Optimizing Away Security in C

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

June 1, 2023

Abstract

Developers strive to write secure cryptographic code that is resistant to various attacks, including timing attacks. It is a well-established fact that C compilers can introduce timing vulnerabilities into programs through optimizations, even when the source code is seemingly secure. However, to our knowledge, the extent of this problem is quantitatively unknown: How bad is it? In this paper, we present OptiFuzz, a tool that can be used to test optimizing compilers for timing vulnerabilities. OptiFuzz has been designed considering the C language specification, and the x86-64 architecture, and includes state-of-the-art statistical analysis. This has resulted in a simple tool that can reliably indicate the extent of the real-world issue of timing vulnerabilities introduced by optimizing compilers. Using OptiFuzz, we have generated and compiled around 500,000 random C programs and shown that between 0.15% and 12.63% of these programs are vulnerable to timing attacks when compiled with `gcc` depending on the configured level of optimization. Contrarily, we show that `clang` introduces timing vulnerabilities much more rarely. This has led us to pinpoint specific optimizations within `gcc` that are responsible for introducing timing vulnerabilities, as well as specific expression patterns that are vulnerable to resulting in timing vulnerabilities when compiled. In light of this result, we have discussed how the issue of timing vulnerabilities introduced by compilers can be mitigated by using language-based security techniques from the literature.

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. Developers try to get around the compiler optimizations by obfuscating their code. However, the obfuscated code might still be vulnerable to an optimization that introduces a timing vulnerability. In a recent study [Jancar et al., 2022], it was shown that the vast majority of developers of prominent open-source 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 [Reparaz et al., 2017, Cauligi et al., 2017, Simon et al., 2018, 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` and `clang` on the x86-64 architecture. We take a data-based approach, implementing a tool, OptiFuzz, that generates, fuzzes and analyzes random C programs. For simplicity, the generated programs are limited to only containing non-branching arithmetic, logical and comparison operations. During the analysis phase, OptiFuzz applies automatic state-of-the-art statistical analysis of the data to detect timing leaks. We use our results from OptiFuzz to investigate what optimization flags are responsible for introducing timing leaks and quantify the extent of the problem. 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 [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], using 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. We also provide a tool, OptiFuzz, that can be used to generate, fuzz and analyze random C programs for further investigation of the issue. Finally, we provide a discussion of how the issue of timing vulnerabilities introduced by the compiler can be mitigated based on our findings.

1.3 Overview

This paper is organized as follows: In Section 2, we provide some background knowledge of timing attacks and optimizing compilers. In Section 3, we provide a detailed description of our tool, and the methodology used to acquire our results. In Section 4, we present our experimental results. In Section 5, we evaluate our tool and methodology, and the quality of our results. In Section 6, we discuss various possible solutions to the issue based on other research. Furthermore, we provide some of our raw results in the appendix.

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 can depend on the input. The history of timing attacks goes back several decades where Kocher showed multiple 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.

```
1 int foo(int x) {
2   if (x < 100) {
3     x *= 2;
4     x += 7;
5   }
6   return x;
7 }
```

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. Code is constant-time if the execution time of the program 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.

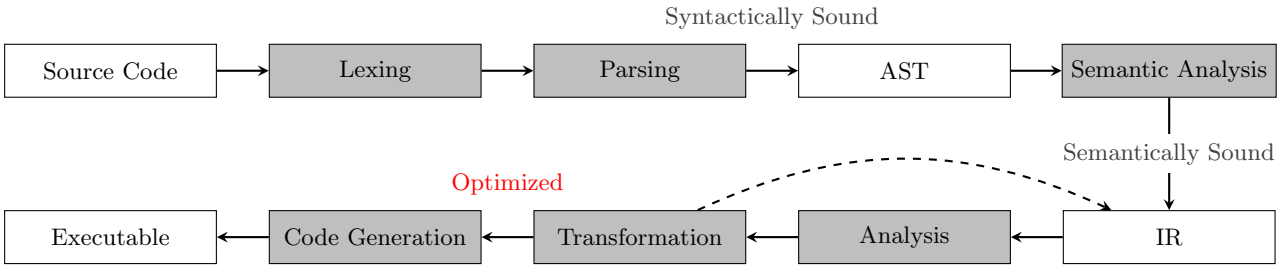


Figure 2: The pipeline of an optimizing compiler. After the transformation phase, the IR is optimized and the compiler may have introduced timing vulnerabilities.

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 execute faster than the `then`-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
    int common = 2 * 3 + 5;
    x = 3 * common;
  }
  return x;
}
```

(b) Optimized code.

Figure 3: An example of how the common subexpression elimination can introduce timing vulnerabilities in code.

3 OptiFuzz

We created a tool, OptiFuzz, that can be used to generate, fuzz and analyze random C programs. The source code for OptiFuzz is available on GitHub[1]. 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 applies Welch's t-test on the fuzzing data to detect whether a timing vulnerability is present. Furthermore, 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 subsections.

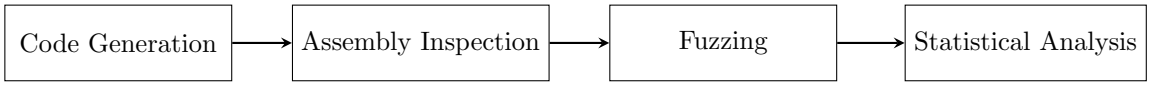


Figure 4: The OptiFuzz pipeline.

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], 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,

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

x and y. We require both to be present since it decreases the likelihood that the program will be optimized into a constant. An example of a program generated by the code generation module is shown in Figure 6.

```
#define false 0
#define true 1
int program(long long int x, long long 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. Non-uniformly random programs, particularly shorter ones, increase the likelihood of encountering certain prevalent operator(s). If the program happens to be variable in time, it may highlight a specific scenario where the operator could pose timing leakage issues. For example, if a generated distribution heavily favors the left-shift operator, then the generated programs will contain a lot of left-shift operations.

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]. 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 in computation and it is very unlikely that a uniformly random 64-bit integer would fall into one of 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 bit-size of the type, is undefined behavior in C [ISO]. 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 [Cauligi et al., 2017, 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 x86-64 assembly.

In x86-64 assembly, `Jcc` (note, this does not include `JMP`), `LOOP` and `LOOPcc` are the only conditional branching instructions [Intel, 2023b]. This means that the assembly inspection module only needs to look for these instructions. Both `Jcc` and `LOOPcc` refer to families of instructions where `cc` is a condition code. For example, `JE` (jump if equal) is in the `Jcc` instruction family. We did not include conditional move instructions in the analysis since they are constant-time [Coppens et al., 2009].

3.2.1 Limitations

An obvious limitation of this approach is that it is tailored for x86-64 assembly. Hence our analysis will not work for other architectures. 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.

3.3 Fuzzing

After the assembly inspection has performed the static analysis and flagged programs with potential timing vulnerabilities, OptiFuzz tries and confirm their presence. For this, we have created a fuzzer. In this section, we explain the process of compiling, running, and timing the programs. This process is illustrated in Figure 8. The process goes as follows: First, the fuzzer itself is compiled into object files (simplified to a single file in Figure 8) that are not yet linked. This compilation happens once. Each of the programs flagged by the assembly inspection is then compiled with the specified compiler and optimization flags. After a program has been compiled it is linked with the fuzzer and is run.

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

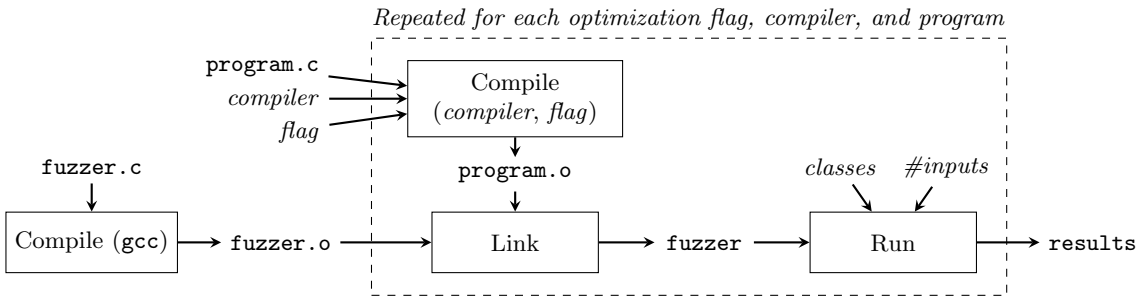


Figure 8: Illustration of the process of compiling and fuzzing programs. The fuzzer is compiled once, after which the process in the dashed box is repeated with different programs, compilers, and flags emitting results.

When running the fuzzer, it generates inputs for the program according to 'input classes' dictating the nature of the generated values. Each input is independently generated according to a uniformly random class among the supplied ones. The fuzzer then executes the linked program for each of the inputs and measures their execution time.

We use different input classes for generating inputs for the program. This is done to try and capture deviating execution times that only take place for a small sample of the values. If we only used uniformly random 64-bit numbers it would be very unlikely that we would identify such a vulnerability. Hence, the need for input classes. As mentioned in Section 3.1, for each program, there are an x and a y value passed as arguments. Each input class generates such a pair of inputs according to the following list:

- `UNIFORM`: x and y are uniformly random 64-bit numbers.
- `EQUAL`: x and y are the same uniformly random 64-bit number.
- `MAX64`: Either x or y is the largest positive 64-bit number, the other is uniformly random.
- `XZERO`: x is 0, y is a uniformly random number.
- `YZERO`: y is 0, x is a uniformly random number.
- `XLTY`: Two uniformly random numbers are generated, x is set to the smaller, y to the larger.
- `YLTX`: Two uniformly random numbers are generated, y is set to the smaller, x to the larger.
- `SMALL`: x and y are uniformly random positive 8-bit numbers (the upper 56 bits are set to 0).
- `FIXED`: x and y have the same fixed number $0x12345678$ (used for Welch's t-test).

These classes have been chosen based on observations of types of values commonly used in conditions in the emitted assembly code.

Precise measurements of a program's execution time are not trivial to perform. Measurements can be influenced by several things including context switches, interrupts, out-of-order execution, varying CPU clock speed, and congestion. For timing, we use the Time Stamp Counter (`TSC`) as our source[2]. The `TSC` is a high-resolution counter that (on newer machines) increases at a fixed rate independent of processor frequency and is the most efficient wall clock resource according to [Intel, 2023b, b]. Alternatives like `HPET` with higher resolutions exist but read the time from a platform resource instead of a register, making it much slower [Intel, 2023b, b]. This method is the one suggested by [Paoloni, 2010] and is the one used in the DudeCT black-box testing tool by [Reparaz et al., 2017]. This approach also allowed us to avoid the timings being skewed by the CPU performing out-of-order execution. Out-of-order executions could lead to the instructions being reordered resulting in not reading the `TSC` (with the `RDTSC(P)` instruction) at the correct time. This is mitigated by utilizing the non-privileged `CPUID` instructions serializing capabilities that ensure that memory transactions for previous instructions are completed before the next instruction is executed [Intel, 2023b, a]. Using `CPUID` and `RDTSCP` allows us to enclose the program call between our time measures using `RDTSC(P)` as wanted.

[2]The Time Stamp Counter measures time in units of crystal clocks [Intel, 2023b, c]. We refer to this unit as CPU clocks.

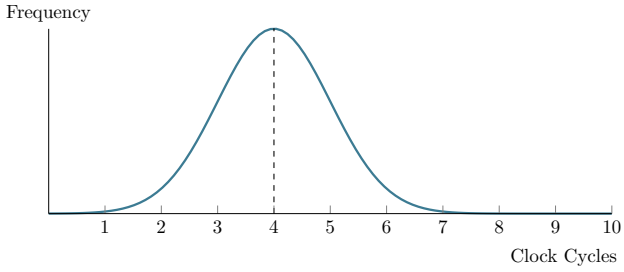
3.3.1 Limitations

One limitation is that the input classes are not necessarily able to capture an event causing variable-time behavior of a program, and hence the fuzzer underapproximates the number of programs vulnerable to timing attacks. Another limitation is that the timings produced are still prone to some noise. Specifically, when running this in user-land the fuzzer could be influenced by interrupts and preemption (OS forces context switches). To combat these, one is required to run the code in kernel space as suggested by [Paoloni, 2010]. We experimented with running in kernel space but found that fuzzing became significantly more inefficient and that noise was still present. It is however possible to mitigate some of this noise by running the fuzzer with higher priority.

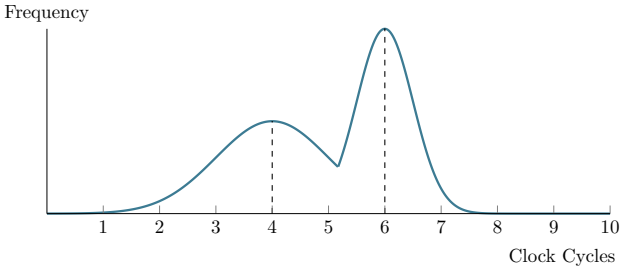
3.4 Statistical Analysis

At the end of the pipeline, we have at our disposal flagged C programs, which have been fuzzed and timed. Now we have to conclude whether or not these programs are variable-time or not. This is a very complex task, as a lot of factors can affect this. Notably, the research conducted by Abel and Reineke demonstrates the difficulty of this, as determining the performance of instructions is very microarchitecture dependant. Additionally, Almeida et al. [2016]'s work suggests that static analysis on LLVM can be done, but again suffers from architecture-specific modeling. Thus, we determined that our method should not consider all the intricate details of architecture-specific implementations. Instead, a statistical approach was deemed more suitable.

One statistical approach is to apply significance tests between classes of fuzzing inputs. If we can detect a difference in time between different kinds of inputs, then we can conclude with good probability, that the program is not constant-time. Ideally, for constant-time programs, we would expect a univariate Gaussian with low variation when we measure the running time of the program for all kinds of inputs. Likewise, ideally, for variable-time programs, we hope to see a mixture distribution composed of two (or more) Gaussians with different means. For instance, if we observe two means: $\mu_1 \neq \mu_2$, then we can conclude with good probability, that most likely a branching instruction on the input dictates whether or not the program will have a running time close to μ_1 or μ_2.



(a) A constant-time program yielding a univariate Gaussian with $\mu = 4$.



(b) A variable-time program, yielding a mixture distribution composed of two Gaussian distributions having means: $\mu_1 = 4, \mu_2 = 6$ with different variances.

Figure 9: An example of what we would expect in theory, when we fuzz constant-time and variable-time programs.

As seen above in Figure 9a, we have an example of what we, in theory, would think the clock-cycles distribution for a constant-time program would look like. Likewise, Figure 9b shows what the distribution could look like for a variable-time program.

Wishing our measurements will yield such distributions, and then detecting timing leakage is, however, not as simple as described above. Coron et al. introduced significance test techniques in general leakage detection; including both timing and power consumption leakage attacks. In the paper, they argue that there is a correlation between measured time and external parameters [Coron et al., 2004]. As an illustrative example, we could use input class A for 10 minutes followed by 10 minutes of fuzzing with input class B while recording the corresponding timing outcomes. It is important to consider that the 10-minute duration of fuzzing with class A might have triggered certain system mechanisms such as a built-in thermal throttle for the CPU, resulting in a reduction of processing speed. Consequently, when evaluating the second input class B it may exhibit a distinct mean value due to the altered conditions induced by the previous measurements. There are a lot of external events to consider that might introduce noise to our data – so the authors' suggested guideline is to alternate between classes when we measure. This essentially makes sure that the external noise is equally applied to both classes.

The previously mentioned tool in 3.3, DudeCT, by Reparaz et al. works in the above fashion. They, however, extended Coron et al.'s approach, by not only interleaving the input classes but by randomly choosing one. According to the authors, this should further reduce noise.

Our methodology extends the above, where we further try to reduce noise, by fuzzing multiple times and picking the minimum measured time. This is visualized in Figure 10 below.

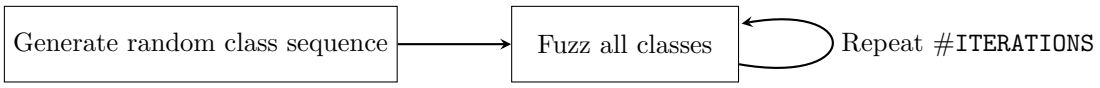


Figure 10: Noise reduction by picking the minimum measurement of multiple runs.

For example, if we in our random class sequence {A, B, B, C} do two iterations:

{A: 2, B: 3, B: 5, C: 2}
{A: 4, B: 3, B: 4, C: 3}

where A: 2 corresponds to the program given input from input class A took 2 clock cycles to run, then the final saved measurements would be: {A: 2, B: 3, B: 4, C: 2}. Note that the random class sequence samples once and fixes the input for all iterations.

Using the minimum time as the true execution time instead of the mean filters out the noise from outliers [Chen and Revels, 2016]. In comparison to DudeCT by Reparaz et al., which just removes the top 5% of the longest measured execution times to reduce such noise, our approach is not prone to accidentally removing measurements of large execution times that are correct and not the result of noise.

Now we apply a significance test to our measurements. There is a great consensus that Welch's t-test [Welch, 1947] works great for the task of leakage detection [Durvaux and Standaert, 2015]. The test tests the null hypothesis that two populations have equal means, which aligns exactly with our described objectives. The strengths of utilizing Welch's t-test are that it is robust to different variances as seen in figure 9b, and that its sampling complexity is low. According to a testing methodology for side channel resistance validation by Goodwill et al., comparing only two of our input classes `FIXED` and `UNIFORM` is sufficient. This is also what is used in DudeCT [Reparaz et al., 2017] with a critical value of $t = 10$. This is a quite high threshold for the test, resulting in constant-time programs being flagged as variable-time with a very low probability (Type I errors). However, the drawback is an increased Type II error, where variable-time programs will not show up as variable-time.

Our methodology takes a more conservative approach, where instead of fixing a critical value, we set $p = 0.05$. This significance level will in turn give us an expected Type I error of 5%; however, the test will perform better when dealing with variable-time programs.

Finally, the minimal measurements are plotted by generating TikZ with a corresponding program assembly, an indicator showing whether the null hypothesis was rejected, and a list of conditional jump instructions are highlighted. We also plot means for different input classes, aiding in identifying whether or not the program has timing leakages. Note that our plot trims outliers by removing the top 5 percentile, however, the t-test is applied to all measurements. See Appendix A for examples.

4 Experimental Results

We have run OptiFuzz to generate around 500,000 programs, which have been fuzzed and analyzed using different compilers and optimization flags. In this section, we present our results.

We have divided this section into four subsections. First, we compare `gcc` and `clang` generally in terms of the number of timing vulnerabilities introduced. Second, we present our results for general optimization flags such as `O2` and `Os`. Third, we present our results regarding specific optimizations. Finally, we present our results regarding what operations are most likely to cause timing vulnerabilities when optimized. All experiments were carried out on an Intel Core i7-9750H running Manjaro Linux 22.1.2 with kernel version Linux 5.15.112-1. For compilers, `clang` version 15.0.7 and `gcc` version 12.2.1 were used. All experiments were run on a single core and with `-20 niceness` to minimize interference from other processes.

4.1 gcc vs. clang

Surprisingly, we were unable to find any timing vulnerabilities introduced by `clang` under any optimization flags. This is in contrast to `gcc`, where we saw timing vulnerabilities across various optimization flags. Specifically, we compiled 50,000 random programs with `clang` using various optimization flags (`O0`, `O1`, `O2`, `O3` and `Os`) out of which none contained conditional branching instructions. The reason for `clang` not introducing any timing vulnerabilities is the compiler's utilization of `cmovcc` instructions. `cmovcc` is a family of constant-time conditional move instructions [Intel, 1995].

Even though our results show that `clang` does not introduce any timing vulnerabilities, our research shows that timing vulnerabilities were identified in fairly recent versions of `clang` [Cauligi et al., 2017, Simon et al., 2018]. Furthermore, we have identified an optimization step in the most recent version of the LLVM backend, which

`clang` uses, that removes `cmovcc` instructions and substitutes them for conditional branches if an efficiency gain can be obtained with high confidence [LLVM, 2023].

Evidently, `clang` introduces conditional branches, and thus potentially timing vulnerabilities, much more conservatively than `gcc`. This is favorable in the context of language-based security avoiding timing vulnerabilities, but seemingly, `clang` is not perfect in preventing timing vulnerabilities even though our results were not able to confirm this. The lack of confirmed results might be ascribed to the simplicity of the generated programs or the sample size. As no timing vulnerabilities were found using `clang`, all following subsections will present data obtained only through `gcc`.

4.2 General Optimizations

We ran the OptiFuzz pipeline with the optimization flags `O0`, `O1`, `O2`, `O3` and `Os`. For each optimization flag, we generated 100,000 programs, which were then fuzzed and analyzed. The ASTs of the generated programs were restricted to have a max depth of 5. Each program was fuzzed with 10,000 different inputs. The results are shown in Table 1 and visualized results for selected data points can be seen in Appendix A.

	`O0`	`O1`	`O2`	`O3`	`Os`
Flagged programs	19.76%	0.23%	1.00%	1.00%	0.37%
H_0 rejected programs (out of total)	12.63%	0.15%	0.62%	0.64%	0.20%
H_0 rejected programs (out of flagged)	63.91%	63.64%	62.09%	64.06%	54.37%

Table 1: Results for the optimization flags `O0`, `O1`, `O2`, `O3` and `Os` using `gcc`. The results are based on 100,000 generated programs for each optimization flag. Each program is generated from an AST of max depth 5. The first row shows the percentage of programs that contained conditional branching instructions after compilation (potential timing leak). The second and third rows show the percentage of programs that resulted in H_0 getting rejected in Welch's t-test (definite timing leak).

Our results show that the problem is very prevalent in `gcc` with `O0`, surprisingly, being the worst offender where over a tenth of the randomly generated programs were rejected by Welch's t-test. The fact that `O0` introduces timing vulnerabilities with such a high probability is concerning since `O0` has the most configurable optimizations turned off [GNU, 2022]. This points to the fact that it is not easy for the developer to mitigate problematic optimizations in `gcc`.

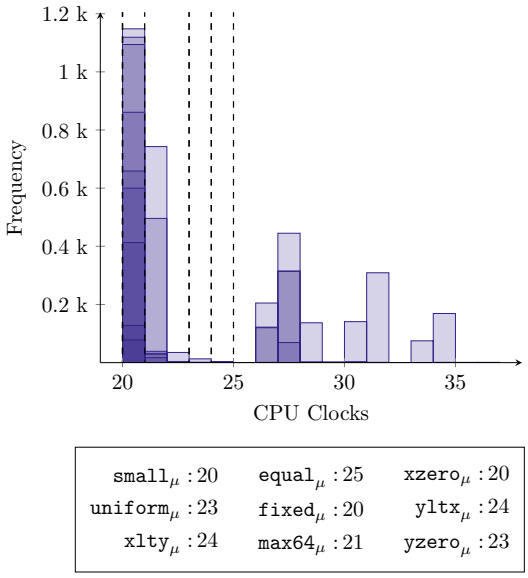
Our experiment revealed some limitations to our approach to finding timing vulnerabilities. False positives, i.e. constant-time code that was rejected by Welch's t-test, were present in the dataset from the `O0` experiment. This is likely due to noise in the measurements. Experimentally, we found that Welch's t-test will reject around 4.4% of programs that are constant-time due to noise. Since such a high number of programs compiled with `O0` contained branching instructions, this noise was enough to cause a significant amount of false positives compared to the number of programs generated for the experiment. It is also possible that seemingly constant-time programs are not constant-time due to hardware optimizations, like branch prediction or speculative execution, which are not taken into account in our model and are out-of-scope for this paper. However, our results also show that definite timing leaks are present in the `O0` dataset as seen in Figure 11.

Findings from the other datasets also deserve mentioning. The `O1`, `O2`, `O3` and `Os` datasets had several false negatives, i.e. programs that were not rejected by Welch's t-test, but were variable-time. Concretely, we saw examples where the assembled code branched on a *very* specific scenario, e.g. if $x = 8129346172934691$, which is near-impossible to detect by fuzzing since the probability of hitting that specific value is very low. We did not find any false positives in any other dataset than `O0`. This is because the vast majority of programs in the `O1`, `O2`, `O3` and `Os` datasets were variable-time. This points to the fact that more aggressive optimizations that introduce conditional branching are more likely to produce actual variable-time machine code. The reason that Welch's t-test did not detect vulnerabilities in all programs in the datasets is likely due to the above-mentioned false negatives, but also because the t-test has some inherent limitations as mentioned in Section 3.4. Concrete examples of true and false positives and negatives can be found in Appendix A.

Finally, we want to address the fact that the general optimizations datasets were generated with programs that had ASTs of max depth 5. This was done to reduce the length of the programs to make them more manageable for analysis. However, we also generated datasets with ASTs of max depth 12 and found that the results were similar in the `O0` and `O1` datasets, while the `O2`, `O3` and `Os` datasets had significantly more programs with conditional branching – around 5%, 5% and 2% respectively. We deem the choice of max depth 5 to be a good compromise between the length of the programs and the number of conditional branches since our results underline the problem we are trying to address.

4.3 Specific Optimizations

The results in Table 1 show that the percentage of vulnerable programs is the highest for the `O0` dataset, then it decreases significantly for the `O1` dataset, and then it increases significantly for the `O2` and `O3` datasets. The significant decrease from `O0` to `O1` points to the fact that some optimizations in `gcc` decrease the number of



(a) Fuzzing results visualized. The means of each fuzz class are shown as vertical lines.

```
1  ...
2    pushq %rbp
3    movq  %rsp, %rbp
4    movq  %rdi, -8(%rbp)
5    movq  %rsi, -16(%rbp)
6    cmpq  $0, -8(%rbp)
7    js  .L2
8    movl  $-29, %eax
9    movl  $-37, %edx
10   movl  %eax, %ecx
11   sarl  %cl, %edx
12   movl  %edx, %eax
13   cmpl  $1, %eax
14   jle .L3
15 .L2:
16   movl  $1, %eax
17   jmp .L5
18 .L3:
19   movl  $0, %eax
20 .L5:
21   popq  %rbp
22   ret
23 ...
```

(b) Fuzzed assembly code.

Figure 11: An example of a true positive, i.e. a variable-time program that was rejected by Welch's t-test, from the `O0` dataset. The source code of the program is shown in Appendix A. Notice that inputs from the input classes `small`, `xzero`, `fixed` and `max64` are significantly faster than the rest. That is because the program branches on whether x is signed or not (`js` instruction in line 7), which it never is in these 4 classes.

conditional branches. Despite running several hundred experiments testing each possible configurable optimization flag [GNU, 2022] individually, we were not able to identify which optimizations were responsible for this decrease. Furthermore, applying all configurable optimizations from `O1` to `O0` at once did not result in a decrease in the number of conditional branches. This points to the fact that the responsible optimizations are non-configurable, i.e. they are always enabled for `O1`. This also underlines the fact that all the `O*` flags enable optimizations that are non-configurable [GNU, 2022], which in turn gives the developer less control over disabling problematic optimizations to increase the security of the code.

The significant increase from `O1` to `O2` and `O3` points to the fact that some optimizations in `gcc` can introduce conditional branches and therefore timing vulnerabilities. We identified a configurable compiler optimization that was responsible for introducing conditional branches in our datasets: `ftree-pre`. This optimization performs partial redundancy elimination [GNU, 2022]. `ftree-pre` is enabled by default for `O2`, `O3`, and `Os`, but not for `O1`.

Partial redundancy elimination (PRE) is a form of common subexpression elimination (CSE) that tries to eliminate redundant computations. We discussed how CSE works and how it can introduce conditional branches in Section 2.2.1. Specifically, PRE tries to eliminate computations that are redundant in some, but not all, control flow paths. This is done by moving the computation to a point in the program where it is guaranteed to be executed only once. This results in different control flow paths with different amounts of instructions, which in turn result in timing vulnerabilities.

4.4 Vulnerable Expressions

Finally, we identified various types of expressions that were compiled into conditional branches. We identified expressions on the form c `op` x `comp` y, where c is a constant, `op` is a bitwise operator, `comp` is a comparison operator, and x and y are variables. Expressions of this form result in constant-time code containing conditional branches, much like the example in Figure 7.

A more severe type of expression that we identified was expressions on the form *undefined* `op` (x `comp` y), where *undefined* contains undefined behavior, `op` is a bitwise operator, `comp` is a comparison operator, and x and y are variables. Expressions of this form sometimes result in variable-time code when compiled with `O0`. However, when compiled with `O1`, `O2`, `O3` or `Os`, the entire program is optimized away and replaced with `return 0;`. This is because the compiler can deduce that the expression contains undefined behavior, and, according to the C standard, the compiler is allowed to replace undefined behavior with anything [ISO]. An example of this type of expression and its corresponding assembly, when compiled with `O0`, is shown in Figure 12.

```c
int program(long long int x, long long
    int y) {
  return (42 >> -1) & (x == y);
}
```

(a) An example of a program containing an expression of the form *undefined* `op` (x `comp` y). Specifically, bit-shifting with a negative number is undefined behavior in C [ISO].

```
...
  pushq   %rbp
  movq    %rsp, %rbp
  movq    %rdi, -8(%rbp)
  movq    %rsi, -16(%rbp)
  movq    -8(%rbp), %rax
  cmpq    -16(%rbp), %rax
  jne     .L2
  movl    $-1, %eax
  movl    $42, %edx
  movl    %eax, %ecx
  sarl    %cl, %edx
  movl    %edx, %eax
  andl    $1, %eax
  jmp     .L4
.L2:
  movl    $0, %eax
.L4:
  popq    %rbp
  ret
...
```

(b) Corresponding assembly code when compiled with `O0` in `gcc`.

Figure 12: An example of a small program that causes `gcc` to generate variable-time code when compiled with `O0`. Specifically, one can obtain whether x is equal to y by timing the program.

Finally, and perhaps most importantly, we identified expressions that will result in variable-time code when compiled with `O2`, `O3` and `Os`. As mentioned in Section 4.3, partial redundancy elimination (PRE) can introduce conditional branches. PRE is enabled for `O2`, `O3` and `Os` [GNU, 2022]. In some expressions of the form e `comp` e', where e and e' contain subexpressions that are not common between e and e', PRE will introduce conditional branches. An example of this is shown in Figure 13.

```c
int program(long long int x, long long int y) {
  return 42 * (x == 42) == (y == 42);
}
```

Figure 13: An example of a program that will be variable-time when compiled with `O2`, `O3` or `Os`, specifically because of PRE. PRE will identify two control flow paths in this expression: One where $x = 42$ and one where $x \neq 42$. If $x = 42$, the expression will always evaluate to 0 since `42 * (42 == 42) == (y == 42)` will evaluate to $42 \stackrel{?}{=} 0$ or $42 \stackrel{?}{=} 1$ which is false in both cases. If $x \neq 42$, the expression depends on whether $y = 42$. Hence `(y == 42)` is redundant in the first control flow path, but not in the second. As a result, PRE will insert a conditional branch that branches on $x = 42$ and only considers y if that is *not* the case.

5 Evaluation

In this section, we evaluate the quality of our results and the OptiFuzz tool. Generally, we found that the problem of timing vulnerabilities in `gcc` is prevalent and, according to OptiFuzz, between 0.15% and 12.63% of random C programs are vulnerable to timing attacks depending on what optimizations are used when compiling. We argue that these results are predominantly representative of the actual real-world problem in the following.

Our results have surfaced real problems in `gcc` since they have led us to pinpoint exact problematic optimizations as well as specific code patterns that are vulnerable to timing attacks (see Section 4.3 and 4.4 respectively). However, some uncertainties are present throughout our pipeline. We would like to address these uncertainties and discuss their impact on our results in the following subsections.

5.1 Code Generation and Inspection

Code generation might seem the most uncertain part of our pipeline since the generated programs are not representative of real-world programs. However, developers tend to write short expressions using arithmetic, bitwise and comparison operators to force their code to be constant-time in the real world, much like the programs generated by OptiFuzz. We have found multiple examples of constant-time code snippets looking dangerously close to the vulnerable expressions we identified in Section 4.4 [Cauligi et al., 2017, Simon et al., 2018, Almeida et al., 2016]. Furthermore, we have identified the use of this paradigm in the wild, for example in x86 AES implementation of the OpenSSL library [OpenSSL, 2023]. Hence quantifying the problem of timing vulnerabilities in the generated programs is indicative of the real-world problem.

The inspection of the generated programs is exact since it finds all problematic instructions in the generated assembly code as argued in Section 3.2. Furthermore, our results show that between 54.37% and 64.06% of

the programs that are flagged by inspection are eventually rejected by the statistical analysis (see Section 3.4), confirming they are variable-time. This means that, even though inspection serves as an overapproximation of the problem, it does not cause too big of an overhead in the OptiFuzz pipeline.

5.2 Fuzzing and Statistical Analysis

Our results show that some noise is present in our fuzzing data. As discussed in Section 4.2, this caused false positives in our results. Concretely, we found that around 4.4% of programs are rejected by Welch's t-test as false positives due to noise when fuzzing. This percentage was found by fuzzing a large number of definite constant-time programs and observing how many of them were rejected by Welch's t-test after fuzzing. This is not alarming since it is well within the chosen significance level of $p = 0.05$ for Welch's t-test.

Furthermore, from manual analysis, we found that there are quite a lot of false negatives present in the data. As argued in Section 4.2, this is due to a large number of programs branching on very specific inputs. This means that the programs are not rejected by Welch's t-test since the branch is not taken often enough to cause a significant difference in the execution time. This is a problem since the programs are still vulnerable to timing attacks. OptiFuzz is still a great tool in this particular case, as our analysis and visualization pipeline aids in the manual discovery of timing leaks.

Additionally, we only applied statistical analysis between the input classes `fixed` and `uniform`. This means that we have missed some vulnerable programs that branch on other input classes. An example of this can be seen in Appendix A (Os False Negative). This observation indicates an opportunity for future work where the statistical analysis could be improved to handle these cases and perhaps more input classes could be considered. But, as seen in the results in Section 4.2, the statistical analysis is still able to reject the majority of vulnerable programs.

In total, the pipeline does not seem to have any major flaws that would cause the results to be unrepresentative of the real-world problem.

6 Solving the Problem

From the previous sections, it is clear that modern compilers introduce timing vulnerabilities in constant-time programs. This is devastating for security since it means that developers cannot reliably write constant-time code. A solution is required. In this section, we discuss how the problem can be solved and what the implications different solutions have on security and efficiency.

As shown in Section 4.1, `clang` is much more conservative when it comes to inserting conditional branches in constant-time programs. Instead of inserting variable-time conditional branches, `clang` uses constant-time conditional moves to implement branching control flow. From this observation, it might be tempting to conclude that always substituting conditional branches with conditional moves is the solution to the problem. However, there is more to the story.

`cmov` instructions severely restrict the out-of-order engine on modern CPUs since `cmov` instructions increase the data dependency between instructions significantly [Intel, 2023a]. Furthermore, branch predictions are not used for `cmov` instructions, which can lead to significant performance degradation. Hence, we have to address this trade-off between security and efficiency.

6.1 Forcing `cmov` Instructions When Necessary

This trade-off between security and efficiency has led multiple researchers to propose solutions that only force `cmov` instructions to be used on critical data [Simon et al., 2018, Sprenkels, 2019]. In both studies, the authors propose to add the possibility to specify forced `cmov` instructions in the source code and restrict the backend of the compiler to not being able to optimize the `cmov` away for those specific instructions. This solution has the advantage that it does not affect the performance of non-critical parts of the program. In fact, Simon et al. show that their solution only introduces a performance penalty of 1% and sometimes even increases performance.

However, this solution also has its disadvantages. First, the solution relies heavily on the developer. The developer has to know which parts of the program are critical, and also be able to use the language extension correctly. In turn, it means that security relies on human factors, which are error-prone. Second, the solution relies on the compiler being configurable to avoid forced `cmov` instructions being optimized away. As we discussed in Section 4.1 and 4.2, `clang` seems to be quite configurable while `gcc` does not. Hence, the solution might not be easily applicable to all compilers.

Another solution is to implement vulnerable parts of the program using a domain-specific language that is designed to be constant-time. Cauligi et al. proposed a domain-specific language, called FaCT, for implementing constant-time cryptographic algorithms [Cauligi et al., 2017]. FaCT is designed to be interoperable with C, and the compiler ensures constant-time code by leveraging the `ct-verif` tool that is based on verification of the LLVM IR Almeida et al. [2016]. `ct-verif` operates on a formalization of constant-time code and verifies based on a theoretically sound and complete methodology. However, Almeida et al. note that the verification process of `ct-verif` is not guaranteed to identify all timing vulnerabilities due to translation between the LLVM IR

and machine code, and due to possible incompleteness of their formalization of constant-time code. Some of these issues were experimentally verified. However, generally `ct-verif` captures significantly more timing vulnerabilities than other verification alternatives [Almeida et al., 2016]. Additionally, `ct-verif` is efficient at verifying constant-time code, often being much faster than the compilation step, but occasionally being a few times slower.

6.2 Bullet-proofing the Compiler

A more radical approach is to force the compiler to always produce constant-time code. Barthe et al. provides a formally verified constant-time C compiler, built on top of the CompCert verified C compiler [Barthe et al., 2019]. The constant-time CompCert compiler guarantees that constant-time source code is compiled into constant-time machine code. This is achieved by instrumenting the operational semantics of CompCert's IR to be able to capture cryptographic constant-time and proving that cryptographic constant-time is preserved during compilation. This has immense advantages for security. However, the security advantages come at the cost of efficiency. The constant-time CompCert compiler is significantly slower than `gcc`. Experimental results show an efficiency penalty of well over 100% in some cases [Barthe et al., 2019].

6.3 Testing Solutions

The final category of solutions we identified is black-box testing solutions like DudeCT [Reparaz et al., 2017] and our tool OptiFuzz. Concretely, one can test specific implementations for timing vulnerabilities using these tools. These tests can even be automated to run as part of a continuous integration pipeline. Solutions in this category have the obvious advantage that they are simple and require no modeling of hardware behavior or language semantics. It is arguably the most practical solution since it does not require any changes to the compiler or the language. It also does not induce any performance penalty.

The testing solutions, however, come with a significant security disadvantage in that they are noisy and not exact. As we showed in Section 4.2, OptiFuzz will produce false positives and false negatives. Reparaz et al. raised similar concerns for DudeCT. Fuzzing tools provide more of an indication of the presence of timing vulnerabilities than a guarantee.

Finally, we want to mention the possibility to combine the solutions. OptiFuzz differs from other existing solutions in that it operates on the tool level instead of the program level. For example, we have used OptiFuzz to conclude that `gcc` is not suitable for compiling constant-time code, while `clang` is safer. Hence OptiFuzz might be tailored to be used in combination with the solutions mentioned in Section 6.1 to detect security vulnerabilities in these. This would allow tool developers to maximize the security of their tools while keeping a low performance penalty.

In summary, various solutions exist to the problem of timing vulnerabilities in constant-time programs. The solutions range from solutions that are very efficient but not very secure to solutions that are very secure but not very efficient. The correct solution depends on the use case. But importantly, many developers do not make use of these tools even though they are aware of the problem [Jancar et al., 2022]. Hence, we urge developers to start using these tools since, as we showed, relying on the compiler to produce constant-time code is not a viable solution.

7 Conclusion and Future Work

We have provided a tool, OptiFuzz, that can be used to test compilers for timing vulnerabilities. OptiFuzz has been optimized considering the C language specification, the x86-64 architecture, and state-of-the-art statistical analysis. This has resulted in a simple tool that is indicative of the real-world issue of timing vulnerabilities introduced by compilers. With OptiFuzz, we have shown that between 0.15% and 12.63% of randomly generated programs are vulnerable to timing attacks when compiled with `gcc` using various optimization flags. We have used our results to pinpoint specific optimizations within `gcc` that are responsible for introducing timing vulnerabilities, as well as specific expression patterns that are vulnerable to resulting in timing vulnerabilities when compiled. We have also shown that due to better utilization of `cmovcc` instructions, `clang` introduces significantly fewer timing vulnerabilities – so much that we could not detect any timing vulnerabilities introduced by `clang` using our methodology. In light of this result, we have discussed how the issue of timing vulnerabilities introduced by compilers can be mitigated by using language-based security techniques from the literature. Generally, solutions to this problem represent a trade-off between security and performance where very secure solutions are not that performant and vice versa.

In the light of our research, we have identified several areas for future work: Extend OptiFuzz to generate more complicated programs, eg. by reintroducing the division and modulus operators while dealing with division by zero in a meaningful way. Implement other relevant input classes. Perhaps a dynamic class containing the hardcoded values from a program. This might be able to catch the very specific but unpredictable values that cause branching. Extend the statistical analysis and consider results from all input classes. This could include researching the usability of other t-tests. Create a tool that warns developers in real-time about constant C expressions that might be compiled to variable-time by comparing it to a database precomputed using OptiFuzz. This tool could furthermore be implemented in a CI/CD environment. Extend OptiFuzz to support other processor architectures, especially ones used by embedded systems.

References

A. Abel and J. Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304062. URL http://doi.acm.org/10.1145/3297858.3304062.

J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 53–70, USA, 2016. USENIX Association. ISBN 9781931971324.

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.

J. Chen and J. Revels. Robust benchmarking in noisy environments, 2016. URL https://arxiv.org/abs/1608.04295.

B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE Symposium on Security and Privacy*, pages 45–60, 2009. doi: 10.1109/SP.2009.19.

J.-S. Coron, D. Naccache, and P. Kocher. Statistics and secret leakage. *ACM Transactions on Embedded Computing Systems*, 3(3):492–508, aug 2004. doi: 10.1145/1015047.1015050. URL https://doi.org/10.1145%2F1015047.1015050.

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.

F. Durvaux and F.-X. Standaert. From improved leakage detection to the detection of points of interests in leakage traces. Cryptology ePrint Archive, Paper 2015/536, 2015. URL https://eprint.iacr.org/2015/536. https://eprint.iacr.org/2015/536.

GNU. *GCC 12.2 Manual: Options That Control Optimization*, 2022. URL https://gcc.gnu.org/onlinedocs/gcc-12.2.0/gcc/Optimize-Options.html#Optimize-Options.

G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi. A testing methodology for side channel resistance. 2011.

Intel. *Pentium Pro Family Developer's Manual*, 1995. URL https://stuff.mit.edu/afs/sipb/contrib/doc/specs/ic/cpu/x86/pentium-pro/vol2.pdf.

Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2023a. URL https://cdrdv2.intel.com/v1/dl/getContent/671488?explicitVersion=true.

Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2023b. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. a. (Vol. 3A, Ch. 9.3, pp. 17-18), b. (Vol. 3B, Ch. 18.17.1, p. 43), c. (Vol. 3B, Ch. 18.17.4, p. 44).

ISO. Programming languages - C ISO/IEC 9899:2018. Standard, International Organization for Standardization, jun 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.

LLVM. *LLVM 17.0.0 documentation: X86CmovConversion.cpp File Reference*, 2023. URL https://llvm.org/doxygen/X86CmovConversion_8cpp.html.

OpenSSL. Openssl github repository, 2023. URL https://github.com/openssl/openssl.

G. Paoloni. How to Benchmark Code Execution Times on Intel® IA-32 and IA-64 Instruction Set Architectures. Technical report, Intel, 10 2010. URL https://www.intel.de/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

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.

A. Sprenkels. Llvm provides no side-channel resistance, 2019. URL `https://electricdusk.com/cmov-conversion.html`.

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`.

B. L. Welch. THE GENERALIZATION OF 'STUDENT's' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED. *Biometrika*, 34(1-2):28–35, 1947. doi: 10.1093/biomet/34.1-2.28. URL `https://doi.org/10.1093/biomet/34.1-2.28`.

Appendices

Appendix Reading Guide

The following is a guide on how to interpret the results produced by OptiFuzz.

At the top of a page, a program signature line will be shown:

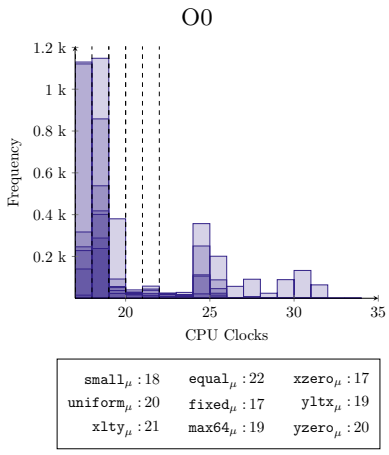
Program 1 – `Seed 257634744` – `gcc O0 O1 O2 O3`
`Classes: uniform fixed xzero yzero xlty yltx small equal max64`

The seed used to generate the program is printed, followed by the compiler and compiler flags. On a new line, all input classes that the fuzzer used are printed. Refer to 3.3 for a description of each input class.

Next, the program is shown:

```
...
int program(long long int x, long long int y) { return (((44 ^ ((false >> (-874552120269441001
    == x)) | ((53 * 29) >= (-5509653365268467218 ^ x)))) * 24) & (false + x)); }
```

The first two lines of the program have been trimmed. Namely, two define statements which define `false` to 0 and `true` to 1. What follows now is a histogram of the measured times:



The histogram caption at the top indicates which compiler flag was used to compile the program. The table below the histogram indicates the means for the different input classes. These means have been drawn as dashed lines over the histogram.

The histogram plot is not conventional, as the saturation indicates agreement across all input classes. That is, if a bucket is close to transparent, then that might indicate, that it only came from one input class. If multiple input classes have bins at the same location with the same height, the bin will be very saturated. If we examine the histogram above, we see that some input classes got about a total of about 1000 measurements in the [25, 32] interval. As no mean sticks out - other than the `equal` class - it might be disregarded as noise.

Next is the assembly, hypothesis rejection flag, and jump indicators, which should be directly below the histogram.

H_0 REJECTED! jg [10]

```
1  ...
2    endbr64
3    pushq %rbp
4    movq  %rsp, %rbp
5    movq  %rdi, -8(%rbp)
6    movq  %rsi, -16(%rbp)
7    movabsq $-5509653365268467218, %rax
8    xorq  -8(%rbp), %rax
9    cmpq  $1537, %rax
10   jg  .L2
11   movl  $1080, %edx
12   jmp .L3
13 .L2:
14   movl  $1056, %edx
15 .L3:
16   movq  -8(%rbp), %rax
17   andl  %edx, %eax
18   popq  %rbp
19   ret
20 ...
```

Only if the hypothesis was rejected, it will be marked with red.

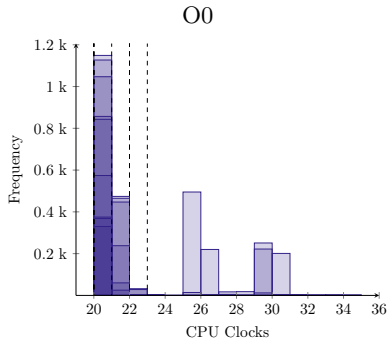
The jump indicator `jg [10]` tells us, that a `jg` instruction was found on line 10.

A Select General Optimizations Results

O0 False Positive – Seed 1000716339 – gcc O0

Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return (~false & ((((~25 >> (y >= x)) - (~true
    - -(29))) * 59) - true)); }
```



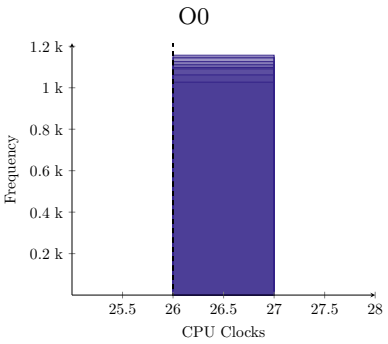
$small_{\mu}$: 22	$equal_{\mu}$: 20	$xzero_{\mu}$: 20
$uniform_{\mu}$: 21	$fixed_{\mu}$: 20	$yltx_{\mu}$: 22
$xlty_{\mu}$: 20	$max64_{\mu}$: 22	$yzero_{\mu}$: 23

H_0 REJECTED! jl [8]

```
...
  pushq %rbp
  movq  %rsp, %rbp
  movq  %rdi, -8(%rbp)
  movq  %rsi, -16(%rbp)
  movq  -16(%rbp), %rax
  cmpq  -8(%rbp), %rax
  jl   .L2
  movl  $-2361, %eax
  jmp .L4
.L2:
  movl  $-3128, %eax
.L4:
  popq  %rbp
  ret
...
```

O0 True Negative – Seed 100055965 – gcc O0
Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return (((-5831744386033941598 - (61 != y)) *
    ((((17 != 2147556796248018429) != (x * 23)) <= (-5674359421836342679 << (y == x))) > (33
    <= y))) - -((y != ~((true != y) * (true * -1327665118481002523))))); }
```



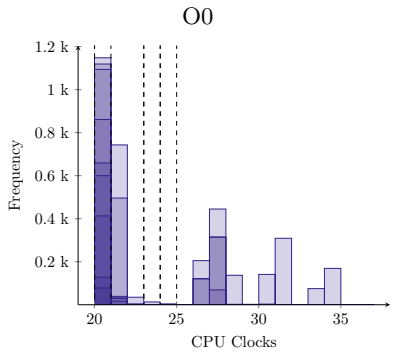
$\texttt{small}_{\mu}:26$	$\texttt{equal}_{\mu}:26$	$\texttt{xzero}_{\mu}:26$
$\texttt{uniform}_{\mu}:26$	$\texttt{fixed}_{\mu}:26$	$\texttt{yltx}_{\mu}:26$
$\texttt{xlty}_{\mu}:26$	$\texttt{max64}_{\mu}:26$	$\texttt{yzero}_{\mu}:26$

je [7, 26, 7, 26]

```
...
  pushq %rbp
  movq  %rsp, %rbp
  movq  %rdi, -8(%rbp)
  movq  %rsi, -16(%rbp)
  cmpq  $61, -16(%rbp)
  je  .L2
  movl  $-107904095, %ecx
  jmp .L3
.L2:
  movl  $-107904094, %ecx
.L3:
  movq  -16(%rbp), %rax
  cmpq  -8(%rbp), %rax
  sete  %al
  movzbl  %al, %edx
  cmpq  $32, -16(%rbp)
  setg  %al
  movzbl  %al, %eax
  cmpl  %eax, %edx
  setg  %al
  movzbl  %al, %eax
  imull %eax, %ecx
  movl  %ecx, %edx
  cmpq  $1, -16(%rbp)
  je  .L4
  movabsq $1327665118481002522, %rax
  jmp .L5
.L4:
  movq  $-1, %rax
.L5:
  cmpq  -16(%rbp), %rax
  setne %al
  movzbl  %al, %eax
  addl  %edx, %eax
  popq  %rbp
  ret
...
```

O0 True Positive – Seed 1005231680 – gcc O0
Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return (((x >= (((8970144495437897379 != x) <<
    (false <= 32)) < ((60 == 0) > -(false)))) <= (((-(37) >> -(29)) <= ((-6646567428393237337
    | 51) <= (1735463287021949172 >= 28))) != (26 != false))) ^ false); }
```



$small_{\mu}$: 20	$equal_{\mu}$: 25	$xzero_{\mu}$: 20
$uniform_{\mu}$: 23	$fixed_{\mu}$: 20	$yltx_{\mu}$: 24
$xlty_{\mu}$: 24	$max64_{\mu}$: 21	$yzero_{\mu}$: 23

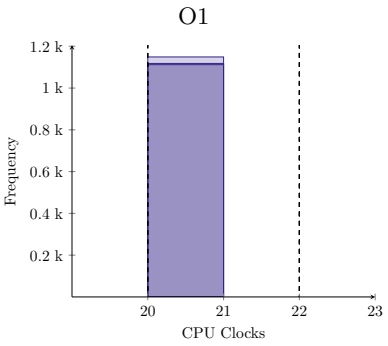
H_0 REJECTED! js [7], jle [14]

```
...
  pushq %rbp
  movq  %rsp, %rbp
  movq  %rdi, -8(%rbp)
  movq  %rsi, -16(%rbp)
  cmpq  $0, -8(%rbp)
  js  .L2
  movl  $-29, %eax
  movl  $-37, %edx
  movl  %eax, %ecx
  sarl  %cl, %edx
  movl  %edx, %eax
  cmpl  $1, %eax
  jle .L3
.L2:
  movl  $1, %eax
  jmp .L5
.L3:
  movl  $0, %eax
.L5:
  popq  %rbp
  ret
...
```

O1 False Negative – Seed 1015904072 – gcc O1

Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return ((!((((x == -4923030354367341 57) ^ 30) >= ((x != 12) | (1803381109880440640 * x)))) != ((7 | y) << (((x >= x) | (49 - 553330948696234685)) * 26))) * (29 & 25)); }
```



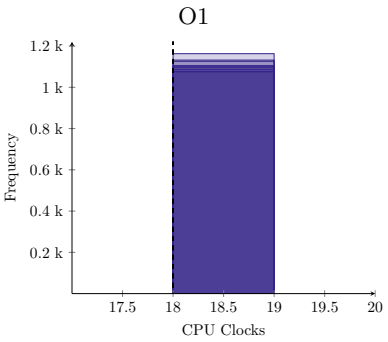
$small_{\mu}$: 20	$equal_{\mu}$: 20	$xzero_{\mu}$: 20
$uniform_{\mu}$: 22	$fixed_{\mu}$: 22	$yltx_{\mu}$: 22
$xlty_{\mu}$: 22	$max64_{\mu}$: 22	$yzero_{\mu}$: 22

je [5]

```
...
  movl  $0, %eax
  movabsq $-4923030354367341 57, %rdx
  cmpq  %rdx, %rdi
  je  .L1
  cmpq  $12, %rdi
  setne %al
  movzbl  %al, %eax
  movabsq $1803381109880440640, %rdx
  imulq %rdx, %rdi
  orq %rdi, %rax
  cmpq  $31, %rax
  movl  $0, %eax
  movl  $25, %edx
  cmovge  %edx, %eax
.L1:
  ret
...
```

O1 True Negative – Seed 567981886 – gcc O1
Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return ((!((((x >> true) | !(true)) - !(0))) -
    44) ^ ~-((x != (false << (1883797080741276612 != 62))))); }
```



$\texttt{small}_{\mu}:18$	$\texttt{equal}_{\mu}:18$	$\texttt{xzero}_{\mu}:18$
$\texttt{uniform}_{\mu}:18$	$\texttt{fixed}_{\mu}:18$	$\texttt{yltx}_{\mu}:18$
$\texttt{xlty}_{\mu}:18$	$\texttt{max64}_{\mu}:18$	$\texttt{yzero}_{\mu}:18$

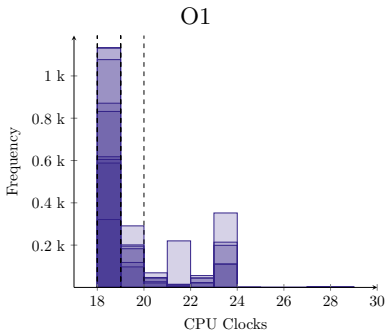
je [5]

```
...
  movq  %rdi, %rax
  sarq  %rax
  cmpq  $1, %rax
  je  .L3
  cmpq  $1, %rdi
  sbbl  %eax, %eax
  movl  $-44, %edx
.L2:
  xorl  %edx, %eax
  ret
.L3:
  movl  $-43, %edx
  movl  $0, %eax
  jmp .L2
...
```

O1 True Positive – Seed 1016239267 – gcc O1

Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return (!((-(((5112480584718607435 - y) <= (58
    == 4826579118379168524))) <= (~!(x) << ((x >= false) * (18 ^ -37105162677356427 99))))) -
    (~((false <= (false >> 5573564991473225987)) * ((-6722922963376055943 <
    -5233458742647055439) & -7328655989583803584)) << (-((-(6600517005617554449) <= (x <<
    8493232900771133046))) >= 68763027978568547 57))); }
```



$\texttt{small}_{\mu}:18$	$\texttt{equal}_{\mu}:19$	$\texttt{xzero}_{\mu}:18$
$\texttt{uniform}_{\mu}:19$	$\texttt{fixed}_{\mu}:18$	$\texttt{yltx}_{\mu}:19$
$\texttt{xlty}_{\mu}:20$	$\texttt{max64}_{\mu}:19$	$\texttt{yzero}_{\mu}:19$

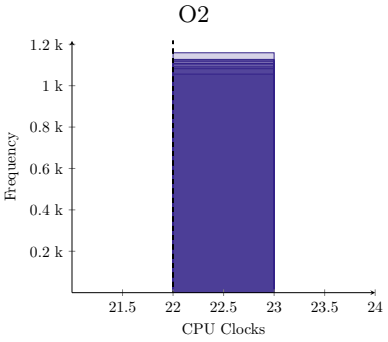
H_0 REJECTED! js [4]

```
...
  movl  $1, %eax
  testq %rdi, %rdi
  js  .L5
.L1:
  ret
.L5:
  movabsq $5112480584718607435, %rax
  cmpq  %rax, %rsi
  setl  %al
  movzbl  %al, %eax
  addl  $1, %eax
  jmp .L1
...
```

O2 False Negative – Seed 1007441663 – gcc O2

Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return ((((((y != y) < ~true) == (35 - (y <=
    true))) < (((47 + y) != true) != ((y ^ y) >> (true <= y)))) << ((((y >> true) < (y ^ 27))
    == (-(y) ^ (41 - 59))) <= ~((y <= 22) - (y >= y)))) * x); }
```



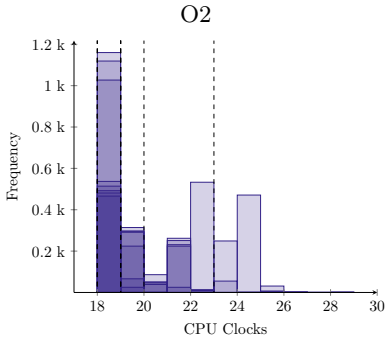
$small_{\mu}$: 22	$equal_{\mu}$: 22	$xzero_{\mu}$: 22
$uniform_{\mu}$: 22	$fixed_{\mu}$: 22	$yltx_{\mu}$: 22
$xlty_{\mu}$: 22	$max64_{\mu}$: 22	$yzero_{\mu}$: 22

je [4]

```
...
  xorl  %eax, %eax
  cmpq  $-46, %rsi
  je  .L1
  movq  %rsi, %rdx
  movq  %rsi, %rax
  sarq  %rdx
  xorq  $27, %rax
  cmpq  %rax, %rdx
  movq  %rsi, %rax
  setl  %dl
  negq  %rax
  movzbl  %dl, %edx
  xorq  $-18, %rax
  cmpq  %rax, %rdx
  sete  %dl
  xorl  %eax, %eax
  cmpq  $22, %rsi
  setle %al
  movzbl  %dl, %edx
  negl  %eax
  cmpl  %eax, %edx
  movl  %edi, %eax
  setle %cl
  sall  %cl, %eax
.L1:
  ret
...
```

O2 True Positive – Seed 1042311246 – gcc O2
Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return (((((x << (x * true)) <= false) < (((x
    <= false) | (true <= false)) <= ((x + y) < ~false))) < y) <= (((((-7517777934437036811 <=
    y) != (false ^ y)) <= ((48 - false) | (false != false))) == -(-((y >= x)))) * 38)); }
```



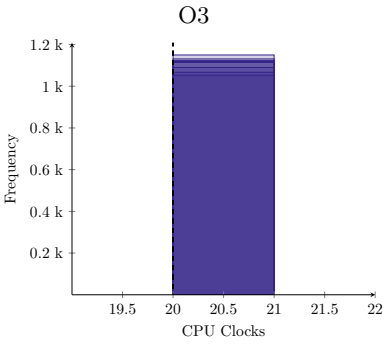
$\texttt{small}_{\mu}:20$	$\texttt{equal}_{\mu}:18$	$\texttt{xzero}_{\mu}:19$
$\texttt{uniform}_{\mu}:19$	$\texttt{fixed}_{\mu}:18$	$\texttt{yltx}_{\mu}:23$
$\texttt{xlty}_{\mu}:18$	$\texttt{max64}_{\mu}:19$	$\texttt{yzero}_{\mu}:19$

H_0 REJECTED! jle [4]

```
...
  movl  $1, %eax
  cmpq  %rsi, %rdi
  jle .L1
  movl  %edi, %ecx
  movq  %rdi, %rax
  salq  %cl, %rax
  leaq  (%rdi,%rsi), %rcx
  testq %rax, %rax
  setle %dl
  testq %rdi, %rdi
  setle %al
  cmpq  $-1, %rcx
  setl  %cl
  cmpb  %al, %cl
  setnb %al
  cmpb  %al, %dl
  setb  %al
  movzbl  %al, %eax
  cmpq  %rsi, %rax
  setge %al
  movzbl  %al, %eax
.L1:
  ret
...
```

O3 False Negative – Seed 1017210177 – gcc O3
Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return ((((((y - 24) ^ (x - y)) <=
    6735857849390429567) <= true) >= ~!(-((x * -109237477041579566)))) *
    ~!((((2695610257113455567 - true) * (-2077761611812603585 == x)) - ((y == true) +
    (1738926233153657090 & true))))); }
```



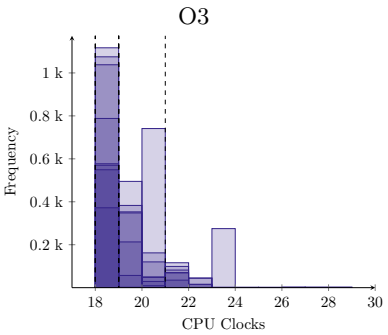
$\texttt{small}_{\mu}:20$	$\texttt{equal}_{\mu}:20$	$\texttt{xzero}_{\mu}:20$
$\texttt{uniform}_{\mu}:20$	$\texttt{fixed}_{\mu}:20$	$\texttt{yltx}_{\mu}:20$
$\texttt{xlty}_{\mu}:20$	$\texttt{max64}_{\mu}:20$	$\texttt{yzero}_{\mu}:20$

je [5]

```
...
  movabsq $-2077761611812603585, %rdx
  movl  $-1, %eax
  cmpq  %rdx, %rdi
  je  .L1
  xorl  %eax, %eax
  cmpq  $1, %rsi
  setne %al
  notl  %eax
.L1:
  ret
...
```

O3 True Positive – Seed 1004183620 – gcc O3
Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return (((false ^ -(((false + 6) | (20 >= x)))
    ) - (~((x > false) ^ ~7) + -(false))) ^ ((x | 5743022236661831965) - (x ^ 41))); }
```



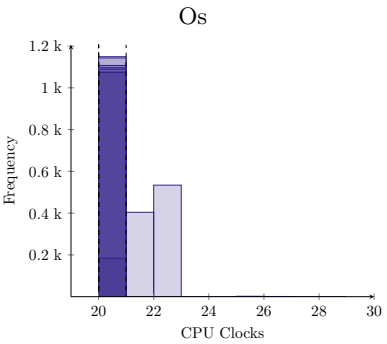
$\texttt{small}_{\mu}: 18$	$\texttt{equal}_{\mu}: 19$	$\texttt{xzero}_{\mu}: 21$
$\texttt{uniform}_{\mu}: 19$	$\texttt{fixed}_{\mu}: 18$	$\texttt{yltx}_{\mu}: 18$
$\texttt{xlty}_{\mu}: 19$	$\texttt{max64}_{\mu}: 19$	$\texttt{yzero}_{\mu}: 19$

H_0 REJECTED! jg [4]

```
...
  movl  $-12, %edx
  cmpq  $20, %rdi
  jg  .L2
  xorl  %edx, %edx
  testq %rdi, %rdi
  setg  %dl
  subl  $14, %edx
.L2:
  movl  %edi, %eax
  xorl  $41, %edi
  orl $-513418979, %eax
  subl  %edi, %eax
  xorl  %edx, %eax
  ret
...
```

Os False Negative – Seed 101484694 – gcc Os
Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return (-7519414859598388463 <= (x << ((((
    false * x) != (y ^ false)) * 6252294692220964647) == (x * x)))); }
```



$\texttt{small}_{\mu}:20$	$\texttt{equal}_{\mu}:20$	$\texttt{xzero}_{\mu}:20$
$\texttt{uniform}_{\mu}:20$	$\texttt{fixed}_{\mu}:20$	$\texttt{yltx}_{\mu}:20$
$\texttt{xlty}_{\mu}:20$	$\texttt{max64}_{\mu}:20$	$\texttt{yzero}_{\mu}:21$

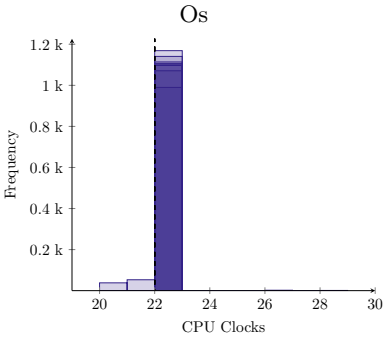
je [3]

```
...
  testq %rsi, %rsi
  je  .L2
  movabsq $6252294692220964647, %rsi
.L2:
  movq  %rdi, %rax
  imulq %rdi, %rax
  cmpq  %rsi, %rax
  movabsq $-7519414859598388463, %rax
  sete  %cl
  salq  %cl, %rdi
  cmpq  %rax, %rdi
  setge %al
  movzbl  %al, %eax
  ret
...
```

Os True Negative – Seed 1052381425 – gcc Os

Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return (((((x == -(y)) > ((5199565972783288834
    - true) * (false != x))) + false) << -((~(y & y) - (54 - (x <= false))))) > -((y == y)));
    }
```



$\texttt{small}_{\mu}:22$	$\texttt{equal}_{\mu}:22$	$\texttt{xzero}_{\mu}:22$
$\texttt{uniform}_{\mu}:22$	$\texttt{fixed}_{\mu}:22$	$\texttt{yltx}_{\mu}:22$
$\texttt{xlty}_{\mu}:22$	$\texttt{max64}_{\mu}:22$	$\texttt{yzero}_{\mu}:22$

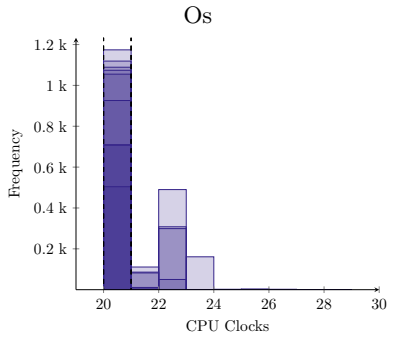
jne [3]

```
...
  testq %rdi, %rdi
  jne .L2
  xorl  %eax, %eax
  testq %rsi, %rsi
  movl  $53, %edx
  sete  %al
  jmp .L3
.L2:
  setg  %dl
  movl  $0, %eax
  movzbl  %dl, %edx
  addl  $53, %edx
.L3:
  leal  1(%rdx,%rsi), %ecx
  sall  %cl, %eax
  notl  %eax
  shrl  $31, %eax
  ret
...
```

Os True Positive – Seed 100426752 – gcc Os

Classes: uniform equal max64 fixed xzero yzero xlty yltx small

```
...
int program(long long int x, long long int y) { return ((61 - !((((x > -2893920677465668034) -
    (42 > 61)) - (22 == (true > true))))) & (((26 - ((false ^ x) * (30 <= true))) - (((true +
    40) & (x > true)) >> (x <= (x - 20)))) & (2438390265689264449 + (22 - ((x + false) > 54))
    ))); }
```



$\texttt{small}_{\mu}$: 20	$\texttt{equal}_{\mu}$: 21	$\texttt{xzero}_{\mu}$: 20
$\texttt{uniform}_{\mu}$: 20	$\texttt{fixed}_{\mu}$: 20	$\texttt{yltx}_{\mu}$: 20
$\texttt{xlty}_{\mu}$: 21	$\texttt{max64}_{\mu}$: 21	$\texttt{yzero}_{\mu}$: 21

H_0 REJECTED! jl [4, 6], jle [6]

```
...
  movabsq $-2893920677465668033, %rax
  cmpq  %rax, %rdi
  jl  .L3
  cmpq  $1, %rdi
  jle .L4
  xorl  %eax, %eax
  cmpq  $55, %rdi
  movl  $25, %edx
  movl  $61, %ecx
  setl  %al
  addl  $170774870, %eax
  jmp .L2
.L3:
  movl  $26, %edx
  movl  $60, %ecx
  jmp .L6
.L4:
  movl  $26, %edx
  movl  $61, %ecx
.L6:
  movl  $170774871, %eax
.L2:
  andl  %ecx, %eax
  andl  %edx, %eax
  ret
...
```