Optimizing Away Security in C

Project Report, Language-Based Security, Fall 2023, Aarhus University

Anders B. Clausen Johan T. Degn Jonathan Eilath

{clausen, johandegn, eilath}@post.au.dk

June 2023

Abstract

TODO: After we have finished the paper

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Contents

1 Introduction

The C programming language is one of the most widely used programming languages in the world. It is used in a wide variety of applications, ranging from embedded systems to cryptographic libraries. However, C is also notorious for lacking security guarantees. Security-related issues in programs written in C stem from both the programmer, but also from the compiler.

The problem with security issues introduced by the compiler is especially prevalent in the context of timing attacks on cryptographic algorithms. Developers try to mitigate timing attacks by writing constant-time code – code where the execution time is independent of the input. However, the compiler may introduce timing leaks by optimizing away the constant-time code. In a recent study by Jancar et. al. [4], it was shown that the vast majority of developers of cryptographic libraries rely on constant-time code practices that in theory result in constant-time code, but may be vulnerable after compilation.

The issues related to timing vulnerabilities introduced by compilers are well-known in the community. Timing vulnerabilities have previously been linked to certain compiler optimizations [2]. They have also been compared across compiler versions [6]. However, to the best of our knowledge, no study has quantified the issue of timing attacks introduced by the compilers.

In this paper, we try to quantify the issue of timing attacks introduced by the two most popular C compilers, gcc[1] and clang[2]. We do this by implementing a tool, OptiFuzz, that generates, analyzes and fuzzes random C programs. For simplicity, the generated programs are limited to only containing non-branching arithmetic, logical and comparison operations. Also `/` and `%` are avoided, to mitigate division-by-zero errors. We investigate what optimization flags are responsible for introducing timing leaks and compare them. Finally, we discuss how the issue of timing attacks introduced by the compiler can be mitigated by using language-based security techniques.

1.1 Related Work

The issue of timing attacks introduced by the clang C compiler across different versions has been researched by Simon et. al. [6]. Several studies have investigated potential solutions to the issue, including using constant-time branching instructions [6], using black-box testing software [5], and notably a verified constant-time C compiler has been developed [1].

1.2 Contributions

We provide a quantitative analysis of timing vulnerabilities introduced by gcc and clang, focusing on specific troublesome optimization flags. Additionally, we provide a tool, OptiFuzz, that can be used to generate, analyze and fuzz random C programs for further investigation of the issue. Finally, we provide a discussion of how the issue of timing attacks introduced by the compiler can be mitigated by using language-based security techniques found in other literature. TODO: Might be more when we finish the paper

1.3 Overview

This paper is organized as follows: ... TODO: After we have finished the rest of the paper TODO: Insert guide on how to read appendix here??

[1] `https://gcc.gnu.org/`.
[2] `https://clang.llvm.org/`.

2 Preliminaries

2.1 C Compiler Optimzations

2.2 Side-Channel Attacks

3 OptiFuzz

Note: OptiFuzz might be used in a CI pipeline to detect timing vulnerabilities automatically.

3.1 Code Generation

The first step in the OptiFuzz pipeline is code generation. The code generation module is written in OCaml and works by generating abstract syntax trees according to the grammar in Figure 1.

$$
\begin{aligned}
e \in \mathit{Expr} ::= & x \mid y && \text{(input variables)} \\
& n \in \{0,1\}^{64} && \text{(64-bit integer literals)} \\
& -e \mid e + e \mid e - e \mid e \times e && \text{(arithmetic operators)} \\
& \texttt{true} \mid \texttt{false} && \text{(boolean literals)} \\
& !e && \text{(logical operators)} \\
& e < e \mid e \leq e \mid e > e \mid e \geq e \mid e = e \mid e \neq e && \text{(comparison operators)} \\
& e \& e \mid e \mid e \mid \sim e \mid e \hat{} e \mid e \ll e \mid e \gg e && \text{(bitwise operators)}
\end{aligned}
$$

Figure 1: The grammar that defines the ASTs generated by the code generation module in the OptiFuzz pipeline.

The grammar defines all programs with 2 variables and non-branching arithmetic, logical and comparison operations in C [3], excluding division (`/`) and modulus (`%`). The reason for excluding division and modulus is that they may cause division-by-zero errors. Generated programs are forced to include both input variables, x and y. x and y are the inputs to the program, and we require both to be present since programs with 0 or 1 input variables are trivially constant-time. Furthermore, Booleans are defined as the integer literals 0 and 1 just like in the C standard library `stdbool` [3]. An example of a program generated by the code generation module is shown in Figure 2.

```
#define false 0
#define true 1
int program(int x, int y) { return !(y * (43 * (x != true))); }
```

Figure 2: Example of a program generated by the code generation module.

The code generation module works by generating a random distribution that selects different symbols in the grammar with a certain probability. This ensures that not all generated programs will be uniformly random. For example, if a generated distribution heavily favors the left-shift operator, then the generated programs will contain a lot of left-shift operations. This is useful for detecting whether certain types of programs are more likely to be constant-time than others.

The grammar in Figure 1 generates programs that are not typical

TODO: Argue why not a more realistic grammar was chosen TODO: Find examples of weird real life programs

TODO: Future Work: More complicated programs

Bibliography

[1] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371075. URL `https://doi.org/10.1145/3371075`.

[2] Vijay D'Silva, Mathias Payer, and Dawn Song. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*, pages 73–87, 2015. doi: 10.1109/SPW.2015.33.

[3] ISO. *Programming languages - C ISO/IEC 9899:2018*. 2018.

[4] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 632–649, 2022. doi: 10.1109/SP46214.2022.9833713.

[5] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1697–1702, 2017. doi: 10.23919/DATE.2017.7927267.

[6] Laurent Simon, David Chisnall, and Ross Anderson. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–15, 2018. doi: 10.1109/EuroSP.2018.00009.

Appendix