15-316: Software Foundations of Security and Privacy **Spring 2017**

Lecture 19: Side Channel Leakage

Lecturer: Matt Fredrikson

19.1 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 $\sigma_\mathsf{L}, \sigma'_\mathsf{L}$. 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.

19.2 Example: Optimized *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.

```
let rec fastmatch(h, l) =
  match h with
    | [] ->
      match l with
      | [] -> true
      | y::ys -> false
  | x::xs ->
    match l with
      | [] -> false
      | y::ys -> if (x = y) then fastmatch(xs,ys) else false
```

What is wrong with this implementation? Nothing, 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 `fastmatch`.

However, if we change the observation model to include the amount of time it takes the function to return `true` or `false`, 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 should 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.
- Therefore, the attacker knows that the longer `fastmatch` executes, the more elements she has successfully guessed at the beginning of the list.

Can we use this intuition to significantly decrease the exponential-time attack bound we studied last week?

1. Let N be the number of possible elements in the lists h and l. We'll assume that this is finite, such as 256 (i.e., lists of ASCII characters) or 2^{64} (i.e., machine integers).
2. First, the attacker tries all passwords `[x]`, for $0 \leq x \leq N$, and notes the amount of time taken t_x.
3. After iterating over all x, the attacker takes the first element to be $x_1^* = \arg\max_x t_x$
4. The attacker can return to step 2, trying all guesses for the second character: $[\texttt{x}_1, \texttt{x}]$ for $0 \leq x \leq N$.
5. The process finishes whenever `fastmatch` returns `true`.

What is the complexity of this attack? Let L be the length of the high-security list, 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, and so obviously poses a big problem.

19.3 Constant-time programming discipline

Looking at the previous example, let's try to generalize from what went wrong to develop a broadly-applicable approach to avoiding such timing leaks. Obviously, the fact that the runtime of the program is influenced by high-security data is the direct cause of the problem. In general, this problem 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 are likely to exist.

Question. *Is secret-dependent control flow either a necessary or sufficient condition for the existence of a timing channel?*

Question. *Suppose that we wanted to design a timing channel mitigation technique that disallows secret-dependent control flow. What assumptions would we need to make in order for such an approach to be sound?*

19.3.1 Writing `fastmatch` in constant-time

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.

To fix this timing channel, we'll switch from OCaml to an imperative language (assume that it's C, but we might take liberties with certain things that the C compiler would reject), as constant-time programming is usually done in these languages. An equally-vulnerable C implementation might be as follows.

```
uint8 fastmatch(const uint8 *h, const uint8 *l, uint len) {
  int i = 0;
  while(i < len) {
    if(h[i] != l[i]) return 0;
    i++;
  }
  return 1;
}
```

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 `h[i] == l[i]`, for $0 \leq \texttt{i} < \texttt{len}$, then `fastmatch` returns 1. If `h[i] == l[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 our loop.

```
uint8 fastmatch(const uint8 *h, const uint8 *l, uint len) {
  int i = 0;
  uint8 r = 1;
  while(i < len) {
    r &= h[i] == l[i];
    i++;
  }
  return r;
}
```

Now that we've seen one way of fixing the timing channel in `fastmatch`, one might naturally wonder why we couldn't have written it differently, perhaps still including a conditional statement.

```
uint8 fastmatch(const uint8 *h, const uint8 *l, uint len) {
  int i = 0;
  uint8 r = 1;
  while(i < len) {
    if(h[i] != l[i]) r = 0;
    else r = r;
    i++;
  }
  return r;
}
```

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 `r = r` 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.

19.3.2 Cache leakage

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.

This type of side channel was famously exploited by Dan Bernstein 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 **exclusive or**. 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.

Fixing the problem. 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.

Question. *Based on what we've discussed about the workings of AES, how might you implement it using this new constant-time discipline? What are the tradeoffs that you are likely to encounter when doing so?*

19.4 Formalizing side channel security

Thinking back to when our observation model was simply the low-security portions of the initial and final states, $(\sigma_\mathsf{L}, \sigma'_\mathsf{L})$, we formalized information flow security as noninterference:

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx_\mathsf{L} \sigma_2 \wedge \langle \sigma_1, c \rangle \Downarrow \sigma'_1 \wedge \langle \sigma_2, c \rangle \Downarrow \sigma'_2 \implies \sigma'_1 \approx_\mathsf{L} \sigma'_2 \tag{19.1}$$

This worked out nicely, because the observations $(\sigma_\mathsf{L}, \sigma'_\mathsf{L})$ are accounted for directly by the semantic relation $\Downarrow$. Now our observations have grown to include either the number of steps taken during execution, or the sequence of memory accesses made. How do we account for these quantities?

Cost semantics. One natural approach is to enrich the semantics with precisely these quantities. 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 or memory usage. To see how this works, recall our original semantic relations for expressions and commands.

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

This notation means that executing expression e (resp. command c) in environment σ yields value v (resp. state σ'). 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.

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

This notation means that executing expression e (resp. command c) in environment σ yields value v (resp. state σ') 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, r could correspond to a list of memory addresses in the case where we want to reason about cache side channel security.

An example cost semantics is shown in Figure 19.1, corresponding to the observation of the number of execution steps taken to execute an expression or command. Although the language we've studied thus far does not contain arrays or pointers, we could extend it appropriately, and define a cost semantics that concatenates the address of each lookup to a list-valued cost annotation in order to reason about cache side channels.

Question. *The cost semantics shown in Figure 19.1 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?*

Formalizing security. With the cost semantics in hand, we can now go about formalizing what it means for a program to be secure with respect to leakage through a given side 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 19.1, and write:

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx_\mathsf{L} \sigma_2 \wedge \langle \sigma_1, c \rangle \Downarrow_{r_1} \sigma'_1 \wedge \langle \sigma_2, c \rangle \Downarrow_{r_2} \sigma'_2 \implies r_1 = r_2 \tag{19.2}$$

$$\textsc{Const}\ \frac{}{\langle \sigma, n\rangle \Downarrow_1 n} \qquad \textsc{Var}\ \frac{}{\langle \sigma, x\rangle \Downarrow_1 \sigma(x)} \qquad \textsc{Op}\ \frac{\langle \sigma, a_1\rangle \Downarrow_{r_1} n_1 \quad \langle \sigma, a_2\rangle \Downarrow_{r_2} n_2 \quad n = n_1\ \mathbf{op}\ n_2}{\langle \sigma, a_1\ \mathbf{op}\ a_2\rangle \Downarrow_{r_1+r_2} n}$$

$$\textsc{Asgn}\ \frac{\langle \sigma, a\rangle \Downarrow_r n}{\langle \sigma, x := a\rangle \Downarrow_{r+1} \sigma[x \mapsto n]} \qquad \textsc{Seq1}\ \frac{\langle \sigma_1, c\rangle \Downarrow_r \sigma_2}{\langle \sigma_1, \mathbf{skip}; c\rangle \Downarrow_r \sigma_2} \qquad \textsc{Seq2}\ \frac{\langle \sigma_1, c_1\rangle \Downarrow_{r_1} \sigma_1' \quad \langle \sigma_1', c_2\rangle \Downarrow_{r_2} \sigma_2}{\langle \sigma_1, c_1; c_2\rangle \Downarrow_{r_1+r_2} \sigma_2}$$

$$\textsc{IfT}\ \frac{\langle \sigma, b\rangle \Downarrow_{r_1} \mathbf{T} \quad \langle \sigma, c_1\rangle \Downarrow_{r_2} \sigma_2}{\langle \sigma, \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\rangle \Downarrow_{r_1+r_2} \sigma_2} \qquad \textsc{IfF}\ \frac{\langle \sigma, b\rangle \Downarrow_{r_1} \mathbf{F} \quad \langle \sigma, c_2\rangle \Downarrow_{r_2} \sigma_2}{\langle \sigma, \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\rangle \Downarrow_{r_1+r_2} \sigma_2}$$

$$\textsc{WhileF}\ \frac{\langle \sigma, b\rangle \Downarrow_r \mathbf{F}}{\langle \sigma, \mathbf{while}\ b\ \mathbf{do}\ c\rangle \Downarrow_r \sigma} \qquad \textsc{WhileT}\ \frac{\langle \sigma, b\rangle \Downarrow_{r_1} \mathbf{T} \quad \langle \sigma_1, c\rangle \Downarrow_{r_2} \sigma_1' \quad \langle \sigma_1', \mathbf{while}\ b\ \mathbf{do}\ c\rangle \Downarrow_{r_3} \sigma_2}{\langle \sigma_1, \mathbf{while}\ b\ \mathbf{do}\ c\rangle \Downarrow_{r_1+r_2+r_3} \sigma_2}$$

Figure 19.1: Step-execution cost semantics for the simple imperative language. The costs indicate the number of steps needed to execute the program in a particular environment, and correspond to non-negative integers.

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 19.2 doesn't account for observation of the low-security final state, but we can easily add this as follows.

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx_{\mathsf{L}} \sigma_2 \wedge \langle \sigma_1, c\rangle \Downarrow_{r_1} \sigma_1' \wedge \langle \sigma_2, c\rangle \Downarrow_{r_2} \sigma_2' \Longrightarrow r_1 = r_2 \wedge \sigma_1' \approx_{\mathsf{L}} \sigma_2' \tag{19.3}$$

Question. *Given these formal definitions of 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?*