EECS 498-007 Project

Adam Austerberry

2022

Abstract

Side-channel attacks are a common class of attacks that plauge cryptographic systems. They're especially notorious since they can be present even on systems that have perfectly sound theory and no bugs in their implementation. This project intends to mitigate some side-channel attacks proactively by designing a domain-specific language to write code that is forced to be less vulnerable.

1 Introduction

Nearly all cryptographic systems rely on the assumption that some or all of the participants have access to memory in which other participants cannot access and the ability to perform calculations without other participants seeing. The presence of side-channel attacks makes this much more challenging to achieve in the real world than it may seem. Side-channel attacks exploit hard-to-notice implementation details that cause a system to behave differently in an externally visible manner based on secret data. By monitoring these behaviors, an attacker is able to reverse engineer what the secret data is. Side-channels attacks are a significant yet subtle threat, because they can be present even when the theory behind a system is perfectly sound and the implementation is bug-free.

The purpose of this project is to define a domain-specific programming language that can force the programmer to write code which behaves as identically as possible regardless of the secret data it is operating on. The reason it wouldn't be useful to make it behave perfectly identically is because it would have to produce the same results regardless of the inputs, which isn't very useful.

The purpose of this project isn't to prevent every side-channel attack. Doing so is well outside of the scope of this project, and may even be impossible to do perfectly. Attacks that are able to extract data directly out of memory (such as Meltdown, cold-boot), will not be mitigated. Attacks such as timing attacks, will.

There has been a lot of previous work on preventing side-channel attacks, since they represent a significant threat the security of real-world computer systems. The difference with that work is that it tends to follow a cat-and-mouse design pattern. An attack is discovered, then a patch for that particular attack is applied. Another attack is discovered, then another patch for that

particular attack is applied. This kind of work can be very effective, since the patches can be tailored very specifically to prevent or mitigate an attack, but it's not a proactive approach. The attacks themselves have to be discovered before they can be defended against.

2 Related Works

2.1 Discovery of Side-channel Attacks

This is by no means an exhaustive list of side-channel attacks that exist, but it provides a diverse selection of the different flavors they tend to come in.

- Cache attacks: An attacker shares a cache with the target and is able to tell what the target is accessing by determining what's in the cache, often by timing it. An example of this attack was FLUSH+RELOAD [7].
- Timing attacks: An attacker is able to reverse-engineer what the target is doing by timing how long it takes to complete a task. This is often aided by feeding maliciously crafted inputs to the target to time it on. An example of this is attack were the timing attacks on various cryptographic systems discovered by Paul C Kocher [4].
- Acoustic attacks: An attacker determines what target hardware is doing simply by listening from a distance. An example of this attack was demonstrated to be able to extract RSA keys by Genkin, Shamir, and Tromer [2].
- Data remanence attacks: An attacker is able to extract traces of data that has ostensibly been deleted due to the physical imperfect deletion process. An example of this is attack was the Cold Boot attack by Halderman et al [3].

2.2 Branch-free Code

A branch occurs in a program whenever it has to make a decision about what to do next, such as with an if-statement or a loop. The reasons why one would invest time in writing branch-free code usually comes down to either performance or side-channel mitigation, but for now we will focus on side-channel mitigation.

Branching is used to cause a program to behave differently than it might on a different execution, which can open it to side-channel attacks if the decision to branch was based on secret data. If this affects the time it takes to execute, it can result in a timing side-channel. If this affects when/where memory is accessed, it can result in a cache side-channel. Even the instructions being executed can emit sounds that can be picked up and decoded by a microphone.

A program without branches is not vulnerable is this way. But, branches are useful constructs. If a program is limited to never branch based on secret data, it will also be good. [6]

2.3 Security-Focused Programming Languages

- Cryptol: Cryptol is a domain-specific language for writing cryptographic code. It allows the programmer to define a formal model and reference specification of their cryptosystem, then compile it to be linked with C, C++, Haskell, or VHDL/Verilog. It is proprietary to Galois, Inc [1].
- F*: F* is a general-purpose language designed for proof-oriented programming. Programmers write metatheorems that describe what they want their software to do. Then when writing the code for it, they also have to write a proof that their code meets the metatheorems. If the code doesn't meet the metatheorems, it won't compile. F* is a part of Microsoft Research's Project Everest, with the goal of implementing a formally verified implementation of TLS. Although suitable for cryptographic purposes, it isn't exclusive to cryptography [5].

3 Results

The domain-specific language enforces two main constraints: Do not branch based on secret data, and do not access memory based on secret data.

3.1 Branching

In order to prevent branching based on secret data, the language will contain no constructs that usually cause a conditional branch, other than if-else-statements. The if-else-statements themselves will be implemented in such as way as not to branch. Instead of running the code in only one of the blocks, both blocks will be run. Two bitmasks will be generated for the two results based on the conditional, then applied to the results of the computations. At this point, the result we want has stayed as it is, and the result we don't want has been masked to all zeros. By bitwise ORing them, we just get the result we want.

```
fn non_branching_if(c: bool, a: &dyn Fn() -> i32, b: &dyn Fn() -> i32) -> i32 {
    let result_a = a();
    let result_b = b();

    let mask_a = -(c as i32);
    let mask_b = c as i32 - 1;

    let masked_a = mask_a & result_a;
    let masked_b = mask_b & result_b;

    masked_a | masked b
}
```

It may be noted that if either `a` or `b` have side effects (an effect other than returning a value), this is not guaranteed to behave identically to a branch-

ing if-else-statement. To solve this problem, we can make our language to be purely functional, which prohibits side-effects. Running these computations then throwing away the value they return is semantically identical to not running them at all.

As a purely functional language, loops won't be possible, so recursion will substitute for it. Recursion will work via fixed points, which are expressions that are capable of containing variables that represent copies of themselves. For example:

```
fix x -> x + x
evaluated by one "step" becomes:
(fix x -> x + x) + (fix x -> x + x)
```

Recursion will continue forever unless at a certain point, the code branches in such a way not to recurse again. In a language without conditional branches, this is impossible. Even our if-else expressions won't save us because they still evaluate everything, even if they don't actually use the results.

To prevent evaluation from taking forever, fixed points will have a limit attached to them. This limit is specified at compile-time. Whenever a fixed point is evaluated, its descendants' limit will be one less than its limit.

```
fix (5, x) -> x + x
evaluated by one "step" becomes:
(fix (4, x) -> x + x) + (fix (4, x) -> x + x)
```

If a fixed point with a limit of zero is evaluated, we'll refer to this as an overflow. Such a fixed point does not evaluate its body. Instead it evaluates to a default value that has the same type that the body would have evaluated to. This type consistency is necessary so that bitwise masking and combination can take place in an if-else expression higher up the expression tree.

It will also be necessary for every evaluated value to have a Boolean flag indicating whether they are the result of a fixed point overflow. If this overflowed value is used in any expression where it's necessary to the result of evaluation, the result of evaluation is also flagged as overflowed, since this result is based on an arbitrary (incorrect) default evaluation.

```
(10, overflowed) + (42, valid)
evaluated by one "step" becomes:
(52, overflowed)
```

```
if true { (10, overflowed) } else { (42, valid) }
evaluated by one "step" becomes:
(10, overflowed)
```

The only case in which the result of an evaluation that contains an overflowed value as an input isn't itself flagged as overflowed is if that overflowed input is in the discarded branch of an if-else expression. It's necessary to have this catch, otherwise it would be impossible to write a recursive expression that doesn't get flagged as overflowed.

```
if false { (10, overflowed) } else { (42, valid) }
evaluated by one "step" becomes:
(42, valid)
```

It is the programmer's responsibility to determine the maximum number of recursions that will be necessary for their program. It is also their responsibility to make sure an if-else expression catch the overflow flag so that it doesn't propagate up to the top.

3.2 Practical Testing

In order to test this language, I implemented an evaluator for it, as well as a suit of test-cases, which is available on GitHub.

One of the tests was the RSA decryption operation, which is a calculation that involves the user's secret key. First I tested it on my evaluator, but without the strict rules for evaluating if-else expressions, meaning that it branched differently depending on what the user's secret key is. This evaluator took 2850% more time on average to decrypt with a key of 127 compared to a key of 3. This exposes the user's key to a timing attack.

I then tested RSA on the non-branching evaluator. This evaluator took 0.6% less time on average to decrypt with a key of 127 compared to a key of 3. Since 127 was faster than 3 only 48% of the time, this small difference can be attributed to statistical error from external influences, such as OS scheduling. If this is correct, then the amount of time it takes to decrypt a message is independent of the key, and timing attacks are impossible.

3.3 Memory Access

Being purely functional, the programmer is not able to directly access memory. There are no pointers nor arrays. Memory is only accessed based on how expressions are evaluated. Since the same expressions are always evaluated in the same order, it's impossible for program data (which is potentially secret) to inform how memory is accessed.

Constant memory access occurs as a byproduct of all measures taken in the previous subsection.

3.4 Continued Work

- To guarentee constant branching, both blocks of an if-else-statement have to be executed, even if only one of the results is used, which is inefficient. This can be especially inefficient if both blocks contain a recursion. If this happens, an expression that would take linear time with branching code would take exponential time. I would like to find a way to mitigate, or at least detect and warn against such inefficiency.

- Right now, reaching the recursion limit results in an overflowed value being returned. If not caught, this overflow value will propagate to become the

result of the whole computation. I would like to find a way to detect (and possibly) prevent such a thing at compile-time.

- Because this DSL is so simple, the only error that can't be detected at compile-time is an overflow. A static analyzer would be able to detect and alert the programmer to every other kind of error before it was even run. Right now, all errors (type errors, uninitialized variables, etc) are treated identically to overflows (in order not to branch on errors), but this makes understanding why an error has occured quite difficult.

- Although originally designed for the purposes of mitigating side-channel attacks, a constant-branching DSL is also ideal for implementing QAP-based zero-knowledge proofs. Since branching is deterministic, all branches can be unraveled into arithmetic and boolean circuits.

References

[1] Inc. Galois. Cryptol, the language of cryptography, now available. `https://galois.com/blog/2008/12/cryptol-the-language-of-cryptography-now-available/`.

[2] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Cryptology ePrint Archive, Report 2013/857, 2013. `https://ia.cr/2013/857`.

[3] Nadia Heninger William Clarkson William Paul Joseph A. Calandrino Ariel J. Feldman Jacob Appelbaum J. Alex Halderman, Seth D. Schoen and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Association for Computing Machinery*, 2009.

[4] Paul C Kocher. Timing attacks on implementations of diffe-hellman, rsa, dss, and other systems. *CRYPTO*, 1996.

[5] Microsoft Research. Proof-oriented programming in f*. `http://www.fstar-lang.org/tutorial/`.

[6] Chess Programming Wiki. Avoiding branches. `https://www.chessprogramming.org/Avoiding_Branches`.

[7] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.