The Correctness-Security Gap in Compiler Optimization

Vijay D'Silva
Google Inc.
San Francisco, CA

Mathias Payer
Purdue University
West Lafayette, IN
mpayer@purdue.edu

Dawn Song
University of California, Berkeley
Berkeley, CA
dawnsong@cs.berkeley.edu

***Abstract*—There is a significant body of work devoted to testing, verifying, and certifying the correctness of optimizing compilers. The focus of such work is to determine if source code and optimized code have the same functional semantics. In this paper, we introduce the *correctness-security gap*, which arises when a compiler optimization preserves the functionality of but violates a security guarantee made by source code. We show with concrete code examples that several standard optimizations, which have been formally proved correct, inhabit this correctness-security gap. We analyze this gap and conclude that it arises due to techniques that model the state of the program but not the state of the underlying machine. We propose a broad research programme whose goal is to identify, understand, and mitigate the impact of security errors introduced by compiler optimizations. Our proposal includes research in testing, program analysis, theorem proving, and the development of new, accurate machine models for reasoning about the impact of compiler optimizations on security.**

I. Reflections on Trusting Compilers

Security critical code is heavily audited and tested, and in some cases, even formally verified. Security concerns have also led to hardware being designed with inbuilt security primitives. In this scenario, a compiler is the weak link between source code that provides security guarantees and hardware that is beginning to provide security guarantees. Concerns that compiler optimizations may render void the guarantees provided by source code are not new. Nearly half a century of work has been devoted to proving the correctness of compilers [16], [36], [42]. In fact, the question of whether a compiler can be trusted was raised prominently by Ken Thompson in his Turing award lecture [55].

In this paper, we highlight and study a situation, which we call the *correctness-security gap*, in which a formally sound, correctly implemented compiler optimization can violate security guarantees incorporated in source code. We make these assumptions to focus on the relationship between optimizations and security rather than design or implementation bugs in compilers.

A well-known example of the correctness-security gap is caused by an optimization called dead store elimination. The code below is derived from CWE-14 [1] and CWE-733 [2]. It contains a function `crypt()`, which we assume receives a key through a secure, protected channel. The function manipulates the key and scrubs the value of the key from memory before returning to the caller. Scrubbing is performed to avoid the key persisting in memory and eventually being discovered by an attacker or being captured in a memory dump.

```
crypt(){
 key = 0xC0DE; // read key
 ... // work with the secure key
 key = 0x0; // scrub memory
}
```

The variable `key` is local to `crypt()`. In compiler optimization terminology, the assignment `key = 0x0` is a *dead store* because `key` is not read after that assignment. Dead store elimination will remove this statement in order to improve efficiency by reducing the number of assembler instructions in the compiled code. Dead store elimination is performed by default in GCC if optimization is turned on [20]. This optimization is sound and has been proved formally correct using different techniques [7], [34].

To see why the optimization is problematic, consider a situation in which the method `crypt()` has been subjected to an extensive security audit and assume that all statements within `crypt()` execute with well defined semantics. There may however be weaknesses elsewhere in the application. The source code is designed to be secure against exploits that access values that persist in memory. The compiled code does not preserve this guarantee despite dead store elimination being a sound optimization and despite the code in `crypt()` having well-defined semantics.

The example above illustrates a gap between the guarantee provided by the compiler and the expectations of a developer. The compiler guarantees that the code before and after compilation computes the same function. The developer has additionally assumed that the optimized code will leave a system's memory in the same state as the unoptimized code. There has been much debate about (i) whether a compiler should preserve security guarantees incorporated in source code, (ii) whether it is the developer's responsibility to understand the impact of compiler optimizations on their code, and (iii) whether the behaviour above qualifies as a compiler bug. See the mailing list archives of the GNU compiler collection (GCC) [58], [22], [54], [59] and Linux kernel [56], [26] for examples of such exchanges.

A. The Gap between Correctness and Security

The literature contains further examples of compiler bugs that affect the correctness and security of operating system kernels and user applications [45], [46], [47], [29], [30], [31], [62], [51]. Given the large number of optimizations implemented in compilers and the volume of work on compiler correctness, we find it natural to ask two questions: (i) What are other examples of formally sound, widely implemented compiler optimizations that may violate a security guarantee in source code? (ii) Given that these optimizations have been proved correct, why does the proof of functional equivalence not translate into a proof of security equivalence? In this paper, we study these two questions. We give examples of standard compiler optimizations and scenarios in which optimizations introduce vulnerabilities into code. We also examine the structure of compiler correctness proofs and propose a model for studying the gap between correctness and security.

The intuition behind our analysis stems from a few observations. The semantics used for reasoning about source code or intermediate representations is based on details from a language standard. For example, in the bug report in [59], a GCC developer emphasises that GCC 3.2 only attempts to preserve the semantics of C as specified by ISO 9899. In particular, aspects of compiled code such as the memory footprint, the size of activation records, stack usage, or time and power consumption are often not specified by the language standard. Consequently, even if source code has been designed to defend against attacks that exploit vulnerabilities in these aspects of a computer system, an optimizing compiler is not guaranteed to preserve those defense mechanisms.

Formal techniques for compiler correctness attempt to establish an equivalence between the states at the beginning and end of a code unit before and after an optimization is applied. To precisely identify the gap between the correctness guarantees available about compiler optimizations and the security guarantees we would like to have, we examine the details of compiler correctness proofs. The most detailed semantics considered when reasoning about compiler optimizations is the small-step operational semantics of the source and machine code. Certain proof techniques consider a more abstract, denotational semantics. The small-step operational semantics accounts for the state of a program but not the state of the underlying machine. Optimizations may change the state of the underlying machine in addition to the state of the program. If ignoring this difference is crucial to the correctness proof, we have a situation in which there is a potential gap between correctness and security.

We emphasise that the observation that a compiler can violate a security guarantee in source code is not new to this paper. We also emphasise that we are not claiming optimizing compilers are buggy because they violate criteria they were not designed to satisfy. Rather, we take the view that the relationship between optimizations and security constraints loosely resembles the relationship between optimizations and modern memory models. Most optimizations were historically designed assuming a memory model that guaranteed a property called *sequential consistency*. When relaxed memory models became prevalent [5], the correctness of these optimizations had to be re-evaluated [4], [21]. Testing and formal analysis tools were developed to discover instances in which existing optimizations produced incorrect results. Mechanisms such as barriers and fence insertion were developed to adapt existing optimizations to new architectures. Similarly, our work is an early step towards understanding which security guarantees are violated by existing compiler optimizations and how these optimizations can be adapted to provide the desired guarantees.

Another perspective on our work is provided by the *weird machines* paradigm [8]. Exploit generation research can be viewed as a constructive proof that a system's runtime enables the construction and execution of a *weird machine*. Defenses against exploits can be understood as eliminating gadgets essential for constructing or deploying such a machine. A compiler optimization that introduces a correctness-security gap is one that reintroduces into the runtime a gadget the developer presumes to have eliminated.

B. Problem and Contribution

We study two problems in this paper. The first problem is to identify compiler optimizations that are sound with respect to the standard semantics of a program but are unsound if security is taken into account. The second problem is to analyze the cause of this discrepancy. We make the following contributions towards solving these problems.

1) We identify instances of the correctness-security gap involving standard, formally verified optimizations such as dead code elimination, code motion, common subexpression elimination, function call inlining, and peephole optimization.
2) We analyze the structure of existing compiler correctness proofs to identify why the correctness-security gap arises. We show that a more refined machine model, combined with existing proof techniques allows for this gap to be eliminated and for security violations to be detected.
3) We identify open problems that must be tackled to address the gap between correctness and security in theory and in practice.

Note that there are simple strategies to eliminate security bugs introduced by optimizations. An extreme option is to disable all optimizations and require that the compiled code execute in lockstep with the source. This approach leads to an unacceptable performance overhead (e.g., 5x [38]). Another option is to avoid certain incorrect transformations using the `volatile` keyword. A number of security bug reports show that developers do not always use such mechanisms correctly and that even compilers do not compile `volatile` correctly [18]. Moreover, using `volatile` is not a solution for many optimizations that we consider.

The paper is organised as follows. We begin with a brief review of compiler optimization terminology in Section II and then presents instances of the correctness-security gap in Section III. We recall the structure of compiler correctness arguments in Section IV. In Section V we analyze the gap and show how it could be avoided using a language semantics that considers the state of the machine in which code executes. Section VI discusses research directions for detecting and defending against such behaviour, as well as implementing such detection. We conclude in Section VIII after reviewing related work in Section VII.

II. A Compiler Optimization Primer

This section contains a recap of compiler optimization terminology. This section is included to keep the paper self-contained and is not intended to be comprehensive. See standard textbooks for more details [6], [39].

The structure of an optimizing compiler is shown in Figure 1. The clear boxes represent data and shaded boxes represent processing steps. A compiler takes *source code* as input and generates *target code*. The target code may run on a processor, in a virtual machine, or in an interpreter. The internal operation of a compiler can be divided into four phases consisting of syntactic processing, semantic checks, analysis and transformation, and code generation.

The syntactic processing phase takes source code represented as a string as input and generates an *Abstract Syntax Tree* (AST). A *lexer* turns source code into a sequence of tokens. A *parser* turns this sequence into an AST. The syntactic processing phase rejects programs that contain syntax errors. *Semantic checks* are then applied, in what is sometimes called the *static semantics phase*. These checks ensure that the program satisfies restrictions of the programming language such as a type discipline and conventions for declaration and use of variables and libraries.

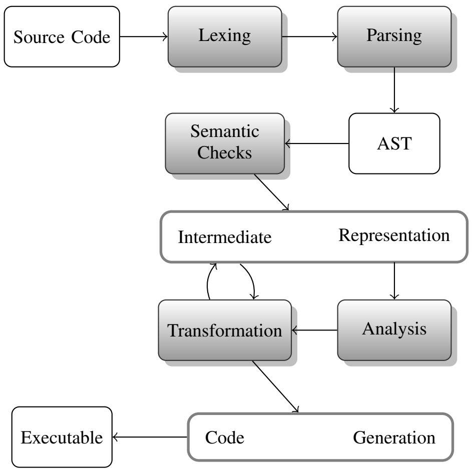


Figure 1. The architecture of an optimizing compiler.

The analysis and transformation phase operates on one or more intermediate representations (IR) of a program. An IR typically has a small set of instructions, is independent of the target architecture and may even be independent of the source language. Intermediate representations used by GCC include RTL, GIMPLE, and GENERIC, while the LLVM infrastructure uses its own IR. Analysis of the IR is used to deduce information about the program such as identifying code that is never executed, variables whose values never change, computations that are redundant, etc. The results of analysis are used to perform architecture independent optimizations. The IR obtained after transformation is supposed to represent a program that contains less redundancy and is more efficient than the source. One transformation may enable others, so a series of analysis and transformations are performed with details depending on the compiler implementation.

The final phase in compilation is *code generation*, in which the IR is translated into either byte code or machine code. If machine code is generated, the compiler may perform architecture specific analysis and transformations to further optimize the code.

Compiler Correctness: We give an intuitive definition of a compiler bug here and provide formal definitions in Section IV. A compiler has a bug if the behaviour of the compiled code deviates from that of the source code as specified in the language standard. Compiler correctness is concerned with finding or proving the absence of bugs in compilers. Much work on compiler correctness focuses on architecture independent optimizations because errors in these optimizations have the widest impact.

III. A Security-Oriented Analysis of Optimizations

In this section we present examples of code that performs secure operations and identify optimizations that violate the security guarantees in source code. All the optimizations we consider respect a specific language specification and have often been formally verified. Our examples use C syntax but are applicable to other languages as well.

We identify three classes of security weaknesses introduced by compiler optimizations: (i) information leaks through persistent state, (ii) elimination of security-relevant code due to undefined behaviour, (iii) introduction of side channels. For each class above, we present examples of code that has been designed to defend against an attack in that class and identify a compiler optimization that weakens or eliminates this defense. These examples highlight gaps between the expectations of a developer and guarantees provided by a compiler.

The security violations we discuss can, in principle, be circumvented using language mechanisms such as the **volatile** keyword, inserting memory barriers, or using inline assembler. Code compiled with Microsoft Visual C compiler can avoid some violations using **#pragma** `optimize("", off)`. We believe that programming in this manner is undesirable for several reasons.

First, it amounts to a developer adopting an adversarial view of the compiler and defending code against compiler optimizations. This in turn complicates software design and implementation and requires knowledge of compiler internals. Moreover, experience indicates that developers misunderstand the semantics, performance, and security implications of **volatile** and related mechanisms. In fact, even compiler writers misunderstand and incorrectly implement such mechanisms [18].

A second reason is that problems due to the gap between security and correctness arise in existing software, as witnessed by existing CWEs and CVEs [15], [1], [2]. Changing existing practice still leaves open the problem of detecting such security issues in existing code. Finally, such mechanisms may lead to coarse-grained optimizer behaviour, where either all optimizations are turned off in a region of code, or all optimizations are applied.

We argue for a more nuanced analysis of compiler optimizations. Understanding the context in which an optimization causes a security violation can lead to the design of protection mechanisms that combine the security guarantees provided by source code with performance improvements provided by a compiler.

A. Persistent State

A persistent state security violation is triggered when data lingers in memory across developer enforced boundaries. A compiler optimization that manipulates security critical code may cause information about a secure computation to persist in memory thereby introducing a persistent state violation. We identify dead store elimination, code motion, and function call inlining as optimizations that can lead to persistent state security violations.

1) Dead Store Elimination: A *store* is an assignment to a local variable. A store is *dead* if a value that is assigned is not read in subsequent program statements. Dead Store Elimination leads to improved time and memory use, especially if a dead store occurs in a loop or function that is called repeatedly.

Operations that scrub memory of sensitive data such as passwords or cryptographic keys can lead to dead stores. Instances of dead-store elimination removing code introduced for security [26], [56], [59] have lead to the dead-store problem being classified as CWE-14 [1] and CWE-733 [2].

```
char *getPWHash() {
  long i; char pwd[64];
  char *sha1 = (char*)malloc(41);
  // read password
  fgets(pwd, sizeof(pwd), stdin);
  // calculate sha1 of password
  ...
  // overwrite pwd in memory
  // Alternative (A) : use memset
  memset(pwd, 0, sizeof(pwd)); // (A)
  // Alternative (B) : reset pwd in a loop
  for (i=0; i<sizeof(pwd); ++i)
    pwd[i]=0; // (B)
  // return only hash of pwd
  return sha1;
}
```

Listing 1. Both (A) and (B) are opportunities for dead store elimination.

Listing 1 contains a function that accepts a password from the user and stores it in memory during the calculation of a hash. After the computation, the password is removed from memory using either (A) `memset` or (B) a **for** loop that resets each character in the password. Since the variable `pwd` is dead, the lines (A) and (B) may be removed by an optimization.

Dead store elimination is a well known, sound optimization. It does not change the functionality of the code, but does affect parts of the state that are relevant for security. Microsoft libraries provide a `SecureZeroMemory()` function to defend against this problem.

2) Function Call Inlining: Function call inlining, also called inline expansion, replaces a function call site with the body of the function being called. Function call inlining merges the stack frames of the caller and the callee and avoids executing the prologue and epilogue of the callee. This optimization eliminates the time and space overheads of calling a function but increases the size of the caller's body. which in turn affects the instruction cache. The performance implications of inlining are not simple, but it may enable optimizations such as code motion, constant propagation, copy propagation, or constant folding and the cumulative effect of these optimizations can improve performance. LLVM and GCC both apply inlining using heuristics based on function length and hotness of code.

Listing 2 illustrates how function call inlining can affect security-sensitive code. Let `getPWHash()` be the function in Listing 1 and `compute()` be a function using a password hash. If the call to `getPWHash` is inlined, the stack frames of `getPWHash()` and `compute()` are merged.

Consider a scenario in which `getPWHash()` has been audited and the developer believes the contents of the stack frame during the execution of `getPWHash()` are secure. After inlining, the local variables of `getPWHash`, which contain sensitive information and lived in a secure stack frame, will now be alive for the lifetime of `compute()`

```
char *getPWHash() {
  // code as in Listing 1
}
void compute() {
  // local variables
  long i, j;
  char *sha;
  // computation
  ...
  //call secure function
  sha=getPWHash();
  ...
}
```

Listing 2. A candidate for function inlining.

```
// Code before optimization
int secret = 0;
if (priv)
  secret = 0xFBADC0DE;
```

```
// After applying code motion
int secret = 0xFBADC0DE;
if (!priv)
  secret = 0;
```

Listing 3. A candidate for code motion.

and be included in an insecure stack frame.

Inlining increases the longevity of variables in a stack frame and can eliminate the boundaries between certain stack frames. If a developer uses function boundaries (hence stack frames) to implement trust-separated domains, security guarantees provided by the code can be violated by inlining because the compiler is not guaranteed to respect these boundaries. A naive defense would be to execute a cleanup function that overwrites the upper portion of a stack frame so that secure variables have the same lifetime before and after a transformation.

3) Code Motion: Code Motion allows the compiler to reorder instructions and basic blocks in the program based on their dependencies. For example, code in a loop that is independent of the rest of the loop can be hoisted out of the loop, eliminating redundant computation. If a compiler proves that a statement does not affect a block of subsequent statements, the statement can be executed after that block. Symmetrically, a statement not affected by a block of preceding statements can be executed before that block.

We show how code motion introduces a persistent state vulnerability. Consider the code before optimization in Listing 3. If a compiler concludes that the **if** branch is part of a *hot path*, meaning that it will be executed often, the code can be transformed so that the assignment `secret = 0xFBADC0DE;` is performed by default without first evaluating a branch condition.

Suppose the flag `priv` is set to true if the code is executing in a trusted environment with appropriate privileges. In the code before transformation, the check `if (priv)` ensures the assignment of the secret value happens in a secure execution environment. After code motion, the secret value is always assigned because this code structure leads to better performance. As with inlining, the compiler is not required to respect the security-related assumptions the developer makes and consequently, the compiled code does not provide the same security guarantees as the source.

Code motion, like inlining, affects the layout of stack frames, the liveness of variables, and the timing of individual execution paths through the control flow graph. This optimization affects security guarantees concerning trust domains and execution times.

Persistent State Discussion: We showed how dead store elimination can remove memory scrubbing code, function call inlining can extend the lifetime of secure variables, and code motion can eliminate security checks. In all these examples, the source code attempted to prevent sensitive data from being accessible after a secure computation. The optimization allowed data from a secure computation to persist in memory longer than intended. Even if an application has no direct pointer to the secure data, the information may still be accessible using an out-of-bounds pointer or a memory-safety violation triggered elsewhere in the code.

B. Undefined Behaviour

The term *undefined behaviour* refers to situation in which the behaviour of a program is not specified. Language specifications deliberately underspecify the semantics of some operations for various reasons such as to allow working around hardware restrictions or to create optimization opportunities. Examples of undefined behaviour in C are using an uninitialized variable, dividing by zero, or operations that trigger overflows. Consult these papers and blog posts for detailed discussions of undefined behaviour [17], [24], [29], [30], [31], [45], [46], [47].

Language specifications leave it to the developer to avoid undefined behaviour but do not restrict what a compiler should do to executions that trigger undefined behaviour. Compilers may opportunistically assume that the code following a computation that triggers undefined behaviour can be deleted [29], [48]. Such deletions are permitted by the language standard. This flexibility is intended to allow for more performance optimizations but is a double-edged sword that can lead to security issues [30], [60].

We say that an undefined behaviour violation occurs if assumptions about undefined behaviour are used to eliminate security checks and make code susceptible to an attack. Listing 10 shows a function that takes an integer variable `nrelems` as input and that allocates an array of `nrelems`

```
1 int *alloc(int nrelems) {
2   // Potential overflow.
3   int size = nrelems*sizeof(int);
4   // Comparison that depends on
5   // undefined behaviour.
6   if (size < nrelems) {
7     exit(1);
8   }
9   return (int*)malloc(size);
```

Listing 4. Undefined behaviour in signed integer overflows.

integers. The function attempts to determine if an overflow occurs in a multiplication expression. Signed integer overflow is undefined according to the C/C++ language specification so the compiler removes the overflow check (lines 6 to 8). In this deliberately simple example, the code after the check terminates execution. More generally, eliminating such code can expose the application to exploits based on a buffer overflow or memory allocation bugs.

Undefined Behaviour Discussion: There has been much debate about whether the developer or the compiler should be responsible for preventing vulnerabilities introduced by undefined behaviour [58], [61]. It is beyond debate that optimizations based on undefined behaviour have introduced exploitable vulnerabilities in the Linux kernel [12], [51], [56], in particular, CVE-2009-1897 [15]. Due to these security implications, GCC and Clang generate warnings for some but not all instances of undefined behaviour [31], [3]. Tools like STACK [60], which is based on LLVM, and [24], which is based on an executable semantics for C [19], detect a wider range of undefined behaviours.

C. Side Channel Attacks

A side channel leaks information about the state of the system. Side channel attacks allow an attacker external to the system to observe the internal state of a computation without direct access to the internal state. We say that a side channel violation occurs if optimized code is susceptible to a side channel attack but the source code is not.

On most platforms, the number of processor cycles required provides a way to measure and compare the time complexity of instructions. For example, floating point division takes longer than integer addition and moving a value into a register takes less time than loading a value from memory. Accurately estimating such timing is a subtle task that requires considering the memory hierarchy and possible cache misses on multiple levels.

Attackers can use timing information to infer security sensitive data such as cryptographic keys [10], [27], [41], [43]. To counter side-channel attacks based on timing, implementers of cryptographic algorithms take care to ensure that the number of instructions executed is the same regardless of data such as a cryptographic key or the message to be encrypted. Such code is deliberately written with inline assembler to defend against side channel violations introduced by compiler optimizations.

1) Common Subexpression Elimination: An expression is a common subexpression of two different statements if it occurs in both. If the variables in a subexpression common to a set of statements have the same value in each of those statements, that subexpression can be calculated once and all occurrences of the subexpression can be replaced with a pre-calculated value. Common Subexpression Elimination (CSE) is an optimization that discovers and eliminates subexpressions of identical value from different statements. For example, `x + y` is a common subexpression in `y = x + y + z;` and `z = x + y - 2z;`. If the variables `x` and `y` have the same value in both the assignments, a new variable and assignment `s = x + y;` can be introduced and the two assignments can be rewritten to `y = s + z;` and `z = s - 2z;`. This optimization can improve the performance of code especially if the common subexpression is complex or is evaluated in a loop.

CSE affects the timing of instructions and consequently, defenses against timing side channels. Listing 5 contains a simple example in which the developer has ensured that both branches of the **if** statement perform the same amount of computation. CSE allows for code in the **else** branch to be simplified so that instead of a multiplication and addition operation being performed three times, a multiplication and addition is performed once followed by a single multiplication. An attacker who can measure these timing differences would be able to observe which branch is taken. A more sophisticated timing analysis may also reveal the kinds of operations being performed.

2) Strength Reduction: The strength of an expression is a measure of how processor intensive it is to evaluate. The strength of an expression can be reduced if it can be written using simpler operations. For example, certain multiplication or exponentiation operations can be replaced by bit ma-

```
int crypt(int k*){
int key = 0;
if (k[0]==0xC0DE){
  key=k[0]*15+3;
  key+=k[1]*15+3;
  key+=k[2]*15+3;
} else {
  key=2*15+3;
  key+=2*15+3;
  key+=2*15+3;
}
return key;
```

Listing 5. Before optimization.

```
int crypt(int k*){
int key = 0;
if (k[0]==0xC0DE){
  key=k[0]*15+3;
  key+=k[1]*15+3;
  key+=k[2]*15+3;
} else {
  // replaced by
  tmp = 2*15+3;
  key = 3*tmp;
}
return key;
```

Listing 6. After CSE.

```
int crypt(int k){
  int key = 0x42;
  if (k==0xC0DE){
    key=key*15+3;

  } else {
    key=2*15+3
  }
return key;
```

Listing 7. Before optimization.

```
int crypt(int k){
  int key = 42;
  if (k==0xC0DE){
    key=(key<<4) \
         -key+3;
  } else {
    key=33;
  }
return key;
```

Listing 8. After strength reduction.

nipulation operations. Strength reduction has performance benefits especially for idiomatic expressions in loops.

The function `crypt` in Listing 7 encrypts and returns a value passed as an argument. The developer has ensured that both branches of the conditional have one multiplication and one addition operation. After strength reduction, the expression in the **if** branch is modified to use a bit-shift and addition operation, while the **else** branch is simplified to a constant value. An attacker with access to an appropriate timing channel can observe more about the difference between the two branches after the optimization than before. Similar to CSE, strength reduction can increase the amount of information available via timing side-channels because it replaces expressions evaluated at runtime with static values or modifies the time-complexity of a computation.

3) Peephole Optimizations: Peephole optimizations are one of the last phases performed by compilers before emitting code. Compilers examine instructions through a small, sliding window and use pattern matching to reorder or exchange instructions to improve some form of performance (e.g., to reduce cache misses, increase throughput, or improve branch behaviour). Similar to CSE and strength reduction, peephole optimization may create a timing based side channel that is explicitly eliminated in source code.

Side Channel Discussion: If code is written with a specific timing cost model in mind, almost every optimization, and the three presented in particular, will destroy the timing guarantees in source code. An outside observer can exploit these timing differences to extract information about operations performed by the application.

Some developers writing security-critical code are aware of the impact of compiler optimizations. Timing guarantees are enforced by writing inline assembler because compilers typically do not modify such code. This approach foregoes the benefits of programming in a high-level language and leads to code that is not portable and not readable. Moreover, optimizations can still enable side channels based on cache hierarchies [9] or virtual machines [63].

IV. A View of Compiler Correctness

The optimizations in the previous section have all been formally shown to be correct. In this section and the next, we examine correctness proofs to identify the gap between functional correctness and security requirements. In this section, we present a high-level summary of the arguments used for showing the correctness of an optimization.

Program Semantics: We consider a small-step operational semantics for programs. A *state* of a program is a valuation of all entities that affect execution. We assume that the semantics of a language defines a set of state changes $[\![st]\!] \subseteq \mathit{States} \times \mathit{States}$ for each statement st. A *labelled transition system* is a tuple $(\mathit{States}, \mathit{Rel}, \ell)$ consisting of a set of states States, a *transition relation* $\mathit{Rel} \subseteq \mathit{States} \times \mathit{States}$ and a *labelling function* $\ell : \mathit{Rel} \to \mathit{Stmt}$ that maps (s, t) to st if st is the statement causing the state change. The labelling function associates a program statement with each transition. We assume that the semantics of a program P is a labelled transition system, denoted $[\![P]\!]$.

Observational Equivalence: An attacker typically has an incomplete view of a system and can access part of a program's state at certain stages in execution, but may not have access to the entire state. We now formalise observations an attacker can make about a program.

An *observation space* $(\mathit{vis}, \mathit{obs})$ is a pair of functions $\mathit{vis} : \mathit{States} \to \mathbb{B}$ and $\mathit{obs} : \mathit{States} \to \mathit{Obs}$, where $\mathit{vis}(s)$ is `true` if a part of s is visible to the attacker and obs maps a state s to an observation if $\mathit{vis}(s)$ is `true`.

Example 1. This example illustrates how different observations can be formalised. Consider the two programs below. Assume that all states are visible but the attacker can only observe the last bit (parity) of a variable. This attacker cannot distinguish between the programs below. If an attacker has a side channel that counts the number of statements executed, the two programs are still indistinguishable.

```
// Prog 1
int main(){
    int x = 5;
    int y = x+1;
}
```

Listing 9. Program 1

```
// Prog 2
int main(){
    int x = 3;
    int y = x+3;
}
```

Listing 10. Program 2

Consider another situation where only the final state of the program is visible but an attacker can access the entire state. Such an attacker can distinguish between the programs above because the programs terminate with different values for `x`. If, however, the attacker can only see the value of `y`, the programs become indistinguishable. ⌟

We formalise observations of an execution. Let (S, E, ℓ) be a transition system. An *execution* is a sequence of states that is either an initial state s_0 or the concatenation $\pi \cdot s$ of an execution π that ends in state r, provided (r, s) is in E. Let Exec be the set of executions. An *observation of an execution* is a partial function $\mathit{obs} : \mathit{Exec} \to \mathit{Obs}^*$ that is defined on an execution π if some state on π is observable. The observation of s is $\mathit{obs}(s)$ if $\mathit{vis}(s)$ is true. The observation of $\pi \cdot s$ is $\mathit{obs}(\pi)$ if s is not visible, and is $\mathit{obs}(\pi) \cdot \mathit{obs}(s)$ otherwise.

Definition 1. A transition system P is observationally subsumed by Q if for every execution π of P, if $\mathit{obs}(\pi)$ is defined, there exists an execution τ of Q such that $\mathit{obs}(\tau)$ is defined and $\mathit{obs}(\pi) = \mathit{obs}(\tau)$. P and Q are observationally equivalent if P observationally subsumes Q and Q observationally subsumes P.

Observational equivalence formalises that the attacker cannot distinguish between the two systems. It does not guarantee that a system is secure, only that the two systems provide the same security guarantees.

At a high level, a compiler maps between programs in different languages. Formally, a *transformation* is a map $\mathit{trans} : \mathit{Lang}_1 \to \mathit{Lang}_2$ between programs of two (possibly identical) languages and a compiler is a composition

$$\mathit{Comp} \mathrel{\hat{=}} \mathit{trans}_1 \circ \cdots \circ \mathit{trans}_k \text{ of transformations}$$
$$\mathit{trans}_i : \mathit{Lang}_i \to \mathit{Lang}_{i+1}.$$

In this paper we are interested in a guarantee of observational equivalence. The guarantee provided by sound compiler optimizations and formally verified compilers can be viewed as an instance of observational equivalence.

Definition 2. A compiler Comp guarantees observational equivalence with respect to an observation space $(\mathit{vis}, \mathit{obs})$ if for all programs P, the labelled transition systems $[\![P]\!]$ and $[\![\mathit{Comp}(P)]\!]$ are observationally equivalent.

We say that a compiler (or an optimization) is *correct* or is *sound* if it guarantees observational equivalence. The definition above is parameterized by a notion of observation, which allows us to apply it to different scenarios. The choice of observation is typically left implicit in the literature, and we shall see that making it explicit is insightful for a discussion of security issues.

Games for Observational Equivalence

Conducting a direct proof of observational equivalence for every optimization is tedious and does not exploit the structure of the program. It is customary to use a structural notion like (weak-) bisimulation or simulation [37], [50] in proofs [28], [34]. We adapt the game-based definition of bisimulation [53] to our setting[1].

We formulate a game between a security analyst and a compiler writer. The security analyst attempts to find an execution of the transformed code that reveals information not observable by executing the source code. The compiler writer attempts to show that the transformation preserves security by finding a source execution that generates the same observations. We discuss the structure of this game because it is relevant for the analysis in Section V of why correctness proofs do not guarantee security.

Definition 3. Let P and Q be programs and $(\mathit{vis}, \mathit{obs})$ be an observation space. The position of the analyst is a visible state t in $[\![P]\!]$ and a position for the writer is a visible state s in $[\![Q]\!]$ that satisfies $\mathit{obs}(s) = \mathit{obs}(t)$. In each round, the players make moves based on these rules.

1) The analyst chooses a finite execution from t to a visible state t'.
2) The writer responds by choosing a finite execution from s to a visible state s' so that $\mathit{obs}(s') = \mathit{obs}(t')$.

The analyst has the first move and must choose an initial state of P. The analyst wins the subsumption game if the writer has no move in a round. If the analyst has a winning strategy starting from P or starting from Q, we say that the analyst has a winning strategy in the observational equivalence game. The writer wins the equivalence game if the analyst does not win.

In game-theoretic terms, the game above is an infinite, two-player game of perfect information. The Borel determinacy theorem guarantees that there will be a winner in such a game [50], [53]. The game above relates to observational equivalence by a theorem of the form below.

Theorem 1. If P and Q are two programs and the writer has a winning strategy in the observational equivalence game, then P and Q are observationally equivalent.

The proof is similar to but more involved than proofs that bisimulation implies trace equivalence. Given an execution of the transformed program, we use the winning strategy of the writer to derive an execution of the original program. The conditions of the game guarantee that such an execution exists and the two executions are observationally equivalent.

To prove that a transformation is correct, it is sufficient to prove that there is a winning strategy for the writer when the game is played on $[\![P]\!]$ and $[\![\mathit{Comp}(P)]\!]$. In the next section, we indicate why arguments with this structure fail to preserve security guarantees included in source code.

V. Analysis of the Correctness-Security Gap

In this section, we analyze the correctness-security gap and suggest how correctness proofs can be extended to close this gap. An intuitive explanation for this gap is that the operational semantics used in correctness proofs includes the state of the program, *but not the state of the underlying machine*. The first step to close the gap is to more accurately model the state of the execution environment. An accurate model of state is not sufficient to detect attacks (especially side-channel ones) because an accurate model of state transitions is also required.

[1] See http://www.brics.dk/bisim/ for animations of bisimulation games.

We make the intuition above precise using the notion of an abstract machine; a mathematical idealization of the machine in which code executes. Rather than associating a single semantics $[\![P]\!]$ with a program, we propose considering a semantics $[\![P]\!]_M$ of a program executing in an abstract machine M. Existing correctness proofs are conducted assuming a certain model (which we call the source machine) while security properties are defined with respect to other machine models. We now identify some abstract machines that can be used in reasoning about the security of optimizations.

Note that the abstract machines we consider are *host machines*, meaning they host a program and define its runtime. Abstract machines are distinct from the *weird machines* of [8], because a weird machines are constructed within a runtime and exploits are programs that run on a weird machine. In contrast, abstract machines define the context in which code executes. Nonetheless, the analysis of exploits using weird machines in [57] uses a remarkably similar game-theoretic formalism.

A. The Source Abstract Machine

We use the term *source machine* for the standard execution model that is assumed when defining the semantics of a language. For example, the C language standard defines a C abstract machine, which is an instance of a source machine. The state of a source machine defines the values of variables, the configuration of the call stack, and the contents of the heap. We describe these components in detail to arrive at a mathematical description of a state of the source machine.

A *value* is either a value of a primitive type such as **`bool`**, **`char`**, or **`int`**, a memory location, or is a special undefined value, denoted **`undef`**. The value **`undef`** is used for the result of overflows, division by zero, the contents of uninitialised memory locations, and other situations that the language standard declares the result of a computation is undefined [29], [30], [31], [45], [46], [47]. Let Val be the set of values.

A source machine typically includes a memory model. Details of the memory model are required for identifying where a correctness proof fails to respect security assumptions. For this discussion, we assume the existence of a memory model, such as the one in CompCert [35].

Let $LVar$ be the set of local variables and $GVar$ be the set of global variables in a program. An *environment* maps variables to values. We write $LEnv$ and $GEnv$ for local and global environments, defined below. The *program counter* indicates the next statement to be executed and PC is the set of program counter values. The *stack* is modelled by a sequence consisting of program counter values and local environments, also defined below.

$$LEnv \mathrel{\hat{=}} LVar \to Val \qquad Stack \mathrel{\hat{=}} (PC \times LVar)^*$$
$$GEnv \mathrel{\hat{=}} GVar \to Val$$

A state of the source machine consists of a global environment, a stack and a heap. Computation within a function can modify the top of the stack, global environments, and the heap, but not the contents of the stack.

$$States \mathrel{\hat{=}} GEnv \times Stack \times Mem$$

A function call causes a new element to be pushed on the stack and a return pops the top of the stack. The stack becomes empty only when the program completes execution.

Issues with the Source Machine: The source machine is based on the language standards and has sufficient detail for proving the correctness of a compiler transformation. However, this machine lacks various details relevant for security. We discuss these details below, assuming a runtime model based on C. The criticisms levied below apply to the correctness-security gap in C compilers.

A first inaccuracy arises from the structure of the machine. The source machine is based on the Harvard architecture, which assumes a separation of control and data, while machines in use today are closer (though not identical) to the von Neumann architecture in which instructions and data both reside in memory. Leveraging the coexistence of control and data is fundamental for writing exploits, but this feature is not captured by the source machine.

A second inaccuracy comes from use of uninitialised local variables and exceptional behaviour, which result in the **`undef`** value. Formal models do not allow **`undef`** to be used in computations. During execution however, there is no special **`undef`** value. Registers are not initialised to **`undef`** after being allocated to new variables, so the old value stored in memory can be accessed and used even if such an access violates the language semantics.

A third inaccuracy, which connects to persistent state violations, is what we call *persistence assumptions*. The source machine assumes that stack and memory contents *are not persistent*. A block that has been deallocated cannot be accessed again and the values it contained are lost. Once a function returns, the values of local variables used by that function are lost. These assumptions are not satisfied by the machine in which a program executes and contribute to the security blind-spot in compiler correctness frameworks.

A fourth family of inaccuracies arises from abstractions that are made to ignore details not relevant to correctness. The source machine ignores details of load and store operations at the processor level, processor specific constructs, and time and power consumption. Additionally, correctness proofs, such as in [28], [35], use a small-step semantics for statements, but a big-step semantics for expression evaluation. The steps involved in expression evaluation are important when reasoning about cryptographic libraries. Ignoring these details is important to facilitate logical reasoning at a high level, but this idealisation leads to correctness arguments that are blind to side-channels.

B. Refinements of the Source Machine

Despite the criticisms of the source machine, it is a convenient and popular formalism for reasoning about compiler correctness. It is important to be aware that it is a starting point, so proofs conducted at this level may not preserve lower-level properties. Such a statement is obvious in retrospect, but to the best of our knowledge has not be made in the context of security and compilation.

We now propose refinements of the source machine that would be suitable for a security-oriented analysis. Our goal is not to work out the details of these machines in full, but to propose various models that can be used in future analysis.

x86 Assembler Machine: The x86 instruction set can be viewed as defining an *x86 assembler machine*. We use the term “assembler machine” because it is not identical to the physical x86 architecture. The x86 assembler machine provides semantics to x86 assembly language but abstracts away from architectural details. The x86 machine consists of registers, an instruction pointer, a set of flags, and the memory. States of the x86 machine define the set

$$\mathit{xStates} \mathrel{\hat{=}} \mathit{rStates} \times \mathit{iPtr} \times \mathit{Flags} \times \mathit{xMem}$$

which contains the states of registers, the value of the instruction pointer, the status of the flags, and the configuration of memory. A state transition can be viewed as a single fetch-decode-execute cycle, in which case this model abstracts away from clock-cycle level details.

Memory Hierarchy and Timing Machines: The x86 assembler machine above does not model memory hierarchy details. The caching policy combined with the sequence of reads in the program affects the number of cache misses. An attacker capable of measuring the time required for a memory lookup can detect, based on latency, whether a cache hit or miss occurred. Such partial information has been used to extract cryptographic keys from implementations of AES, RSA and DSA [43]. Cache-based attacks have reconstructed AES keys in the order of minutes of execution [23].

Formal reasoning about timing attacks is facilitated by extending an abstract machine with time. For example, a *timed source machine* contains *timed states* $\mathit{tStates} \mathrel{\hat{=}} \mathbb{N} \times \mathit{States}$, which extend source machine states with time stamps. The semantics of a program with respect to a timed source machine would have incremented the timestamp with every statement that is executed. The increment would depend on the timing cost of the statement. A timed source machine may be too abstract to model certain attacks in which case the x86 machine can be extended with timing information. Such an extension would allow for time-differences in expression evaluation to be modelled.

Semantics in a Machine: To use an abstract machine M in analysis of compiler correctness, one requires a framework for defining the semantics $[\![P]\!]_M$ of a program P executing in M. Such a framework must provide a convenient way for describing how each statement affects the state of M. In the case of the assembler machine, such updates are well known, albeit tedious, because they require a high-level specification of what assembler instructions should correspond to the source language. In the case of the timing machine, a cost model for statements has to be specified.

While tedious, such definitions are necessary for formal analysis of low-level security issues. For instance, work on detecting timing and cache-based side-channel attacks, already uses such models.

C. Analysis of Dead Store Elimination

In this section, we analyze dead store elimination using the source machine and a simplification of the assembler machine. We show that the observational equivalence game can be used to identify the persistent state violation caused by dead store elimination. Our contribution is to highlight that though more precise machine models are required to reason about security, new proof techniques are not. The gap between correctness and security is not due to a limitation of the proof technique used but due to the model used.

The program on the left below contains a variable x. Dead-store elimination removes the assignment to x to produces the code $\mathit{trans}(P)$ on the right.

```
// P
int inc(int a){
  int x = a+1;
  return a+1;
}
```

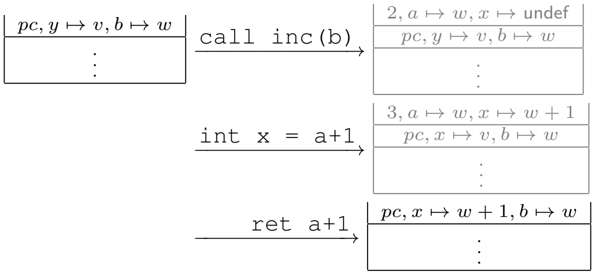
Listing 11. Program 1

```
// trans(P)
int inc(int a){

   return a+1;
}
```

Listing 12. Program 2

Assume that the set Obs of observations consists of values of program variables that are arguments or return values of procedure calls. Such observations abstract away from internal computations and the program counter. Let vis be true at the state before a call or after a return and false otherwise. Every call-site for the procedures above must be of the form y = inc(b). The transitions in $[\![P]\!]_{\mathit{Src}}$ of the program P in the source machine Src, for the call y = inc(b) are below, with the program counter of the caller written as pc.


$pc, y \mapsto v, b \mapsto w$
call inc(b)
$2, a \mapsto w, x \mapsto$ undef
$pc, y \mapsto v, b \mapsto w$
int x = a+1
$3, a \mapsto w, x \mapsto w + 1$
$pc, x \mapsto v, b \mapsto w$
ret a+1
$pc, x \mapsto w + 1, b \mapsto w$

The states in gray exist in $[\![P]\!]_{\mathit{Src}}$ but are not visible to the analyst in the game. The state reached after **int** x=a+1 does not exist in the transformed program. In these two programs, states affected by the optimization are not visible to the analyst. A winning strategy for the writer is to choose for every visible, transformed state a visible state with the same local environment. Such a choice is guaranteed to exist, though in general, a translation between program counter values may be required to derive it. By Theorem 1, it follows that the two transition systems are observationally equivalent, so dead store elimination is sound *with respect to the source machine*.

We now consider what the semantics of the transformed code may look like in the assembler machine. Potential states of the original and transformed code after the program returns are shown below. To avoid depicting a memory model, we present the state with a similar structure to the source machine state.

$$\begin{array}{|c|}
2, a \mapsto w, x \mapsto w + 1 \\ \hline
\triangleright pc, y \mapsto w + 1, b \mapsto w \\ \hline
\vdots \\ \hline
\end{array}
\qquad
\begin{array}{|c|}
2, a \mapsto w \\ \hline
\triangleright pc, y \mapsto w + 1, b \mapsto w \\ \hline
\vdots \\ \hline
\end{array}$$

The symbol $\triangleright$ indicates the top of the stack and text in gray depicts the local variables of `inc(b)`, which are not accessible but persist in memory. Consider a game played where the persistent, but non-accessible part of the state is observable. The analyst has a winning strategy by choosing an execution leading to the state on the right above. There is no corresponding state in code before compilation, so the analyst can show that the persistent state before and after optimization differ.

This analysis shows that dead store elimination is correct for the source machine but not for the assembler machine if one assumes that the contents of memory are visible. Similarly, undefinedness violations can be analyzed by making undefined values observable in the correctness proof. Side channel attacks can be analysed using a timed machine. A state of a timed machine is of the form (t, s) where s is a state and t is a timestamp. The set of observations consists of timestamps and the visible part of the state. The goal of the analyst is to find an execution whose timing after compilation does not correspond to the timing of an execution before compilation.

VI. Open Problems and Future Work

The main question that now arises is how one can further understand, detect, and mitigate the consequences of the correctness-security gap. We identify three families of open problems our research raises.

A. Deeper Understanding of the Correctness-Security Gap

1) The landscape of the correctness-security gap: One direction for research is to comprehensively identify the security impact of known compiler optimizations, especially widely used ones. It would be useful both for understanding and designing detection mechanisms to identify categories of security violations these optimizations belong to.

2) Generalized compiler correctness proofs: There are currently sophisticated techniques available for reasoning about compiler correctness. To incorporate reasoning about security, we believe compiler correctness proofs should be generalized to compiler correctness with respect to an abstract machine and an attacker. A research goal here is to identify general, reusable, automation-friendly proof principles applicable to families of machines and attackers. Developing such a framework requires research developments in a few areas listed below.

3) Abstract machine models: A large body of work in programming language semantics can be understood as developing representations of source machines. This includes interpreters based on language specifications, formal, executable semantics, and logical encodings of language semantics, all of which define some notion of a source machine. To reason about the low-level behaviour of code requires abstract machines at different levels of abstraction that are implemented as interpreters or in theorem provers.

4) Language semantics with respect to a machine: A major challenge for using abstract machines is defining the semantics of a high-level programming language with respect to a low-level machine. Such a semantics would, in effect, incorporate a formal notion of a compiler. Techniques for compositionally defining semantics with respect to a machine would simplify the construction of abstract machines and generalized correctness proofs.

5) Attacker models: Each abstract machine supports notions of observation, which define the channels an attacker has access to. There are multiple attackers one can consider for a given set of channels. A research goal here is to identify practically occurring attacker models and channels they have access to. Similar to abstract machine models, attacker models can also be execution environments or logical representations in theorem provers.

B. Detection of Correctness-Security Violations

The goal of a tool for detecting differences in security-critical behaviour before and after an optimizing transformation is fundamentally different from that of bug finding or static analysis tools.

1) Testing tools: A testing tool for this problem would combine elements of compiler testing tools with litmus tests for weak memory models. One approach testing is to execute two abstract machines side by side. The first machine can include the full set of optimizations applicable and the second machine performs no optimizations but interprets the source code directly. Before every transition between code regions of different security privilege, a memory snapshot could be taken. These snapshots can be compared after projecting out the visible parts of the state to identify differences observable by an attacker.

A second direction is to develop compilers for executable abstract machines, execute code in these machines, and compare memory snapshots taken at trust boundaries. These compilers should model the optimizations that will be applied in practice. A third direction is to construct test suites of small pieces of code that represent security intent and use techniques like those sketched above to detect correctness-security violations. Existing CWEs and examples in this paper are at the level of complexity for such tests. Such research has a natural extension to synthesis of test suites given a specification of a machine, attacker and optimization.

2) Correctness-Security Violation Detectors: Bug-finding tools typically incorporate a notion of a bug and search for occurrences of that bug. Static tools for detecting correctness-security violations need to incorporate a notion of observational equivalence, a family of applicable optimizations, and a model of optimization sites and trust boundaries. The goal of static tools would be to identify code patterns that represent security intent and determine if this intent is violated by an optimization. To avoid the well-known issue of flooding a developer with false alarms, such tools would have to accurately model secure coding patterns and the details of optimizations.

A tool for detecting undefined behaviour has recently been developed [60]. Detection tools need not be static. They can be dynamic, use the compiler and combine static and dynamic techniques. A compiler can be tested for security violations by evaluating memory or timing side effects at the security boundaries.

C. Implementing Security-Preserving Compilation

The goal of security-preserving compilation, from our perspective, is to allow for developer-introduced security measures to coexist with compiler optimizations to enable code that is both secure and executes efficiently.

1) Developer Annotations: One mechanism for preserving security guarantees in source code is to use code annotations that inform the compiler that certain optimizations are disallowed in a region of code. Specific examples of such regions are (i) side-effect free computations where the memory side effects must not be different from the source computation (i.e., the compiler disables dead store elimination, inlining, code motion and disallows undefined behaviour in this section) and (ii) timing adhering computations, where the compiler must not apply any optimizations due to possible changes in the timing of the generated code (i.e., the compiler disables all optimizations in this region).

This idea can be implemented using developer provided annotations. The annotations (or additional language keywords) would allow the developer to demarcate to the compiler that certain regions of code are critical for security (or timing). For example, two useful keywords would be, `secure` and `lockstep`. The `secure` keyword would indicate that the compiler should not modify the way data resides in memory as this may affect security. The `lockstep` keyword would indicate that the timing of code is critical and the compiled code should execute in lock-step with source code. Both these keywords should allow for a notion of scope, so that (some) optimizations can be temporarily disabled in some local region of code.

2) Statically Generated Annotations: An alternative to a developer-guided defense against compilers is to use automated analyses to detect common instances of the correctness-security gap. We can draw an analogy here to tools for insertion of memory fences in concurrent programs to guarantee that optimizations designed for sequential consistency architectures is not applied to code that runs in a relaxed memory architecture. An automated tool can be used to highlight to the developer regions of code where an optimization might apply that introduces a correctness-security gap. This approach would also be similar to the automatic insertion of bounds-checks into programs to catch run-time violations the developer did not prepare for. Bugs in the correctness-security gap that could be prevented in this way are elimination of memory scrubbing and dissolution of implicitly assumed trust boundaries (such as stack frames).

It is generally undecidable if a computation or variable is security critical or not as these notions depend on the context of the computation. One approach to such detection is heuristic, where pattern matching on certain code idioms or detecting the use of certain libraries can signal security-critical code. Another approach is to pivot around a *security seed*, e.g., libraries for security-critical operations or requests that change the privilege with which code runs. Data-flow analysis can be used to identify code that manipulates or is affected by variables flowing through the security seed, and this analysis can be used to identify security-critical code.

VII. Related Work

The first proof of compiler correctness we are aware of is over 40 years old [36]. The compiler verification bibliography [16] summarises a large body of related work. Work in compiler correctness can be classified along a problem and a solution dimension. Problems range from proving correctness to discovering bugs. Solutions range from using existing techniques to developing new frameworks. We restrict our attention to recent work in this context that targets C/C++, security relevant issues, and automation.

Our work is inspired by verification based on formalized, operational semantics such as in CompCert [34]. The idea of an assembler machine was inspired by the executable semantics of [25]. Abstract machines have also been used for correctness and security analysis in [19], [35], [40].

Compiler correctness has been formalized in terms of the theory of abstract interpretation in [13]. The notion of visibility in our work derives from theirs but the bisimulation-based proofs are closer to [34]. There is debate in the literature on operational versus denotational approaches to compiler correctness. To quote [34] "*it is unclear yet whether such advanced semantic techniques can be profitably applied* *to low-level, untyped languages*." A similar question applies to our setting where the combinatorics of machine behaviour eclipses the algebraic abstraction of a high-level language.

A contrary opinion to the one above appears in [7], where Hoare logic is extended to Relational Hoare Logic (RHL) for compiler correctness proofs. Curiously, RHL doubles as a proof system for secure information flow, that is as powerful as a type system for non-interference [52]. The security issues we have identified in compilers can be viewed as information leaks from portions of the state that are not modelled by the standard semantics. An interesting question for future work is whether a typing discipline based on this system can defend a program against insecure optimizations.

A distinct class of formal techniques attempts to automate correctness proofs. Cobalt [32] and Rhodium [33] use domain-specific languages and techniques, and such techniques have been applied to reason about cryptographic primitives [44]. In [28], dead code elimination, constant folding and code motion were encoded as rewrite rules on control flow graphs conditioned by temporal logic formulae. Both these approaches apply to the IR semantics. A natural extension of our work is to adapt these systems for security-sensitive reasoning about compilers.

The practical impact of compiler optimizations on security is visible in CWE-14 [1] and CWE-733 [2] that identify the dead store problem. The implementation of OpenSSL explicitly uses assembly code that will not be modified by the compiler to defend against compiler optimizations introducing side-channel vulnerabilities. The goal of our work is to highlight the role of sound optimizations in introducing security vulnerabilities.

Fuzzers provide a practical approach to discovering compiler bugs [14], [11]. Compiler fuzzing is challenging due to the structured nature of inputs and the depth of the software pipeline. Once a bug is discovered, test case reduction tools are applied to minimize the size of inputs [49]. We believe these tools can be used to discover more instances of the correctness-security gap.

Despite folk awareness that compiler optimizations may affect security we have not seen the problem articulated in detail as we have done, especially for sound optimizations. Concurrent to our work, Wang et al. [60] presented a model of undefined behaviour and developed the STACK system, which detects patterns of undefined behavior in existing applications. The goal of their work was to find bugs arising from undefinedness optimizations, while ours was to highlight and formally analyze the correctness-security gap. Undefinedness violations are an instance of this gap. Another important difference is our proposal of abstract machines for extending current reasoning approaches to detect such problems in a formal context.

We are not aware of research on counter measures that protect source code against compiler optimizations. To counter side-channels, [38] relies on source to source transformations and usually exhibits high performance (5x) and memory (3x on stack space) overhead. These approaches are oblivious to security-critical regions and use a conservative approach. We argue that it is important to infer security-critical and timing-critical regions to enable full compiler optimizations for most of the code while avoiding optimizations in specific regions.

VIII. Conclusion

In this paper, we identified the correctness-security gap, which arises when a formally sound, correctly implemented optimization violates security guarantees in source code. We demonstrated with several case studies that this phenomenon manifests with well known optimizations implemented in all compilers. We believe this behaviour is surprising and unintuitive because discrepancies between source and compiled code are typically attributed to bugs in compilers. Our analysis reveals that standard correctness proofs are performed in a model that ignores the details used to construct an attack. As a step towards addressing the problem, we used a game-theoretic framework to identify how a correctness proof can overlook a security property. Moreover, we suggested extensions to existing correctness frameworks that can be used to model security issues.

There is much work to be done to understand how prevalent the problem is, and to develop countermeasures. Towards this end, we have identified several open problems raised by this work with respect to achieving understanding, detection of such bugs and mitigation of the consequences. These new problems open the door to a range of work in formal logic, architectural modelling, tool development, and language design. In all these areas, we believe it is possible to reuse existing advances in compiler correctness, vulnerability detection, and runtime monitoring while extending research in these areas in fundamentally new ways. We look forward to undertaking such research and collaborating with the research community in such efforts.

References

[1] "Cwe-14: Compiler removal of code to clear buffers," http://cwe.mitre.org/data/definitions/14.html, 2013.

[2] "Cwe-733: Compiler optimization removal or modification of security-critical code," http://cwe.mitre.org/data/definitions/733.html, 2013.

[3] "Gcc 4.2 release series: Changes, new features, and fixes," https://gcc.gnu.org/gcc-4.2/changes.html, Jun. 2014.

[4] S. V. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware," *Communications of the ACM*, vol. 53, no. 8, pp. 90–101, Aug. 2010.

[5] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

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

[7] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *Proc. of Principles of Programming Languages*. ACM Press, 2004, pp. 14–25.

[8] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to "weird machines" and theory of computation," in *;login:*. Usenix, 2011.

[9] B. B. Brumley and N. Tuveri, "Cache-timing attacks and shared contexts," in *2nd Int'l. Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 2011, pp. 233–242.

[10] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proc. of the 12th conference on USENIX Security Symposium - Volume 12*. USENIX Association, 2003.

[11] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. F. , E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proc. of Programming Language Design and Implementation*. ACM Press, 2013.

[12] J. Corbet, "Fun with NULL pointers, part 1," http://lwn.net/Articles/342330/, Jul. 2000.

[13] P. Cousot and R. Cousot, "Systematic design of program transformation frameworks by abstract interpretation," in *Proc. of Principles of Programming Languages*. ACM Press, 2002, pp. 178–190.

[14] P. Cuoq, B. Monate, A. Pacalet, V. Prevosto, J. Regehr, B. Yakobowski, and Y. Xuejun, "Testing static analyzers with randomly generated programs," in *Proc. of NASA Formal Methods*. Springer, 2012, pp. 120–125.

[15] "Cve-2009-1897," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897, Jun. 2009.

[16] M. A. Dave, "Compiler verification: a bibliography," *SIGSOFT Software Engineering Notes*, vol. 28, no. 6, pp. 2–2, Nov. 2003.

[17] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in *Proc. of Int'l Conf. on Software Engineering*. IEEE Computer Society Press, 2012, pp. 760–770.

[18] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proc. of the Intl. Conf. on Embedded Software*. ACM Press, 2008, pp. 255–264.

[19] C. Ellison and G. Rosu, "An executable formal semantics of C with applications," in *Proc. of Principles of Programming Languages*. ACM Press, 2012, pp. 533–544.

[20] GCC, "Options that control optimization," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, Apr. 2013.

[21] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill, "Programming for different memory consistency models," *Journal of Parallel and Distributed Computing*, vol. 15, pp. 399–407, 1992.

[22] T. Glaser, "Bug 30477 - integer overflow detection code optimised away, -fwrapv broken," https://gcc.gnu.org/bugzilla/show_bug.cgi?id=30477, Jan. 2007.

[23] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on AES to practice," in *Proc. of Security and Privacy*. IEEE Computer Society Press, 2011, pp. 490–505.

[24] C. Hathhorn, C. Ellison, and G. Roşu, "Defining the undefinedness of C," in *Proc. of Programming Language Design and Implementation*. ACM Press, 2015.

[25] W. A. H. Jr. and M. Kaufmann, "Towards a formal model of the X86 ISA," University of Texas at Austin, Tech. Rep. TR-12-07, May 2012.

[26] R. Kluin, "[patch v1] compiler: prevent dead store elimination," https://lkml.org/lkml/2010/2/27/273, Feb. 2010.

[27] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Int'l Cryptology Conference*, 1999, pp. 388–397.

[28] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen, "Compiler optimization correctness by temporal logic," *Higher Order and Symbolic Computation*, vol. 17, no. 3, pp. 173–206, 2004.

[29] C. Lattner, "What every C programmer should know about undefined behavior #1/3," http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html, May 2011.

[30] ——, "What every C programmer should know about undefined behavior #2/3," http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html, May 2011.

[31] ——, "What every C programmer should know about undefined behavior #3/3," http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html, May 2011.

[32] S. Lerner, T. Millstein, and C. Chambers, "Automatically proving the correctness of compiler optimizations," in *Proc. of Programming Language Design and Implementation*. ACM Press, 2003, pp. 220–231.

[33] S. Lerner, T. Millstein, E. Rice, and C. Chambers, "Automated soundness proofs for dataflow analyses and transformations via local rules," in *Proc. of Principles of Programming Languages*. ACM Press, 2005, pp. 364–377.

[34] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *Proc. of Principles of Programming Languages*. ACM Press, 2006, pp. 42–54.

[35] X. Leroy and S. Blazy, "Formal verification of a C-like memory model and its uses for verifying program transformations," *J. of Automated Reasoning*, vol. 41, no. 1, pp. 1–31, Jul. 2008.

[36] J. McCarthy and J. A. Painter, "Correctness of a compiler for arithmetic expressions," in *Proc. of the Symposium in Applied Mathematics*, vol. 19. American Mathematical Society, 1967, pp. 33–41.

[37] R. Milner, “An algebraic definition of simulation between programs,” in *Int’l. Joint Conference on Artificial Intelligence*, D. C. Cooper, Ed. William Kaufmann, 1971, pp. 481–489.

[38] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, “The program counter security model: automatic detection and removal of control-flow side channel attacks,” in *Proc. of the 8th Int’l. conference on Information Security and Cryptology*, ser. ICISC’05. Springer, 2006, pp. 156–168. [Online]. Available: http://dx.doi.org/10.1007/11734727_14

[39] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[40] M. Norrish, “C formalised in HOL,” Ph.D. dissertation, University of Cambridge, 1998.

[41] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: the case of AES,” *IACR Cryptology ePrint Archive*, vol. 2005, p. 271, 2005.

[42] J. A. Painter, “Semantic correctness of a compiler for an Algol-like language,” Stanford University, DTIC Document ADA003619, 1967.

[43] C. Percival, “Cache missing for fun and profit,” in *Proc. of BSDCan 2005*, 2005.

[44] L. Pike, M. Shields, and J. Matthews, “A verifying core for a cryptographic language compiler,” in *Proc. of the workshop on the ACL2 theorem prover and its applications*. ACM Press, 2006, pp. 1–10.

[45] J. Regehr, “A guide to undefined behavior in C and C++, part 1,” http://blog.regehr.org/archives/213, accessed Apr. 2013, Jul. 2010.

[46] ——, “A guide to undefined behavior in C and C++, part 2,” http://blog.regehr.org/archives/226, accessed Apr. 2013, Jul. 2010.

[47] ——, “A guide to undefined behavior in C and C++, part 3,” http://blog.regehr.org/archives/232, accessed Apr. 2013, Jul. 2010.

[48] ——, “Its time to get serious about exploiting undefined behavior,” http://blog.regehr.org/archives/761, accessed Apr. 2013, Jul. 2012.

[49] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, “Test-case reduction for C compiler bugs,” in *Proc. of Programming Language Design and Implementation*. ACM Press, 2012, pp. 335–346.

[50] D. Sangiorgi, “On the origins of bisimulation and coinduction,” *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 4, pp. 15:1–15:41, May 2009.

[51] R. Sastry, “Bug 14287 - ext4: fixpoint divide exception at ext4_fill_super,” https://bugzilla.kernel.org/show_bug.cgi?id=14287, Oct. 2009.

[52] G. Smith and D. Volpano, “Secure information flow in a multi-threaded imperative language,” in *Proc. of Principles of Programming Languages*. ACM Press, 1998, pp. 355–364.

[53] C. Stirling, “Bisimulation, modal logic and model checking games,” *Logic Journal of the IGPL*, vol. 7, no. 1, pp. 103–124, 1999.

[54] M. Stump, “[patch] C undefined behavior fix,” https://gcc.gnu.org/ml/gcc/2002-01/msg00518.html, Jan. 2002.

[55] K. Thompson, “Reflections on trusting trust,” *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984.

[56] L. Torvalds, “Linux bug in cfs,” https://lkml.org/lkml/2007/5/7/213, May 2007.

[57] J. Vanegue, “The weird machines in proof-carrying code,” in *IEEE Security and Privacy Workshop on Language-Theoretic Security*. IEEE Computer Society Press, 2014.

[58] F. von Leitner, “Bug 30475 - assert(int+100 > int) optimized away,” https://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475, Jan. 2007.

[59] J. D. Wagner, “Bug 8537 - optimizer removes code necessary for security,” https://gcc.gnu.org/bugzilla/show_bug.cgi?id=8537, Nov. 2011.

[60] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, “Towards optimization-safe systems: Analyzing the impact of undefined behavior,” in *Proc. of Symposium on Operating Systems Principles*. ACM Press, 2013.

[61] F. Weimer, “Bug 19351 - [dr 624] operator new[] can return heap blocks which are too small,” https://gcc.gnu.org/bugzilla/show_bug.cgi?id=19351, Jan. 2005.

[62] N. Zeldovich, “Undefined behavior in google chrome,” https://codereview.chromium.org/12079010, Jan. 2013.

[63] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-VM side channels and their use to extract private keys,” in *Proc. of Conference on Computer and communications security*. ACM Press, 2012, pp. 305–316.