Side-Channels in Runtime Systems

Tegan Brennan Miroslav Gavrilov

June 7, 2017

1 Introductory thoughts on side-channels

A practically useful system always presents a complicated landscape in an information-theoretical view: to be deemed "useful", computations have certain characteristic behaviours they should follow (e.g. be of a certain complexity class, execute at a certain speed, etc.), which more or less exerts certain properties of the physical implementation of the system in question. By not focusing on the result of the computation, but rather on the side-effects manifest in this physical implementation, one can gain knowledge of some implementational details, and thus the structure of the computation itself. This is referred to as using a side-channel to learn things.

One of the foci of side-channel analyses is preventing side-channel attacks, in which an attacker is using one or potentially more side-channels to discover some private data or gain insight into a hidden process.

This phenomenon builds a kind of philosophical construct similar to the Heisenberg uncertainty principle wherein the limitations of our physical reality are stopping us from being both optimized (take less time in most cases) and, for example, completely computationally private (as observed in [1]). The case in which the least side-channel activity is present is the case in which the lengths of all the possible routes of execution of the main-channel are of similar size, and thus observation becomes noisy in the presence of other physical factors. This, unsurprisingly, is the case in which there is no optimization made and all the lengths are similar to the longest one.

1.1 Project outline

Runtime systems are complex architectures built for speed, and as such are a logical source of many observable side-channels. Given that they operate over different but generally equally-powerful languages, our main field of exploration, as well as our main contribution, covers the following questions:

- How does the runtime influence the side-channels found in a program?
- How do we analyze side-channels in runtime systems?
- How does language design influence observable side-channels?

- How do changes in runtime options (JIT, GC, etc.) influence any observed side-channels?

The answers to these questions could give an idea of how certain implementation details decide on how suitable a runtime or language is for tasks that need a level of discretion in lieu of a potential side-channel, as well as how they could decide some further optimizations.

2 Experiments

We will talk about our answers to these questions through examples, for the sake of brevity. We will briefly go over the experimental setup used, and then go over the contributions we make, in the order given in the outline. The tools we created for this project are introduced along the way, without giving them too much space, although they are fully open-source and referenced.

2.1 Setup

All of our experiments have been run on the Java Virtual Machine (Java HotSpot 64-bit Server), on a 7th generation Intel Core i7 at 3.8GHz with 4 cores and 32GB of RAM.

To showcase side-channels, we select to focus on timing side-channels, as they are common and easily observable by direct measurement. As we're measuring time on a method-call level, we created a driver which, using `System.nanoTime()`, measures various method lengths, via calling a single interface method, which many implementations of our examples override.

For the purpose of clarity, we use several versions of a single program, `PasswordChecker` which is a well-known example of timing side-channels. All source code is listed in the appendix.

`PasswordChecker.checkPassword(input)` is a method with simple semantics: given a public `input`, if a `secret` value exists, not present in any direct input-output flow, the method returns whether `input` is, character for character, equal to `secret`.

Different implementations of `PasswordChecker` have different behaviours. For example, the naïve implementation that would be built for speed, includes early returns as soon as it is clear that either the lengths of the two strings are different, or that a character mismatch happened at a certain position. If we try to represent the time needed by the method to finish, we can write it as $T = t_{len} + n \cdot t_{match} + \epsilon$, where t_{len} is the time needed for checking the lengths of the two strings, t_{match} is the time needed to match two characters, and ϵ is the noise in the measurement. We consider that $\epsilon \ll \texttt{min}(t_{len}, t_{match})$, or that the information lost as a result of ϵ being present is recoverable through taking multiple samples.

We can notice that all the results of our measurements of T can be put into at most l equivalence classes, where $l = \texttt{length}(\texttt{secret})$. By knowing this, we can engineer a side-channel attack where we can generate the `secret` by trying

out characters letter by letter and seeing where we access a higher equivalence class. At this point, we know that we can cement the current letter and repeat the process to get the information about the next one, until the full `secret` has been leaked.

By rewriting this, we can decrease the information leak, at the cost of speed. Our main interest is in exploring ways in which a runtime system can be better or worse for exploiting side-channels, be it by ways of special bytecode, in which side-channel noise is greater by default, special language constraints or constructs, that make programmer-created side-channels harder to make, or side-channel specific optimizations or mechanisms.

2.2 Methods of side-channel analysis in runtime systems

Except for the dynamic analysis via measurements, we recognize that some side-channels can be found by statically analyzing the structure of the program in question. For example, the naïve `PasswordChecker`'s flaw is in the peculiar shape of it's control-flow, where obvious length differences can be observed if we unroll the loop and count how many instructions there are in those branches.

For this reason, we've searched for a tool which had several properties we found important: *i*) functional minimality, *ii*) no dependency overhead, and *iii*) modularity and elasticity. Having found no tools that satisfy all three of these, we decided to build our own[1], which enabled us to extract control-flow graphs at different levels of abstraction. It is written as a server-client tool, in the hopes of accumulating control-flow data in more than one research field and project; its development will be continued.

Once we'd extract the control-flow graph from the interesting method, we'd annotate it, first by the number of instructions, and then in a more complex, symbolic way, which represents all the possible valid costs that a single connection in that graph could have, accounting for loops and other special forms of control-flow. The symbolic cost we get by inductive steps, starting with the instruction cost, and every step is introducing an at-most linear increase in complexity. The connection costs, at this point, can be used as a cost model for the whole method, which we solve using an off-the-shelf SMT-solver[2], basically solving for any valuation of the variables present that would make the difference in the costs of branches larger than some δ.

This static method can inform us of any shape-dictated side-channels occurring in the program, however is weakened by any form of code transformations, be it the JIT, optimizations or, in scripting languages, obfuscation.

2.3 Differences in side-channels due to language design

To explore how different language syntax behaves with side-channels, we compared the implementation of `PasswordChecker` in Java and in Scala. Both

[1] Conflow, https://goo.gl/FnomgF
[2] *Z*3, for example

languages share the JVM, although Scala is usually written in a mostly functional manner. The result of this paradigm shift is that most of the instructions found in Scala bytecode are method calls, presented in the bytecode as artificial anonymous classes, and subject to a large array of Scala analyses and optimizations at compile-time. This causes the side-channel to not appear, if written in the way most Scala programmers would typically align themselves with.

From this, we can see that the choice of valid language syntax and semantics based on formalism and consistency can help improve side-channel prevention, even though it isn't necessarily looking to solve that problem. The main reason behind why the side-channel isn't present in the Scala code is that all traversals through the sequence of characters are designed to be exactly $n * O(1)$, where n is the length of the array, instead of being $O(n)$, as is the case with languages that allow early-returns, as is Java. Furthermore, if an attempt be made to write Java-like loops in Scala, a weak space side-channel could form as well, as the early break mechanisms are implemented via lazy structures, thus holding the rest of the calculation in memory while evaluating the start of the sequence. It follows that good performance strongly follows good practice in this case.

2.4 Runtime options and their effect on side-channel strength

We have tested multiple versions of the `PasswordChecker` application on the JVM described above, but with different options. As our application doesn't use a lot of memory, checking any garbage collection behaviour was pointless, but we did try toggling the JIT compiler, to measure its influence on our program.

Once we noticed that without the JIT enabled, both performance and our side-channel disappeared, we had to dig deeper.

References

[1] Jason Oberg, Sarah Meiklejohn, Timothy Sherwood, and Ryan Kastner. *Leveraging Gate-Level Properties to Identify Hardware Timing Channels* http://ieeexplore.ieee.org/document/6879637/?reload=true