Lane Gramling
EECS 700
Reading 21

Summary - Eliminating Timing Side-Channel Leaks using Program Repair

This paper focuses on exploring how to robustly eliminate leaks of information from a program's execution through timing side-channels. To conquer this, the authors explore a proposed implementation of a new layer of static analysis they have developed to guard against the leaking of secret variables from timing side-channels. Their method works by inputting the program with a list of secret inputs, transforming and outputting a functionally identical program except that all timing leaks potentially related to the values of the secret inputs have been obfuscated to have no discernible difference in CPU cycle count or memory access times.

Contributions

The first contribution of this paper is the robustly defined threat model that exhausts scenarios where a program's execution time is affected by differences in the content being executed in relationship with specific input values. Two prominent examples of these include execution times on unbalanced conditional jumps, and memory accesses in a lookup table. The first of these can indirectly reveal information about a secret input if the condition relies on a secret input's value, and the two branches are unbalanced in execution time. The second of these can reveal information about a secret input through the timing of memory accesses, if a secret input is at least related to a lookup table index. In this case, the difference in timing for cache hits & misses can be used to indirectly gain information on a secret input's value. The authors explain in their proposal that their effort is fundamentally different from most timing leak mitigation techniques in this way since they typically focus only on instruction-level timing leaks like unbalanced conditional branches as side-channels to mitigate, and do not make an effort to mitigate timing leaks enabled through the tangible differences in memory access times between cache hits and misses. Beyond this, the paper contributes a much deeper elaboration on the issue with timing leaks due to caching in memory accesses, which I am going to have to read a couple more times and stew on for a bit to understand more fully. To wrap up the exploration of their implementation, the authors contribute data backing up their claims of mitigating timing leaks. In the provided data on the tests, it is seen that there is no timing information surrounding the designated inputs that significantly differs based on execution and thus timing information resulting from instruction executions and memory accesses can't be used to differentiate anything meaningful about the sensitive inputs.

Limitations

One apparent issue with this technique is that it slows down execution, which could be amplified in situations exhibiting a worst-case scenario for the amount of code or memory accesses that need obfuscation to guard against timing leaks. In general cases, however, the execution time does not seem to be significantly executed, when considering the added security benefit. Along with this, it seems to eliminate potential benefits of performance optimizations in execution granted in particular paths through the logic where the path is determined by the value of a secret input. This is because of how the analysis captures and obfuscated timing differences in instructions when sensitive variables are present in conditionals having unbalanced branches. This technique certainly seems most viable in cases where timing information could provide particularly damning details to expose values of secret inputs, and the software in question necessitates a high requirement for security. Additionally, this attack does not provide security against adversaries with access to hardware and capabilities to probe it. Since the data contents are assumed to be guarded from the user, this extra layer of security is only beneficial when the adversary is resource-limited to a layer above it.

Future Work

Future work on this could include working to find other ways around this from an adversarial position, as is a natural response to a paper outlining a defense technique. For example, patterns could be emergent in timing information of instructions that are obfuscated, based on the enumerated techniques used to consume extra CPU cycles; however, the explained technique does have the added advantage of restricting itself to the domain of timing leakage mitigation and thus should be logically impenetrable in ideally defensible scenarios, and can then afford to dodge blame for failing to prevent the exploitation of present vulnerabilities by other side-channels or attacks.