Optimizing Away Security in C

Group 10
Anders B. Clausen **Johan T. Degn** **Jonathan Eilath**

May 23, 2023

Abstract

TODO: After the paper is done

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 [Jancar et al., 2022], 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 issue of timing vulnerabilities introduced by compilers is well-known in the community. Several proposals have been made to mitigate the issue [Simon et al., 2018, Reparaz et al., 2017, Cauligi et al., 2017, Barthe et al., 2019]. However, the problem persists and to the best of our knowledge, no study has quantified the issue of timing vulnerabilities introduced by compilers.

In this paper, we try to quantify the issue of timing vulnerabilities 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 vulnerabilities introduced by the compiler can be mitigated by using language-based security techniques.

1.1 Related Work

The issue of timing vulnerabilities introduced by the clang C compiler across different versions has been researched by Simon et. al. [Simon et al., 2018]. Several studies have investigated potential solutions to the issue, including using constant-time branching instructions [Simon et al., 2018], using black-box testing software [Reparaz et al., 2017], domain-specific languages [Cauligi et al., 2017], and notably a verified constant-time C compiler has been developed [Barthe et al., 2019].

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. 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??

2 Preliminaries

2.1 Timing Attacks

Timing attacks are a class of side-channel attacks that exploit the fact that the execution time of a program depends on the input. The history of timing attacks goes back several decades where Kocher showed multiple

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

successful timing attacks on well-known cryptographic algorithms such as Diffie-Hellman and RSA [Kocher, 1996]. An example of vulnerable code is shown in Figure 1.

```
int foo(int x) {
  if (x < 100) {
    x *= 2;
    x += 7;
    return x;
  } else {
    return x;
  }
}
```

Figure 1: Example of a program vulnerable to a timing attack. By analyzing the execution time of the machine code, an attacker can infer whether the input is less than 100 or not.

2.2 Optimizing Compilers

Cryptographers will avoid code like the example in Figure 1 and write constant-time code instead. Constant-time code is code where the execution time is independent of the input. However, the compiler may introduce timing vulnerabilities through optimizations by adding variable-time branches to the machine code. The issue arises in the analysis and transformation phases of the compiler as illustrated in Figure 2.

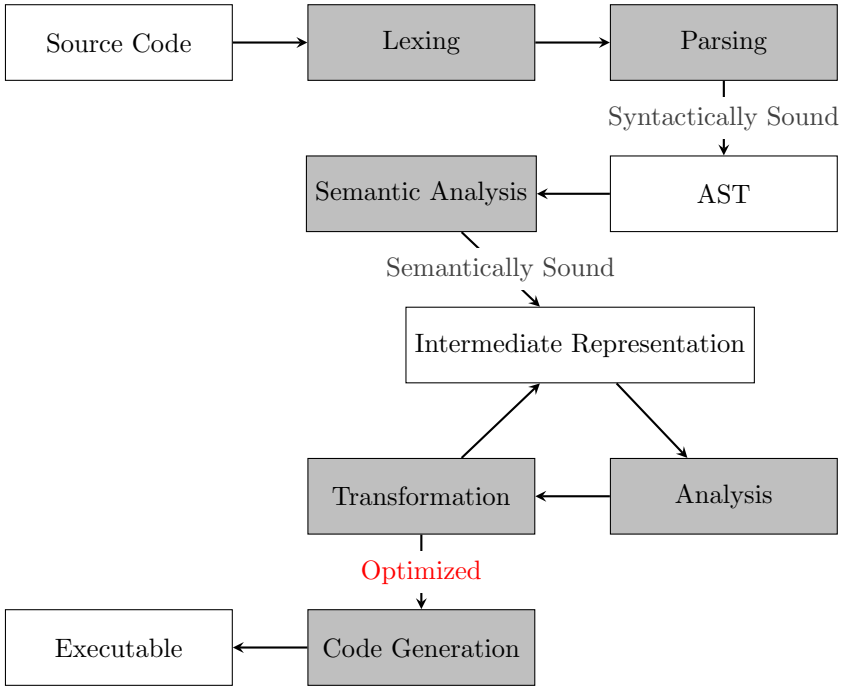

Figure 2: The pipeline of an optimizing compiler. After the transformation phase, the IR is optimized and the compiler may have introduced timing vulnerabilities.

Many different kinds of optimization techniques are carried out by optimizing compilers, some of which can introduce timing vulnerabilities. Some common optimization techniques like common subexpression elimination and strength reduction have been shown to introduce timing vulnerabilities [D'Silva et al., 2015]. To illustrate this point, we look at how common subexpression elimination can introduce timing vulnerabilities.

2.2.1 Timing Vulnerabilities Through Common Subexpression Elimination

Common Subexpression Elimination is an optimization technique that extracts subexpressions that are common across multiple expressions and replaces them with a single variable. This optimization technique can introduce timing vulnerabilities since it can decrease the number of instructions executed for a specific branch of the code as illustrated in Figure 3. Here the common subexpression `2 * 3 + 5` is extracted and assigned to the variable `common`, making the `else` branch faster than the `if` branch.

```c
int foo(int x, int arr*) {
  if (x == SECRET) {
    x = arr[0] * 3 + 5;
    x += arr[1] * 3 + 5;
    x += arr[2] * 3 + 5;
  } else {
    x = 2 * 3 + 5;
    x += 2 * 3 + 5;
    x += 2 * 3 + 5;
  }
  return x;
}
```

(a) Original code.

```c
int foo(int x, int arr*) {
  if (x == SECRET) {
    x = arr[0] * 3 + 5;
    x += arr[1] * 3 + 5;
    x += arr[2] * 3 + 5;
  } else {
    // optimized
    common = 2 * 3 + 5;
    x = 3*s;
  }
  return x;
}
```

(b) Optimized code.

Figure 3: An example of how the common subexpression elimination can introduce timing vulnerabilities in code. (a) shows the original code and (b) shows the optimized code.

3 OptiFuzz

We created a tool, OptiFuzz, that can be used to generate, analyze and fuzz random C programs. The source code for OptiFuzz is available on GitHub[3]. The goal of OptiFuzz is to quantify the issue of timing attacks introduced by C compilers with different optimization flags enabled. The tool works as follows:

- OptiFuzz generates random C programs consisting of non-branching arithmetic, logical and comparison operations.
- OptiFuzz then compiles the generated C programs with different specified optimization flags enabled and inspects the generated assembly for conditional branching instructions introduced by the compiler. If branching is found, the program is flagged.
- OptiFuzz then fuzzes the flagged programs with various random inputs to test whether the branching instructions can be exploited to leak information about the input.
- At last, OptiFuzz reports the results of the fuzzing in the form of a PDF report.

The OptiFuzz pipeline is illustrated in Figure 4. Each of the steps in the pipeline is described in detail in the following sections.

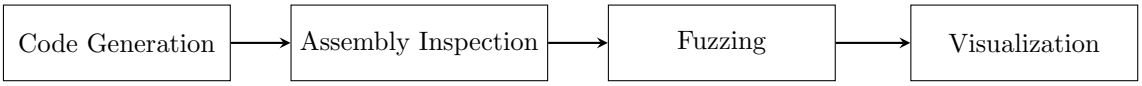

Figure 4: The OptiFuzz pipeline.

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 5.

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

Figure 5: 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 [ISO, 2018], 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

[3] `https://github.com/anbclausen/optifuzz`.

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. An example of a program generated by the code generation module is shown in Figure 6.

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

Figure 6: 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 code generation module generates programs with integer literals in different ranges. Booleans represent the first range. Defining Booleans as integers is a common practice in C [ISO, 2018]. Integers from this range are included since the constants 0 and 1 are interesting in many operations. The second range we consider is $[0, 64]$ as numbers in this range are lower than the size of 64-bit integers. Hence numbers in this range are interesting in bit-shifting operations. The third range is the range of signed 64-bit integers. Generally, we have included these smaller ranges of integers since they are interesting and it is very unlikely that a uniformly random 64-bit integer would be in these ranges.

The grammar in Figure 5 generates programs with undefined behavior. For example, bit-shifting with a negative number, or a number larger than the size of the type, is undefined behavior in C [ISO, 2018]. We chose to include undefined behavior in the grammar since it is a source of timing vulnerabilities [Simon et al., 2018], and it is common in real-life code [Wang et al., 2012]. The grammar also does not distinguish operations on Booleans and integers and hence generates untypical C programs. For example, the program in Figure 6 features multiplication between a Boolean and an integer. We chose to include this in the grammar since it is a used trick to avoid branching in constant-time code [Simon et al., 2018].

3.1.1 Limitations

The biggest limitation of this approach is that the generated programs are not representative of real-life code since they use such a limited subset of the C language. However, as argued above the simple constant-time programs are representative of how real code might look, and thus gives a useful insight into the issue of timing vulnerabilities introduced by the compiler.

3.2 Assembly Inspection

The next step in the OptiFuzz pipeline is assembly inspection. The assembly inspection module is written in Python and works by inspecting the assembly generated by the compiler across different optimization flags. The compiled assembly is inspected for conditional branching instructions and flagged if that is the case. The assembly inspection module is only able to analyze x86 assembly.

In x86 assembly, `Jcc` (note, this does not include `JMP`), `LOOP` and `LOOPcc` are the only conditional branching instructions [Intel, 2023]. This means that the assembly inspection module only needs to look for these instructions.

3.2.1 Limitations

An obvious limitation of this approach is that it only works for x86 assembly. Furthermore, the programs that are flagged by the assembly inspection module are not necessarily vulnerable to timing attacks. Hence, the assembly inspection module overapproximates the set of programs that are vulnerable to timing attacks. For example, the program in Figure 7 is flagged by the assembly inspection module, but it is constant-time since both branches take the same amount of time to execute.

```
...
  cmpl    $1, -4(%rbp) ; compare TOS to 1
  je      .L2          ; jump to .L2 if equal
  movl    $43, %eax    ; move 43 into %eax
  jmp     .L3          ; goto end
.L2:
  movl    $0, %eax     ; move 0 into %eax
.L3:
  ret
```

Figure 7: Example of a program that is flagged by the assembly inspection module, but is constant-time.

3.3 Fuzzing

TODO: Refer to the fact I just wrote that the assembly inspection overapproximates

3.3.1 Limitations

TODO: Underapproximates since fuzzing might not be perfect/noise

3.4 Visualization

4 Experimental Results

TODO: Note: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8077809` says clang had issues with not compiling to cmov back in the day (2017)

5 Conclusion and Future Work

TODO: Future Work: More complicated programs

References

G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. 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`.

S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan. FaCT: A Flexible, Constant-Time Programming Language. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 69–76, 2017. doi: 10.1109/SecDev.2017.24.

V. D'Silva, M. Payer, and D. 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.

Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2023. URL `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

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

J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. 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.

P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68697-2.

O. Reparaz, J. Balasch, and I. 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.

L. Simon, D. Chisnall, and R. 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.

X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined Behavior: What Happened to My Code? In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316699. doi: 10.1145/2349896.2349905. URL `https://doi.org/10.1145/2349896.2349905`.

Appendix