure}^t$ (2, second premise)
$\Sigma \models \beta\ \mathsf{secure}^t$ (3, third premise)
$\Sigma \vdash \omega_1 \approx_\ell \omega_2$ (4, assumption)
$\mathsf{eval}\ \omega_1\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) = (n_1, \nu_1)$ (5, assumption)
$\mathsf{eval}\ \omega_2\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) = (n_2, \nu_2)$ (6, assumption)
...
$\Sigma \vdash \nu_1 \approx_\ell \nu_2$ and (to show)
$n_1 = n_2$ (to show)

Because $\Sigma \vdash \omega_1 \approx_\ell \omega_2$ and $\bot \sqsubseteq \ell$, we have that $\mathsf{eval}\ \omega_1\ P = \mathsf{eval}\ \omega_2\ P = (k, b)$ for some nonnegative time k and Boolean b. We consider the case where $b = \top$; the other is symmetric. Then

$$\mathsf{eval}\ \omega_1\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) = (k + m_1 + 1, \nu_1)$$

where $\mathsf{eval}\ \omega_1\ \alpha = (m_1, \nu_1)$ and $n_1 = k + m_1 + 1$.

Also

$$\mathsf{eval}\ \omega_2\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) = (k + m_2 + 1, \nu_2)$$

where $\mathsf{eval}\ \omega_2\ \alpha = (m_2, \nu_2)$ and $n_2 = k + m_2 + 1$.

Since α is secure (that is, $\Sigma \models \alpha\ \mathsf{secure}^t$, which comes from the second premise) we get $\Sigma \vdash \nu_1 \approx_\ell \nu_2$ (which is one of the two properties we needed to prove) and $m_1 = m_2$.

Therefore, also $n_1 = k + m_1 + 1 = k + m_2 + 1 = n_2$, which is the second property we needed to prove.

Loops. For the same reason as conditionals, we require the loop guard to be of low security.

$$\frac{\Sigma \vdash P : \bot \quad \Sigma \vdash \alpha\ \mathsf{secure}^t}{\Sigma \vdash \mathbf{while}\ P\ \alpha\ \mathsf{secure}^t}\ \mathbf{while}F^t$$

Tests. If the test fails and the program aborts we have a different kind of channel. Again, the simplest condition that avoid information flow here (even if outside of the definition) is to require the test to be of low security.

$$\frac{\Sigma \vdash P : \bot}{\Sigma \vdash \mathbf{test}\ P\ \mathsf{secure}^t}\ \mathbf{test}F$$

Assignment Revisited. Since we eliminated the ghost variable pc, we simplify the rule for assignment.

$$\frac{\Sigma \vdash e : \ell \quad \ell \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e\ \mathsf{secure}^t}\ {:=}F^t$$

A summary of the rules can be found in Figure 2. It is remarkably simple because it is not afraid to rule out many programs. The soundness of all the rules taken together gives us the soundness of the whole type system.

Theorem 1 (Soundness of timing-sensitive information flow typing)
If $\Sigma \vdash \alpha\ \mathsf{secure}^t$ *then* $\Sigma \models \alpha\ \mathsf{secure}^t$

Proof: All the rules are sound in the sense that they preserve semantic validity of the premises. By a trivial induction over the derivation of $\Sigma \vdash \alpha\ \mathsf{secure}^t$ we obtain $\Sigma \models \alpha\ \mathsf{secure}^t$. □

Comparing the type system against the one from the last lecture for termination-sensitive noninterference, we see that it actually enforces that as well. One can update the definition of noninterference and statement of the above theorem to account for that.

6 Randomization

Another defense against timing attacks is to introduce randomization into the program. Even though an observer might still be able to observe timing, this information will then be insufficient to determine the secret. In our example (using the interpretation of pins and guesses as large integers) we could write something like:

```
$r := \mathsf{random}()$ ;
$pin := pin + r$ ;
$guess := guess + r$ ;
$auth := 1$ ;
$i := 0$ ;
**while** $(i < len)$
  **if** $pin[i] = guess[i]$
  **then** $i := i + 1$
  **else** $(auth := 0 \; ; \; i = len)$
```

Every time this program is run, it is like to use a different random number, essentially making it impossible to draw useful conclusions from the running times. Some care must be taken for the randomization and its result to be sufficiently uniform.

References

David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, August 2005.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Symposium on Security and Privacy (SP 2019)*, pages 1–19, San Francisco, California, May 2019. IEEE.

$$
\begin{array}{lcl}
\mathsf{eval}_{\mathbb{Z}}\ \omega\ c & = & (0, c) \\
\mathsf{eval}_{\mathbb{Z}}\ \omega\ x & = & (1, \omega(x)) \\
\mathsf{eval}_{\mathbb{Z}}\ \omega\ (e_1 + e_2) & = & (n_1 + n_2 + 1, c_1 + c_2) \\
 & & \text{where } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_1 = (n_1, c_1) \\
 & & \text{and } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_2 = (n_2, c_2) \\
 & & \\
\mathsf{eval}_{\mathbb{B}}\ \omega\ (e_1 \le e_2) & = & (n_1 + n_2 + 1, c_1 \le c_2) \\
 & & \text{where } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_1 = (n_1, c_1) \\
 & & \text{and } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e_2 = (n_2, c_2) \\
\mathsf{eval}_{\mathbb{B}}\ \omega\ (\top) & = & (0, \top) \\
\mathsf{eval}_{\mathbb{B}}\ \omega\ (\bot) & = & (0, \bot) \\
\mathsf{eval}_{\mathbb{B}}\ \omega\ (P \wedge Q) & = & (n_1 + n_2 + 1, b_1 \wedge b_2) \\
 & & \text{where } \mathsf{eval}_{\mathbb{B}}\ \omega\ P = (n_1, b_1) \\
 & & \text{and } \wedge\mathsf{eval}_{\mathbb{B}}\ \omega\ Q = (n_2, b_2) \\
 & & \\
\mathsf{eval}\ \omega\ (x := e) & = & (n + 1, \omega[x \mapsto c]) \\
 & & \text{where } \mathsf{eval}_{\mathbb{Z}}\ \omega\ e = (n, c) \\
\mathsf{eval}\ \omega\ (\alpha\ ;\ \beta) & = & (n_1 + n_2 + 1, \nu) \\
 & & \text{where } \mathsf{eval}\ \omega\ \alpha = (n_1, \mu) \\
 & & \text{and } \mathsf{eval}\ \mu\ \beta = (n + 1, \nu) \\
\mathsf{eval}\ \omega\ (\mathbf{skip}) & = & (1, \omega) \\
 & & \\
\mathsf{eval}\ \omega\ (\mathbf{if}\ P\ \mathbf{then}\ \alpha\ \mathbf{else}\ \beta) & = & (k + n + 1, \nu) \\
 & & \text{where } \mathsf{eval}\ \omega\ P = (k, \top) \text{ and } \mathsf{eval}\ \omega\ \alpha = (n, \nu) \\
 & & \text{or } \mathsf{eval}\ \omega\ P = (k, \bot) \text{ and } \mathsf{eval}\ \omega\ \beta = (n, \nu) \\
\mathsf{eval}\ \omega\ (\mathbf{while}\ P\ \alpha) & = & (k + n + 1, \nu) \\
 & & \text{where } \mathsf{eval}\ \omega\ P = (k, \bot) \text{ and } n = 0 \text{ and } \nu = \omega \\
 & & \text{or } \mathsf{eval}\ \omega\ P = (k, \top) \text{ and } \mathsf{eval}\ \omega\ (\alpha\ ;\ \mathbf{while}\ P\ \alpha) = (n, \nu)
\end{array}
$$

Figure 1: Timed Evaluation

$$\frac{\Sigma \vdash e : \ell \quad \ell \sqsubseteq \Sigma(x)}{\Sigma \vdash x := e \ \mathsf{secure}^t} \ {:=}F^t$$

$$\frac{\Sigma \vdash \alpha \ \mathsf{secure}^t \quad \Sigma \vdash \beta \ \mathsf{secure}^t}{\Sigma \vdash \alpha \ ; \beta \ \mathsf{secure}^t} \ ;F^t \qquad \frac{}{\Sigma \vdash \mathbf{skip} \ \mathsf{secure}^t} \ \mathbf{skip}F^t$$

$$\frac{\Sigma \vdash P : \bot \quad \Sigma \vdash \alpha \ \mathsf{secure}^t \quad \Sigma \vdash \beta \ \mathsf{secure}^t}{\Sigma \vdash \mathbf{if} \ P \ \mathbf{then} \ \alpha \ \mathbf{else} \ \beta \ \mathsf{secure}^t} \ \mathbf{if}F^t \qquad \frac{\Sigma \vdash P : \bot \quad \Sigma \vdash \alpha \ \mathsf{secure}^t}{\Sigma \vdash \mathbf{while} \ P \ \alpha \ \mathsf{secure}^t} \ \mathbf{while}F^t$$

$$\frac{\Sigma \vdash P : \bot}{\Sigma \vdash \mathbf{test} \ P \ \mathsf{secure}^t} \ \mathbf{test}F$$

Figure 2: Timing Sensitive Information Flow Typing