Side-Channels in Runtime Systems: Literature Survey

Tegan Brennan Miroslav Gavrilov

April 17, 2017

Introduction

This document will frame primarily the information extracted from the papers related to the topic at hand, and then our thoughts and conclusions about the papers, as we go through them.

1 D. Page – A Note on Side-Channels Resulting from Dynamic Compilation

1.1 Main Goals

The main goal of this paper is to determine whether the dynamic compilation of Java programs can introduce side channel vulnerabilities into the executed code when there were none in the original code.

In attempting to extract higher levels of perfomancing, a dynamic compiler might transform a secure program into an insecure one. This is particularly concerning as the programmer might have no idea that such a vulnerability was introduced.

1.2 Case Study – Elliptic Curve Cryptography

The double-and-add method is an additive version of binary exponentiation used in ECC based systems. Generally, the profiles of the addition and doubling phases are distinguishable from one another. An attacker can thus determine the sequence of additions and doubling, allowing them to recover the secret key used in encryption. One way to eliminate this side channel is to always perform both an addition and doubling, but this is very inefficient. Another, more commonly used approach, is to split the more expensive addition step into two parts, each of which resembles the doubling step in terms of execution profile.

A java implementation of this side channel resistant version of the double-and-add program was analyzed in order to determine the effect of optimization

on side channel vulnerabilities. It was run on the Jikes Research Virtual Machine (RVM) with the default options for the adaptive optimization system. Because the doubling method was called more frequently than the addition methods, it was identified as a hotspot in the program and was optimized by the RVM. The addition method was eventualy optimized as well, but nevertheless there is a window in which the optimized double method is in use at the same time was the unoptimized addition method. The profiles of these two methods are very distinguishable. Even once the addition method is optimized, the two are distinguishable, though less so, due to differing results of optimization. The RVM adaptive optimization system therefore introduced a side channel vulnerability when there was none in the original code.

1.2.1 Possible Solutions

One possible patch highlighted in the paper is to allow code fragments to be annotated with information that allows the virtual machine to avoid introducing vulnerabilities into the code. A basic solution would be to include an annotation that instructs the dynamic compiler to leave the associated method alone. This of course trades performance for security. Another approach would be to specify that dynamic compilation is permitted provided that the two methods still match each other in terms of cost. While more work is needed to determine how such a compilation phase might occur, there is already pre-existing work that allows for the detection of any mismatch between the results of re-compilation that might be reusable for this task.

1.3 Conclusion

Optimizations introduced by dynamic compilation can indeed introduce side channels into code when there was none in the souce.

2 S. Crane, A. Homescu, et al – Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity

2.1 Main Goals

The main question of this paper is whether dynamic compilation can be used to reduce potential side-channel information leakage?

Here, software diversity is explored as a defense against side-channel attacks through dynamically randomizing the control flow of programs. Traditionally, software diversification focuses on randomizing the program's representation, such as the in-memory addresses of data. However, side-channel attacks rely on dynamic properties of programs such as execution time and memory latencies. This calls for a software diversification technique that randomizes the program's execution as opposed to representation.

Pre-existing diversification techniques may already have an impact on side-channels. For example, even re-ordering functions can have a large effect on cache usage and performance. However, static compile time or load time diversity is not enough since an attacker can simply use the static target binary to generate a profile of the runtime characteristics of the program. Even re-diversifying and switching to a new variant during execution is insufficient as side-channels are fast enough to complete between re-diversification cycles. Instead, the authors create a diversification model which creates a version of the program consisting of replicated code fragments with randomized control flow to switch between alternative replicas at runtime. The program can now take numerous different paths which prevents the attacker from developing a reliable model of program behavior.

The randomized variants of a program fragment are made using diversifying transformations. This results in functionally equivalent program fragments that have differing runtime characteristics. These program fragments are then integrated into a program that dynamically chooses a control flow path at runtime.

2.2 Case Study – Mitigating Cache Side-Channel Attacks

Pre-existing side-channel attacks against the AES encryption scheme exploit cache behavior to determine the secret key. The authors implemented these attacks against AES-128 routine in libcrypt 1.6.1. Two cache based side-channel attacks were evaluated, one which was able to recover 96% and the other 82% of the secret bits for a set number of iterations in the base case. Code diversification was demonstrated to have an impact on the success rate of both attacks. Static diversification performed decently, reducing the number of correctly guessed bits to $104 - 108$ of 128 for the first attack but to 52 for the second. However, the dynamic approach presented by the authors mitigated the side-channels much more powerfully, reducing the number of correctly guessed bits to 20 for the first attack, and to 14 for the second.

2.3 Conclusion

This in itself demonstrates the potential for compiler-based code diversification to protect against side-channel attacks, which could be a promising avenue for greater security.