An Investigation of Compiler-Induced Vulnerabilities and Insecure Optimizations

L. Joshua Crotts
Department of Computer Science
University of North Carolina at Greensboro

***Abstract*—Modern compilers use advanced and abstracted techniques to optimize code that aim to further improve performance at the other end of the pipeline. Such optimizations, however, sometimes pose and create obscure threats that may counteract safety measures enforced by the developers of the uncompiled code. While the end goal, from the programmers, is a fast and secure program, an unchecked compiler may violate security implications through unintended consequences. This paper investigates and summarizes compiler-induced vulnerabilities (i.e., vulnerabilities automatically imposed by a compiler in the binary of compiled code), as well as compiler optimizations that either remove or alter data or code safeguards. We also discuss previous research experiments performed to analyze compiler optimization safety, as well as what to explore in the future and lingering open questions.**

***Index Terms*—Compiler optimization, Compiler-induced vulnerability, Compiler security, Code optimization**

I. Introduction

AHO et al. define a compiler as a program that translates code written in a source language to an equivalent target language suited for the operating system and platform architecture [1]. This process involves syntactic and semantic checking of the source code to ensure it solidly follows the standards of its source language. Traditionally, compilers perform the translation in five steps: lexical/token analysis, code parsing (involving syntactic and semantic analysis), intermediate code representation generation, code optimization, and finally target code generation. Compilers often employ code optimization modules to improve performance that may otherwise result in a slower target program. These optimizations range from dead code removal to register allocation algorithms that ultimately reduce to the graph coloring NP-complete problem. Additionally, optimizations of different types which vary in complexity may appear in one or more stages—it is not exclusive to the dedicated "code optimization" stage. In this paper, we will discuss several compiler optimization techniques that have introduced vulnerabilities in the target language that are not present in the source language. Moreover, we will also examine compiler optimizations that outright remove safety checks either due to standards/specifications of the language, or faulty-designed flags. Most importantly, however, we will discuss what we have tried and tested in the research world, what we want to know, as well as what open questions currently exist. Before we begin, we will briefly review the five traditional source code translation steps of a compiler. While the exact number of stages varies depending on the chosen literature (e.g., Honka et al. [2] divides the process into a front-end and back-end) we will use the Aho et al. [1] definition as follows:

1) Lexical analysis is the first step of a compiler, in which a lexer translates input source code into tokens.
2) Syntax analysis, also known as parsing, uses the tokens to build an abstract syntax tree. Parsing also ensures that the input follows the specification in the language grammar.
3) Semantic analysis uses the abstract syntax tree from parsing to check for language-specific issues. For example, type checking in a strongly-typed language.
4) Intermediate code generation, or the creation of an intermediate representation often constructs primitive instructions, sometimes in the form of three-address code. In addition, optimization passes are used to reduce the footprint of the source code and complexity of the final compiler stage. Furthermore, because intermediate representation is usually language and architecture independent, optimization is likely to occur. Consequently, this opens the possibility for insecure optimizations that affect a larger audience due to the independent instruction set.
5) Code generation, also known as target code generation, uses the intermediate representation to create target-dependent code. Sometimes, architecture-dependent optimizations occur here.

II. Background and Prior Work

Compiler optimization history dates back to the mid-1950's, when computer programmers were concerned that compilers for high-level languages could not produce optimal assembly code—an albeit laborious task which was already achievable by programmers themselves. Given the advancement of modern computing power, minuscule optimizations may appear unnecessary. This is counteracted, however, by the reliance placed on compilers nowadays from software developers [2]. After all, software engineer working on non-performant critical code should not need to worry nor care about the implementation at the compiler level, right? In the ideal scenario, it is an abstraction that allows developers to focus on their product and its quality without understanding the transformation from high-level code to a running application on potentially dozens of platforms and architectures. The unsurprising issue, though, is that this reliance comes at the

unforeseen cost of potential security issues. The study and focus of this paper is to analyze previous attempts at compiler correctness observation/proofs and the idea of secure code "mistransformation".

OWASP [3] provides a somewhat incomplete definition of insecure compiler optimization, noting its potential to remove dead store operations. While this is a critical issue in compiler optimization, it severely understates its associated dangers. D'Silva et al. [4] provide a more formal and broader definition which encompasses the correctness of a compiler, reflecting its ability to preserve code behavior from the original source to the intended target. This seemingly intrinsic promise, however, is sometimes violated when it comes to the security of source code.Their work also defines the notion of a correctness-security gap: the delta between a compiler's ability to produce correct code while at the same time ensuring it remains sound with respect to security. In other words, this gap refers to a compiler optimization violating security promises. Compiler optimizations and whether a compiler can create sound and trustworthy code is not a new idea to the research world—programmers and cryptographers alike have fought with compilers (and thus compiler designers) for decades. There is an apparent disconnect between what a developer expects from a compiler, and the optimizations in place by compiler writers. This juxtaposition brings several questions to the table, ranging from whether developers ought to understand and simply mitigate the optimizations from a compiler, or if the compiler should omit optimizations from critical components of code. It has been observed that programmers are fighting with the compiler in the sense that they create temporary workarounds to avoid compiler optimizations. Because compiler writers are always trying to find innovative optimizations, however, programmer solutions are more often than not futile. Further, these workarounds are not bulletproof solutions; if a compiler becomes smart enough in a future version, it may, for example, overwrite workarounds in legacy software, defeating the purpose of the solution. Simon et al. [5] and D'Silva et al. [4] mention some compiler optimization workarounds in the popular OpenSSL[1] program, where cryptographers write inline assembly and create constant-time integer comparison functions. OpenSSL is a heavily-audited program, so it raises an important question about the legitimacy and effectiveness of security implementations in other, less rigorously vetted source code.

Venkatesh et al. [6] evaluated six open-source cryptography libraries to measure their attempts to cope with compiler optimizations. Their report discusses that several techniques are used, with some relying on the platform (i.e., libraries that interface with a particular operating system), and thus require developer intervention to be used properly. They note that this solves a symptom of the greater problem without actually fixing the root cause, and further investigation and communication amongst developers and compiler writers is necessary.

While, of course, it is easy for a developer to turn off all optimizations of a compiler, there are two significant issues with this approach: the first being that the resulting performance overhead is often too great and beyond what is acceptable to the developer. The second is that, in certain unsafe languages e.g., C, some optimizations are not switchable. Several ideas have been proposed to offset the race and decades-long war between compiler writers and programmers. Both Venkatesh et al. [6] and D'Silva et al. [4] propose solutions to use annotations or keywords which designate to the compiler that some code should not be optimized. [4] suggests two keywords where one signifies that data should not be modified in memory, and the other tells the compiler that code branch timing is significant and an optimization may compromise its security. An analogous idea through the `volatile` keyword in C. `volatile` informs the compiler that the state of a variable marked `volatile` is subject to arbitrary change. Unfortunately, the problem with existing solutions such as this is that developers often misunderstand their usage or use it incorrectly.

Moreover, Simon et al. [5] mention the side effects of target code, and how compilers affect these side effects. Not to be directly confused with side effects of, say, a subroutine or function, in this context a side effect is an outside property affected by the target code. For instance, power drain, execution time, and others are side effects. Common compiler optimizations such as subexpression elimination, dead-code removal, constant folding, function inlining, and more all influence, alter, and amplify side effects. As we will further discuss in section 3, cryptographers must have unpredictable code to prevent an adversary from guessing secret passwords or keys as a result of measuring when and what code is executed.

Honka et al. [2] performed a study on the GCC, Clang, and Microsoft Visual C/C++ compilers with particular emphasis on the flag options available to developers. They note that it is confusing to understand what each optimization level does due to the simple naming scheme e.g., `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`. Further, even if developers use the default compiler flags, it enables a surprising number of optimizations using `-O0`. Their data likewise mentions that the `-O1` optimization level contains some uncontrollable and immutable optimizations, thus leading to confusion when the option is utilized. They discovered this problem via compiling source code and comparing the results: one used the `-O1` flag, while the other manually toggled the flags that `-O1` is supposed to enable.

Returning to the paper by D'Silva et al. [4], they evaluate this problem from a formal proof perspective, where they conclude that current compiler correctness proof techniques are insufficient for validating security properties of compiler optimizations because the models fail to account for their implications.

III. Discussion and Experimentation

A. Compiler-Induced Vulnerabilities

In this section, we will discuss how compilers can introduce or inject vulnerabilities into target code via optimizations, rather than focusing on how they remove code that is considered safe.

[1] https://www.openssl.org/

1) Undefined Behavior: Undefined behavior of a language is when source code deviates from the standards of said language, and the behavior/actions of the code turn unpredictable. Undefined behavior generally has no set action, and the output behavior depends on the compiler. One of the reasons C is considered dangerous is because of its undefined behavior mechanics, and the consequences of code that has undefined behavior. For instance, dividing by zero, signed integer overflow, buffer overflow, and pointer arithmetic overflow are all examples of actions that, according to the language, are undefined and the results cannot be accurately (and guaranteed) predicted [2]. Whereas other languages like Java throw exceptions and halt program execution, rarely will C programs terminate without outside influence with the exception of segmentation faults for egregious memory access violations. As we previously stated, because the language does not define what actions to take in these scenarios, compilers do whatever it best sees fit, meaning a compiler may rewrite code sections that it proves are either useless or unnecessary.

```c
int main(int argc, char *argv[]) {
    int buf_size = atoi(argv[1]);

    if (buf_size + 5 > 0x7fffffff) {
      printf("err overflow\n");
      exit(EXIT_FAILURE);
    }

    if (buf_size < 0) {
      printf("buf_size less than 0\n");
      exit(EXIT_SUCCESS);
    }

    printf("%d\n", buf_size);
    return 0;
}
```

Listing 1. Signed integer overflow in C

Listing 1 shows an example of signed integer overflow in action. The program reads and converts a string to an integer on line 2 from the terminal arguments. Then, just for experimentation, we add five to this value and compare it to $2^{32} - 1$: the largest value for a signed 32-bit integer. Because signed integer overflow is, as we mentioned, undefined in the C language standard, the compiler optimizes the first if statement away in its entirety, meaning this check never occurs. The second if statement, on the other hand, executes "correctly" even with signed integer overflow. Listing 2 shows the assembly output of this small program. Because this check never occurs, the programmer or user of the program will not know if they overflow the maximum space for the buffer.

```
main:
.LFB41:
  .cfi_startproc
  subq  $8, %rsp
  .cfi_def_cfa_offset 16
  movq  8(%rsi), %rdi
  movl  $10, %edx
  movl  $0, %esi
  call  strtol@PLT
  testl %eax, %eax
  js  .L4
  movl  %eax, %edx
  leaq  .LC1(%rip), %rsi
  movl  $1, %edi
  movl  $0, %eax
  call  __printf_chk@PLT
  movl  $0, %eax
  addq  $8, %rsp
  .cfi_remember_state
  .cfi_def_cfa_offset 8
  ret
```

Listing 2. Assembly output of signed integer overflow

2) Cryptographic Branch Prediction Obfuscation Elimination: A branch in computer programming is a segment of code where a boolean decision affects how the program continues, or the path where a program proceeds. Suppose we have a program that uses secret values, or even a single secret value/bit. If the source code is not careful, or the compiler is unaware of the associated potential security risks, then it may produce target code vulnerable timing side-channel attack. Cryptographic code relies on unpredictability; if an adversary can predict with reasonable probability the outcome of a branch (and thus the secret values), then such a branch is not cryptographically secure. Simon et al. [5] describe the idea of constant-time selection, meaning that a branch chosen by the program is indistinguishable to an adversary and cannot be reliably predicted. Another way to view this is that the execution time cannot guarantee or reliably prove that one branch was taken over another. The issue with compilers, however, is that they may view a branch or a segment of code as unnecessary, and optimize it away. This causes problems when cryptographers want to guarantee all paths in a program influence side effects in the same way to produce indistinguishable and unpredictable results. Their report also describes an experiment with several versions of the Clang compiler and different variations of a boolean integer selection statement. They discovered that, as Clang saw more and more updates and upgrades, the more that programs in their test suite turned susceptible to side-channel timing attacks. One speculation for this is that the compiler writers working on Clang found new optimizations that are beneficial in the general case, but in certain security-sensitive instances, it breaks legacy and preexisting code.

Another optimization that influences compiler side effects is known as common subexpression elimination. Listing 3 is a example of code that is compromised by an optimization which becomes vulnerable to a side-channel attack.

```c
int main(int argc, char *argv[]) {
    int v = atoi(argv[1]); // Convert argv[1] to
    integer.
    int res = 0;
    if (v == 17) {
       // 17 * 34 is a common subexpression.
       v += 17 * 34;
       v += 17 * 34;
       v += 17 * 34;
    } else {
       res += v * 6;
       res += v * 7;
       res += v << 1 & 0xffff;
    }

    printf("%d\n%d\n", v, res);
    return 0;
}
```

Listing 3. Side-channel attack vulnerability via common subexpression elimination

A common subexpression is a repeating expression inside an existing expression. In listing 3, the subexpression `17 * 34` is used three times in a row, meaning it can be optimized out as `578`, and because there are three compound addition operators, this can be translated into `v += 1743`. As a result, this branch that previously took several instructions now takes significantly less—an identifiable observation in a side-channel timing attack. There exist other optimizations that influence the likelihood of side-channel timing attack possibilities such as peephole optimizations, constant folding, and others. The broader research question is: how can we, as security researchers and cryptographers effectively communicate our needs to the compiler?

B. Safeguard-Removal Compiler Optimizations

In this section, we will describe how a compiler may remove safety precautions from the target code written by the programmer in the source code. We shall begin our discussion with a review of the idea behind persistent state violations. D'Silva et al. [4] define a persistent state violation as the accessibility of sensitive data in an prohibited location. We will describe three classes of vulnerabilities that may lead to persistent state violations.

1) Dead Store Elimination: Dead store elimination, as defined by CWE-14 [7] and CWE-733 [8], is an optimization that removes an assignment operation of a variable that is not read later in the code. A programmer's intent is to use this assignment as a security safeguard which removes sensitive data from memory. The compiler recognizes that the variable has no use outside after the assignment, so it assumes that the line is useless. This optimization, in turn, reduces target code size and the performance footprint if applied continuously in a large program. Dead store elimination, however, poses a major issue for programmers attempting to erase secret keys or passwords still in memory.

```
1  int foo(int x) {
2      return (x + 5) >> 1 & 0x7fff;
3  }
4
5  int secret_function() {
6      int secret = 0x681f2021;
7      int val = foo(secret);
8      secret = 0;
9      return val;
10 }
```

Listing 4. Dead Store Elimination

Listing 4, for instance, shows an example of a C function `secret_function` that uses a variable `secret`. This variable is used in the function call `foo` to perform some operation. Finally, it is overwritten on line 8. A compiler can see that line 8 is, effectively, superfluous and has no meaning outside of setting that variable to 0. Several previous research papers and experiments [4] [2] [5] [6] provide similar examples of dead store elimination as well as the compiled assembly. In our test, however, we found something rather interesting. Listing 5 shows a snippet of the functions in listing 4 compiled to assembly with GCC and the `-O1` and `-S` flags. Without any optimizations (i.e., the `-O1` flag), every line of target code is as it was in the source code, including the dead store. With the optimization flag enabled, on the other hand, it not only scrubs line 8, but also every other line in the function. An assumption about why this happens is, because `foo` is not a complex function (having only one arithmetic and two bitwise operations), the compiler attempts to inline the function. In the process, though, because `secret` is a constant, it evaluates the expression, and thus the function, at compile-time.

```
1  foo:
2  .LFB0:
3    .cfi_startproc
4    endbr64
5    leal  5(%rdi), %eax
6    sarl  %eax
7    andl  $32767, %eax
8    ret
9    .cfi_endproc
10 secret_function:
11 .LFB1:
12   .cfi_startproc
13   endbr64
14   movl  $4115, %eax
15   ret
16   .cfi_endproc
```

Listing 5. Assembly output of dead store elimination

2) Dead Code Elimination: Dead code elimination is very similar to dead store elimination, and CWE-561 [9] states that dead code is code that, provably, either does not affect program results or never executes. The difference between dead code and dead store elimination is that dead code is not exclusive to assignment operations, and holds a greater risk as a result. For example, suppose a C developer wishes to `memset` an array of sensitive data to zero after it has outlived its usefulness. Clearing an array of values to zero, generally, has no effect on the resulting program if that array/variable is dead from that point onward. As a proposed solution, the C11 standard introduced the `memset_s` function: a `memset` alternative that securely sets memory. Windows, on the other hand, provides the `SecureZeroMemory` function which promises that this function will never be optimized from memory. Being that the latter is a Windows-only solution, it omits any developer not using Windows or Microsoft's compilers. The former has the issue that `memset_s` is abnormally slow, is deemed optional by the C standards (meaning not all implementations have the function), and developers are slowly adopting it with little success [6].

Listing 6 shows an example of dead cod elimination. The programmer allocates an array of 100 integers and stores a secret value inside the array. After processing, to make sure the data is erased and cannot be accessed later in the program, we make a call to `memset` on line 12. In listing 7, though, this function call is removed from the generated assembly because the compiler proves that the array is never accessed again, so modifying its data is seemingly superfluous.

```
1  static const size_t SIZE = 100;
2
3  int main(void) {
4    int secret = 0x681f2021;
5    int *mem = malloc(sizeof(int) * SIZE);
6    mem[67] = secret;
7    for (int i = 0; i < SIZE; i++) {
8      mem[i] = (i * SIZE) & 0xff;
```

```
  }

  printf("%d\n", mem[67]);
  memset(mem, 0, SIZE);
  free(mem);
  return 0;
}
```

Listing 6. Dead code elimination of `memset` function

```
... ; Note that this ASM code starts at the loop.
.L2:
  movzbl  %al, %ecx
  movl  %ecx, (%rdx)
  addl  $100, %eax
  addq  $4, %rdx
  cmpl  $10000, %eax
  jne .L2
  movl  268(%rbx), %edx
  leaq  .LC0(%rip), %rsi
  movl  $1, %edi
  movl  $0, %eax
  call  __printf_chk@PLT
  movq  %rbx, %rdi
  call  free@PLT ; No call to memset function!
  movl  $0, %eax
  popq  %rbx
  .cfi_def_cfa_offset 8
  ret
  .cfi_endproc
```

Listing 7. Assembly output of dead code elimination

3) Function Inlining Problems: Secure functions, or functions that rely on the destruction of its activation record when it exits are vulnerable to a compiler optimization technique called function inlining. Sometimes, a function may be so short or simple that pushing its arguments to the call stack is unnecessary. Therefore, the compiler omits passing parameters and setting up the activation record altogether by copying the function code into its caller. The problem with eliminating the prologue and epilogue (i.e., activation record setup and destruction for the callee) is if a function is supposed to have a guarantee that data/variables/memory accessible in the function are secure for the lifetime of the callee, then inlining its body into the caller removes this guarantee, inducing a persistent state violation [4] [2].

IV. Future Work and Consequences

Compiler optimization research has unearthed several issues and open questions, many of which pertain to the relationship between the compiler and the using programmer. Several researchers [4] [2] [5] [6] indicate that there needs to be a stronger tie among compiler writers and software developers and cryptographers—communication about what is necessary for writing secure code is extremely pertinent. There is a severe power struggle and race between the groups, where software developers/cryptographers find that the existing tools to write secure code is often insufficient for their needs and have to take matters into their own hands. These workarounds, as we have mentioned, often result in undefined behavior and cause even greater problems. Plus, with compiler designers consistently discovering new and improved optimizations, it affects existing code because certain paradigms used in the "real world" do not always reflect theory and academic practices. Understanding these real-world consequences can further improve this communication pipeline. When software developers have to write workarounds and patches for unintended optimizations by a compiler or dig into the generated target code, it distracts them from their primary goals. An interesting point raised by Simon et al. [5] is that there will always be a battle between compiler writers/software writers and those adversaries wanting to attack their systems. Though, this should not be a catalyst for frustrations and updates that break existing code, introduce bugs, and force the developer to fight their tool instead of with their tool.

Additionally, most of the examples in this paper and current research tie back to the dangerous language of C. It is easy to dissect C and understand its flaws because of its age, relative size, and lack of complex features such as object-oriented programming in comparison to modern counterparts. We, however, question the effectiveness of modern compilers for modern and higher-level languages or platforms such as Rust via rustc, Java via OpenJDK, C# via Microsoft's Visual C#, and others. It appears that there is not a lot of research in this area yet due to its infancy compared to the existing work with C. Plus, modern solutions are designed to be portable, efficient, and most likely have learned from C's mistakes with manual memory management, undefined behavior, etc. Though, there has been some experimentation with WebAssembly and the emscripten compiler toolchain. Lehman et al. [10] wrote a report about the vulnerabilities generated from WebAssembly when compiling twenty-six open-source programs. Their static analysis showed that the WebAssembly target code generated vulnerabilities that had not been present in the native binaries for decades. While this is not strictly a compiler optimization in the sense that we have previously described, it demonstrates that even modern compilers and languages have (and even introduce) security reliability issues.

V. Conclusion

In this paper we investigated compiler optimizations and their relation to program security. We discussed previous research experiments which gained a better understanding of what types of code compilers tend to optimize, and the different optimizations that a compiler can perform. We also described cryptographic programming issues in which a compiler introduces holds in the target code that violate security invariants. Additionally, we mentioned the struggle between software developers and compiler designers/writers and the need to improve the staggering lack of communication. Several researchers proposed solutions to combat this communication problem that allow software developers to tag certain sections of code that inform compilers of its security significance. We discussed and posed several questions about the future of compiler optimization research in not only C code, but also modern languages such as Rust with modern compilers like rustc and even WebAssembly's framework. Compilers are incredibly complex tools and their abstraction allows software developers to focus on their code. Though, developers should not have to fight compiler optimizations that overwrite secure code or introduce vulnerabilities. It boils down to several important questions on the reliance of compilers and the

disconnect of software developers and compiler writers. We hope that future research improves the pipeline and allows for each developer to stay in their respective domain without having to constantly cross the streams.

ACKNOWLEDGEMENTS

This research paper was written for my graduate Principles of Computer Security (CSC 681) course at the University of North Carolina Greensboro in the Fall 2021 semester. All code examples were self-written, compiled, and tested.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] M. J. Hohnka, J. A. Miller, K. M. Dacumos, T. J. Fritton, J. D. Erdley, and L. N. Long, "Evaluation of compiler-induced vulnerabilities," *Journal of Aerospace Information Systems*, vol. 16, no. 10, pp. 409–426, 2019.

[3] "Insecure compiler optimization." [Online]. Available: https://owasp.org/www-community/vulnerabilities/Insecure_Compiler_Optimization

[4] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Security and Privacy Workshops*, 2015, pp. 73–87.

[5] 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)*, 2018, pp. 1–15.

[6] A. P. S. Venkatesh, A. B. Handadi, and M. Mory, "Security implications of compiler optimizations on cryptography – a review," 2019.

[7] "Cwe-14: Compiler removal of code to clear buffers," July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/14.html

[8] "Cwe-733: Compiler optimization removal or modification of security-critical code," October 2008. [Online]. Available: https://cwe.mitre.org/data/definitions/733.html

[9] "Cwe-561: Dead code," July 2006. [Online]. Available: https://cwe.mitre.org/data/definitions/561.html

[10] D. Lehmann, J. Kinder, and M. Pradel, *Everything Old is New Again: Binary Security of Webassembly*. USA: USENIX Association, 2020.