Enhancement And Deterioration of Software-Security Due To Compiler Extensions And Optimization

Anony, mous
Chair of Communication and Distributed Systems
RWTH Aachen University
anony.mous@rwth-aachen.de

***Abstract*—A compiler, apart from compiling code, can have other capabilities such as optimizing the resulting executable. The optimized executable requires less memory space or has an improved run-time. It is also possible to extent the compiler with certain security enhancing extensions.**
We discuss how formally correct compiler optimizations violate security guarantees in the source code. Thus examples of code that have been designed to defend against certain classes of attacks are presented alongside with how the compiler optimization weakens or eliminates these defences. Three different classes of security weaknesses that are introduced by compiler optimization are elucidated: 1) information leaks through persistent state 2) elimination of security-relevant code due to undefined behaviour 3) introduction of side channels. Some possible solutions to these compiler-introduced vulnerabilities are presented.
Furthermore, we discuss how compilers can contribute to software-security, by presenting some compiler extensions making it more difficult to exploit buffer overflow vulnerabilities on the stack. The effectiveness and the performance penalty of these methods will be analyzed and elucidated.

I. Introduction

A compiler translates source code of one programming language into another. In the following we will consider C compilers, which translate source code written in the C language to assembly code. Other than the C language which is independent of the computer architecture, the assembly code is specific for to a particular machine architecture [1].
Security-critical code is often reviewed and audited several times, and sometimes even formally tested. The compiler may pose a weak link between the heavily audited code and the compiler optimized binary [2]. The optimization performed by the compiler could render the security measures, taken by the programmers, useless. A lot of work was dedicated to the correctness of compilers and optimization [3] [4]. The question whether to trust the compiler or not is not a new subject and has been discussed by several researchers [5] [6] [7].
We discuss how compilers which have been proven to be formally correct can violate security precautions taken by the programmer. Dead store elimination, is a well known example of a security violating compiler optimization. Consider the code in Listing 1. Since the variable `key` is not being read after it has been assigned to zero, `key` is a dead store. When using GCC compiler optimization, dead store elimination is performed. This means that the assignment of the variable key to zero is optimized away. This saves assembler instructions and will result in a more efficient binary. The optimization has been proven formally correct with respect to `C` by [8] and [9]. Due to the optimization of the memory scrub, the value of the secret key persists in memory. Now an attacker, exploiting another vulnerability in the program, might be able to read the value of the secret key that persists in the memory. Even though the programmer took precautions to prevent such an attack, the compiler optimization still made it possible to retrieve the key, persisting in the memory. Listing 1 is an example describing the gap between the security precautions taken by the developer, and the formally correct compiler. In this case the programmer took the security precautions, and the compiler deleted that precaution and thus inserted a security hole. Considering code which is less security critical, the programmer and the compiler take the opposite roles. The developer is inserting the vulnerabilities, and the compiler has means to prevent an attacker from exploiting them. For example, the programmer uses the *get()* function in a program to read user data, even thought it may lead to a buffer overflow vulnerability (Explained in Section III-A). Then the compiler might be able to detect the buffer overflow and mitigate a possible attack. There are several compiler extensions which aim to prevent certain vulnerabilities from being exploited.

```
int crypt(){
  int key = 0x1234; //read the key
  //...do some operations with the key
  key = 0x0000; //scrub memory
}
```

Listing 1. In the function crypt, the developer made sure to erase the value of the key after its use. However the variable key is a dead store. Thus the precaution of memory scrubbing could be eliminated as a consequence of the compiler optimization technique *dead store elimination*. (adapted from [2])

A. Outline

In Section II the *correctness-security gap* is introduced. It arises when the compiler optimization preserves the functionality, but violates security measures taken by the developer. Several examples of how compiler optimizations weaken the security of the program are elucidated. At the end of the

section, some possible solutions to the *correctness-security gap* are presented. In Section II-D3 it is explained, how buffer overflows are working, and how an attacker can use them to take control over the program. Then it is discussed, how the compiler can prevent an attacker from exploiting these vulnerabilities, and taking control over the program's behaviour. The compiler can, for example, perform additional boundary checks or insert a *canary word* on the stack, or detect a buffer overflow. In the following we call the attacker Eve and the developer Bob.

II. Security-Oriented Analysis Of Compiler-Optimization

In this section, the *correctness-security gap* will be discussed. The term *correctness-security gap* was introduced by V. D'Silva, M. Payer and D. Song in the paper *The Correctness-Security Gap in Compiler Optimization* [2]. The researchers from Google found that compiler optimization can violate security guarantees made by the developer. Furthermore did they discuss possible solutions to that problem. Thus the following section is based on their work. A lot of work was dedicated towards the correctness of compilers and their optimization [3] [4] [9] [10]. Testing and formal analysis tools were developed to discover incorrect optimization and compiler bugs, e.g. the tool CSmith [11]. New mechanisms were developed to adapt existing optimization techniques to new architectures [12].
Similarly, we discuss, how the compiler produces incorrect results by violating security guarantees made in the source code.
A naive solution for the *correctness-security gap* problem would be to not use compiler optimization. That approach would not lead to a satisfiable result as there would be a severe performance overhead. Another solution would be to use the *volatile* keyword. This would require the programmers to use the keyword correctly, and the compilers to compile it correctly. Whereas the latter is not always the case according to [13]. In the following we elucidate three different classes of security weaknesses that are introduced by compiler optimization, namely 1) information leaks through persistent state 2) elimination of security-relevant code due to undefined behaviour 3) introduction of side channels. For each of these classes, we introduce a code example which was designed to defend against these classes of attacks, and a compiler optimization which weakens or eliminates that precaution. The examples will demonstrate the difference between how the programmer (Bob) expects the executable to behave and how it actually behaves. [2]

A. Information Leaks In Persistent State

Persistent state refers to characteristics of a thread or function that outlives the function or thread that created it. Specifically a persistent state violation is triggered when data remains longer in the memory than the programmer intended. A compiler optimization, altering code which processes security critical data, can cause information to persist in memory. Thus triggering a persistent state violation. In the following, three optimization techniques are presented, which could trigger information leaks in persistent state, namely *dead store elimination*, *function call inlining*, and *code motion*.

```
char *getPWHash() {
  long i; char pwd[64];
  char *sha2 = (char *)malloc(41);
  // read password
  fgets(pwd, sizeof(pwd), stdin);
  // calculate sha2 of password
  ...
  // Alternative (A)
  memset(pwd, 0, sizeof(pwd));
  // Alternative (B)
  for (i=0; i<sizeof(pwd); ++i)
    pwd[i]=0; // (**)
  return sha2;
}
```

Listing 2. The function `getPWHash()` calculates the hash of the password entered by the user. To make sure nobody can read the user's secret after the function call, Bob overwrites it with zeros. Thus Bob can use either alternative (A) or alternative (B). However dead store elimination will optimize both alternatives away rendering the code less secure than intended. (taken from [2])

1) Dead Store Elimination: A dead store is a local variable which is assigned a value, even though the variable will not be read in subsequent instructions. Dead store elimination thus eliminates assignments to variables, that are not being read in subsequent program statements. The optimization technique leads thus to an improved memory and time usage, especially if the dead store on a hot path.
Operations that scrub memory of sensitive data, e.g., erasing a cryptographic key after its usage, can lead to dead stores. Eliminating these dead stores will then lead to security deterioration of the program. Dead store elimination is a sound optimization, that preserves the functionality of the program. However the optimization technique affects parts of the security relevant program state.
Certain security critical variables, such as keys, should not remain longer in memory as needed. Setting the keys to all zeros when they are not needed anymore is one way to do it. Listing 2 presents a functions which calculates the hash of a password. Two options are presented how the password can be overwritten after its use. Option (A) uses the `memset()` function, and Option (B) uses a loop to set the buffer, storing the password, to all zeros. Both methods are prone to dead store elimination. Both memory scrubbing alternatives may be be optimized away by the compiler, eliminating the security precaution taken by Bob. Microsoft libraries are providing a function called `SecureZeroMemory()` [14], making sure that the memory scrubbing is not optimized away by the compiler.

2) Function Call Inlining: When compiler optimization is turned on, or if a function is marked with the keyword

```
int main(int argc, char* argv[]){
  char buf[100];
  char hash[256];
  //copy user input into buffer
  strcpy(buf, argv[1]);
  //Gets lnlined.
  hash = getPWHash();

  //format string vulnerability
  printf(buf);

}
```

Listing 3. The following program depicts such a vulnerability introduced by inlining. The `getPWHash()` function, containing some secret, gets inlined. Due to the format string vulnerability [16], Eve is able to read from the stack, allowing him to read the secret. (taken from [2]

`inline`, the compiler can decide to inline a function into another. This means that the body of the called function is being inserted directly into the callers function. The stack frames of caller and callee functions are merged. This saves the cost of creating a new stack frame, including return address, saved frame pointer, and the jumps of the instruction pointer. The time and space overhead of calling a function is eliminated but the size of the calling function's body increases. Thus the execution of an inlined function is as fast as if the function code would have been inserted directly [15]. If a security critical function gets inlined, the content of that function may have a longer lifetime as intended by Bob. The inlined function's stack frame will now be alive for the lifetime of the callers function. Consider the scenario in Listing 3. Bob thinks that the content of the stack frame during the execution of `getPWHash()` is secure. When the compiler inlines that function, the local variables of `getPWHash()`, which contain sensitive information and lived in their own stack frame, will now be alive for the lifetime of the main function.

Due to the unintended lifetime extension of the local variables of the `getPWHash()` function, sensitive information may be extracted from exploiting, e.g., a format string vulnerability [16] in the main function. An format string vulnerability can occur when the `printf()` function evaluates the format strings in e.g. a buffer controlled by the attacker. This could allow the attacker to read from the stack. Bob uses function boundaries as trust-separated domains. Inlining a function may violate the boundaries of these domains. A naive solution to that problem could be to delete the upper part of the stack after the inlined function has finished executing to enforce the same lifetime to the local variables as without inlineing.

3) Code Motion: Code motion allows the compiler to move certain code without changing the semantics of a program [2]. loop-invariant code can be moved outside of the loop to eliminate redundant computation. Loop-invariant code will thus be moved before or after the loop. Code motion can introduce a persistent state vulnerability. Consider the scenario in Listing 4. If the compiler decides that the if-statement is

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

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

Listing 4. Code which gets vulnerable to persistent state through code motion.(taken from [2])

part of a hot path, which means that the if-branch is executed often, then the code may be transformed such that the variable `secret` is assigned per default. If the `priv` flag is set to true, it is made sure that the variable `secret` is assigned in a secure environment. After the code motion, the variable `secret` is always assigned to the secret value, even thought the user does not necessarily has the required privileges.

Similar to inlining, is code motion changing the layout of the stack frames and the lifetime of the variables.

B. Elimination of Security-Relevant Code due to Undefined Behaviour

In the C language there is a specific set of rules defining the language. These rules are standardized by the International Organization for Standardization (ISO) [17]. These rules are referred as the *C standard revision* or simply `C standard` [18] [19].

The standardization of these rules allow the compilation of the same code on different architectures with any C compiler without any trouble. When breaking the rules of the C standard, it is possible to evoke undefined behaviour [20]. The developers of C wanted it to be a very efficient language. Thus the programmer is responsible to follow the specific rules of the C standard and avoid undefined behaviour. Safe languages like Java avoid undefined behaviour. Java is willing to sacrifice performance to have safe and reproducible behaviour across architectures [21] [22].

An example of undefined behaviour is the use of an uninitialized variable in C. Having the requirement of setting every variable to be initialized with zero, (as in Java) deteriorates performance. The overhead of initializing scalar variables with zero does not have a significant impact on performance. However stack arrays and dynamically allocated memory would require to overwrite the storage, and can be quite costly [23]. Any undefined behaviour gives licence to the implementation to produce arbitrary code, e.g., to format your hard drive. According to [21], compilers may assume that instructions, following a statement that triggers undefined behaviour, can be deleted. Undefined behaviour is thus a double-edged sword. On one-hand is undefined behaviour increases performance, e.g., by not having the requirement of setting every variable to be initialized with zero. On the other hand it might introduce

```
void process_something(int size) {
  // Catch integer overflow.
  if (size > size+1)
    abort();
  ...
  // Error checking from this code elided.
  char *string = malloc(size+1);
  read(fd, string, size);
  string[size] = 0;
  do_something(string);
  free(string);
}
```

Listing 5. Code which triggers undefined behaviour due to an integer overflow. (taken from [22])

security holes. To illustrate the point, consider the Listing 5 taken from [22]. The code is designed to catch an integer overflow. The codes makes sure that the allocated memory region is big enough to hold the data read from the file (the null byte needs to be added). When the variable `size` has its maximal value, line 2 `if (size > size+1)` leads to undefined behaviour due to the integer overflow. The compiler thus deletes the if clause and the `abort()` function call.

C. Introduction of Physical Side Channels

A physical side channel attack, is an attack which is based on information and measurements of the physical system, rather than the algorithm itself. Thus leaking information about the internal state of the system to an external attacker [24]. Information like timing, power consumption, sound or electromagnetic leaks can be used to get additional insights. Timing attacks are based on the time spent on different computations and were introduced 1996 by Kocher [25]. In cryptographic functions there are often secret parameters involved. If the processor cycles are measured and compared, it is possible to figure out the time complexity of certain instructions [2]. If Eve has access to enough information about the exact implementation of these cryptographic algorithms, she could be able to recover complete keys [24]. If Bob is aware of a possible timing attack he can develop a specific timing cost model, rendering timing attacks unfeasible. These timing cost models ensure, that the number of instructions is the same regardless of the cryptographic algorithm's parameters. If the compiler optimizes certain program paths, the guarantee that the algorithm has the same number of instructions, independent of the parameters, might become obsolete [2]. The problem could be solved by writing these security critical sections of the code in inline assembler. Compilers typically do not modify such code. The benefits of readability, maintainability and portability of high level languages are lost. Moreover, optimizations can sill enable side channels based on cache hierarchies [26] or virtual machines [27].

In the following three different compiler optimization techniques, which can introduce side channel vulnerabilities, are considered.

```
int crypt(int *k){
 int key = 0;
 if(k[0]==0x61){
  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 6
BEFORE CSE

```
int crypt(int *k){
 int key = 0;
 if(k[0]==0x61){
  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 7
AFTER CSE

1) Common Subexpression Elimination: One compiler optimization technique, that can introduce side channel vulnerabilities, is called common subexpression elimination (CSE) [2]. A common subexpression, is an expression that occurs in at least two different statements. During CSE, the compiler searches the code for multiple instances of the same expression. If the variables in these expressions have the same value, the expression gets pre-calculated. That pre-calculated value will then replace all occurrences of the common subexpression. Consider the following example where $x + y$ is a common subexpression:

$$c = x + y + z;$$
$$z = x + y - 2z;$$

If the variables x and y have the same value in both assignments, a new variable s can be introduced, and the assignments can be rewritten to:

$$s = x + y$$
$$c = s + z;$$
$$z = s - 2z;$$

This optimization can also easily be implemented by the programmer himself. However this gets more complicated if the subexpressions are getting more complex. Common subexpression elimination affects the timing of instructions. Thus the timing guarantees preventing timing attacks, made by the developer, might become obsolete.

In Listings 6 and 7, which are taken from [2], it is illustrated how CSE can affect the timing of the code. Before the optimization, the if and the else branch were taking the same amount of time. Each branch performs a multiplication and addition operations three times. However after the CSE, the else branch performs only one multiplication and one addition, before doing another multiplication. If Eve is able to measure the timing difference, she can figure out which branch is taken.

2) Strength Reduction: Operator strength reduction (SR) is a compiler optimization technique which improves compiler generated code. The optimization technique reformulates costly (strong) expressions into cheaper (weaker) ones [28].

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

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

Listing 8
BEFORE SR

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

Listing 9
AFTER SR

Replacing $x * 2$ with $x + x$ or $x << 1$ would be a simple example of strength reduction. A stronger optimization scenario would be to replace certain multiplications with equivalent additions in a loop. This case occurs systematically in loops, where array address calculation takes place [28]. The function `crypt(int k)` (taken from [2]) in Listing 8, and Listing 9, takes an integer argument and encrypts it. Bob made sure that in both if/else branches the same operations are performed, namely one multiplication, and one addition operation. After the SR, the operations in the `if` branch are replaced by a bit-shift, an addition, and a subtraction. In the `else` branch the operations are replaced by a constant. After the optimization, the time difference of the different branches is more significant than before. An attacker, which has access to the timing channel, can now more clearly distinguish between both paths. Thus SR increases the amount of information leaked via timing side-channels because it alters the computation time of certain branches.

3) Peephole Optimization: Peephole optimization is a technique to improve the performance of the resulting binary by examining a short sequence of instructions (called the peephole or window). Whenever possible, these peepholes are replaced by shorter of faster instruction sequences [10]. Peephole optimization is one of the last phases performed by compilers before emitting code [2]. The Peephole is a small window, which is moving on the target program. Peephole optimization is able to eliminate redundant instructions, or unreachable code. It can furthermore simplify algebraic expressions. E.g., statements such as $x = x+0$ or $x = x*1$ can be eliminated [10]. Similar to CSE or SR, peephole optimization can alter the execution time of certain branches, and thus introducing timing based side channels.

All the optimizations used in this section were shown to be formally correct [2].

D. Possible Solution To The Correctness-Security Gap

Until now discussed several cases, where the compiler destroyed the security guarantees taken by the developer. The question arises how these security violations can be mitigated.

1) General compiler correctness proof: The compiler correctness proof can be extended, such that the gap between correctness and security is closed. An intuitive explanation why current compiler correctness proofs do not preserve certain security guarantees, is that they do not include the state of the underlying machine, but only the state of the program. Thus the state of the execution environment should be modeled in a first step. This measure is although not sufficient. To detect side channel attacks, an accurate model of state transitions is required [2].

2) Detecting Correctness-Security Violations: A tool that analyzes the correctness-security violations could be implemented. One way that tool could work, would be to execute two abstract machines side by side. One machine would execute the program with full optimization, and the other machine would do no optimizations at all. Every time the program makes a transition to a region with a different security level, a memory snapshot would taken. The parts of the snapshots, which could also be visible to an attacker, could be compared to identify possible differences [2].

Another way a tool could find correctness-security violations, could work on a pattern-recognition basis. The static tool could identify patterns, that represent security relevant code included by the developer. Once the pattern is identified, the tool could analyze whether the optimization would delete that code or not. The problem with the tool could be a high false positive alarm rate, if the patterns do not exactly match secure coding patterns. [2].
A related tool has already been developed by Wang in 2013 [29]. The tool detects occurrences of undefined behaviour in a program. This tool is although not able to find every batch of instructions, triggering undefined behaviour. In the tests of [29] it was able to detect 7 out of 10 occurrences. The extension of that tool could lead to the desired goal.

3) Security-Preserving Compilation: As analyzed in [2], the standard compiler correctness proofs are performed in a model, that ignores the details used to construct an attack. A desirable goal is the coexistence of optimization and security guarantees implemented by the developer. This way the program will benefit from compiler optimization, and still includes the security guarantees made by the developer.

One method to implement this feature, could be developer annotations. With specific code annotations, the developer could notify the compiler not to use optimization in that region of the code. This way the developer could, for example, indicate to the compiler to omit optimization during the calculation of the hash of a password. This would avoid the user password to persist in memory, due to dead store elimination. The same holds for timing adhering computation, and side channel attacks. In [2] two possible keywords are introduced: `secure` and `lockstep`. The former indicated the compiler, not to modify the way data resides in memory. Whereas the latter signals, that the timing of the code is critical, and should not be optimized.
An alternative to a developer-guided defence would be, that the compiler would statically generate the annotations.

In this section, a possible compiler extension to mitigate the correctness-security gap was presented. In the past, compiler-

extensions have been designed to mitigate other types of attacks, e.g., buffer overflows attacks [30]. In the following sections, certain security enhancing compiler-extensions are presented.

III. Compiler-Extensions Preventing Buffer Overflow Vulnerabilities

Over the years, many techniques preventing buffer overflows have been developed. In 1997 Richary Jones and Paul Kelly developed a GCC patch which performed array bounds checking for C [31]. One year later the compiler extension StackGuard was introduced [30]. In 2000 PointGuard was first presented in a paper of Crispin Cowan [32]. In the following sections these compiler extensions will be presented and their effectiveness and impact on performance are shortly elucidated.

A. Buffer Overflow and Pointer Corruption

Buffer overflow vulnerabilities exists since the early days of programming, and still exists today. Internet worms, like the Morris worm in 1988 [33], are using buffer overflows to propagate. On $May\ 16^{th}\ 2017$ a buffer overflow vulnerability was reported in the voice over IP software Skype [34], that would allow an attacker to execute malicious code. A buffer overflow can occur in all kinds of programs requiring user input, and is one of the most common vulnerabilities. Buffer overflow vulnerabilities can lead to program crashes, remote machine access or to privilege escalation exploits. Privilege escalation means that a attacker, exploiting the buffer overflow vulnerability, has elevated rights, and in the worst case unlimited access to all resources [35].

A buffer overflow can occur, when the program does not check the size of the user input, which will be stored in a buffer array. If the user input is larger than the buffer array which was intended to store the input, the memory area adjacent to the array will be overwritten by the super-induced data. Overwriting the adjacent data is possible in languages like C and C++, because no boundary checking is performed. This means that it is not checked whether the data written to the buffer lies within the boundaries of the buffer. The bound-checking is not performed due to performance reasons (In Section III-B the performance penalty of bound-checking is elucidated) [32].

The goal of such an attack could be to overwrite a function's return address. The return address of a function, saves the address of the instruction which will be executed after the function returned. When overwriting that address, Eve can choose where the program will continue its execution, once the function returned. If Eve can manage to change the return address of the function to point at her own code, she will be able to execute arbitrary code. Eve can use this for, e.g., granting her a terminal with the attacked program's privileges [36].

Figure 1 shows the stack frame, after the function `foo` in Listing 10 is called. Assuming the variable `a` is under Eve's control, then she can manage to overwrite parts of the stack, e.g., the variable `local`, the saved frame pointer (SFP), and the return address (RET). Taking into account that even though the stack grows towards lower addresses, the buffer is growing towards higher addresses. Eve could write her attack code into the variable `a`, and overwrite the return address such that it points to `a`. Another possibility would be to call a function from loaded libraries, instead of executing code on the stack. This technique is referred to as return-to-libc attack. This has the advantage that an attacker can compromise a program, even if it does not allow code being executed on the stack [37].

The same technique can be used to corrupt pointer variables. This is done by overwriting a pointer or function pointer variable. This may lead to the execution of arbitrary code, or crashing the program.

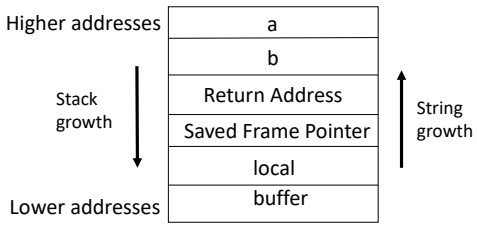

Fig. 1. The Figure shows how the stack frame of the function `foo`. The saved frame pointer, located between the local variables and the return address, saves the position of the stack pointer before the function call. (Adapted and redrawn from [30])

```
int foo(int b, char* a){
        int local;
        char buffer[100];

        strcpy(buffer, word);
        return b;
}
```

Listing 10. This listing illustrates code which is vulnerable to a buffer overflow, assuming that Eve is in control of the variable `a`. Thus by inputting a string longer than 100 characters, other parts of the stack will be overwritten.

B. Array Bound Checking

The array bound checking method, by Kelly and Jones [31], presents an approach to enforce array bounds and pointer checking in C language. This is done by checking if a pointer, after having applied an arithmetic operation, is still referring to the same array or struct (storage object). While other methods like StackGuard (Presented in Section III-C) do not detect every possible occurrence of a buffer overflow, is array bound checking completely stopping the buffer overflow vulnerability and attack. If it is not possible to overflow a buffer, it can not corrupt the adjacent variables and program state.

The code with array bounds checking is inter-operable with

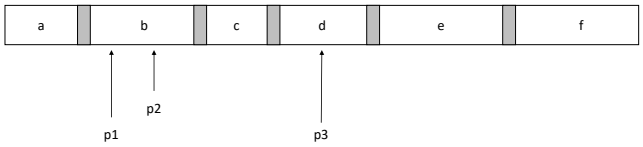

Fig. 2. The Figure shows an example of how the table of the storage objects (object table), arising from static, dynamic or automatic memory allocation, could look like. (Adapted and redrawn from [31])

non-checked code because the representation of the pointers remains unchanged. Thus the overhead of the bounds-checking can be reduced by only applying the technique to security critical modules.

The technique works by maintaining a table of all known valid storage objects. The table itself was stored in a splay tree. A mapping between pointers and the descriptors of the storage objects in the table is made. Whenever a pointer arithmetic is used, it is checked in the object table whether the pointer still refers to the intended storage object. An example for a table maintaining valid storage objects can be seen in Figure 2. Pointer operations are only permitted to take place within one object, and not between them. That means that operations on $p1, p2$ are permitted, but operations on $p2, p3$ and $p1, p3$ are not.

The performance cost of the method is severe. A pointer intensive program may experience a slowdown of 3000%. As shown in the experiments of [32] the throughput with the updated compiler led to a 12x slowdown of SSH. In 2012 the tool *Asan*, short for Address Sanatizer was developed [38]. *Asan* has similar functionality but the performance penalty is less severe, the largest slowdowns measured by [38] were by a factor of 2.67.

[38].

C. StackGuard

It is fairly easy to prevent individual buffer overflow vulnerabilities. This can be done by the programmer, who can include a simple range check before copying data into a buffer. It would be preferable to have a solution, which eliminates the buffer overflow vulnerabilities when compiling the code. Thus the security of the program would not totally rely on the programmer, but also on the compiler.

In this section, StackGuard is introduced. It is a compiler extension, which is helping to mitigate buffer overflow vulnerabilities with modest performance penalties [30]. StackGuard was released for GCC in 1997, and published in 1998 [32]. StackGuard was available from GCC 2.7.2.3 through 2.96. StackGuard was not built into GCC 3.x.

IBM implemented ProPolice [39], a buffer overflow protection which was based on StackGuard. The main change was the rearrangement of the variables on the stack, such that the $char$ buffers are always allocated at the bottom. Thus the rearranged buffers could not harm other local variables when overflown. ProPolice was implemented as a patch to

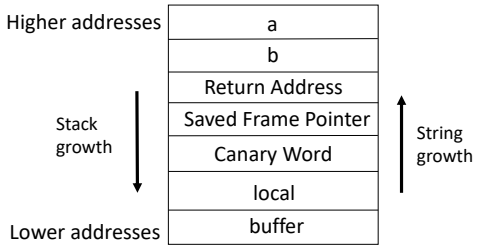

Fig. 3. The Figure shows how the stack frame of the function *foo* with the *canary word*. (Adapted and redrawn from [30])

GCC 3.x and re-implemented in GCC 4.1. In 2012, Google developed the stack protection further, by enhancing the balance between run-time efficiency and protection. Google's version is available since GCC version 4.9 [40]. For the reason that StackGuard is a compiler extensions, it suffices to recompile a program to integrate the security enhancement technique.

The StackGuard functionality is transparent to the program functions. As elucidated in Section III-A the attacker usually overwrites the return address of the stack frame. StackGuard is able to thwart a buffer overflow attack, by detecting the change of the return address before the function returns. To detect the change of the return address before the function returns, StackGuard places a so called *canary word* between the local variables and the saved frame pointer (thus also before the return address). The stack with the *canary word* is depicted in Figure 3. Before the function returns, it first checks the integrity of the *canary word*. If it is intact, the function returns, else the program terminates with an error. To prevent that Eve can guess the *canary word*, it is chosen at random for each function call. While it is not completely impossible for a dedicated attacker to guess the canary value, it is difficult (The attacker must be able to examine the memory image of the process).

When taking a look at the performance of the StackGuard approach, the canary mechanism imposes additional cost at two different points in a function call. Firstly pushing the canary on the stack, when the function gets called, which only imposes a small cost. Secondly check the integrity of the canary word, when the function returns, which imposes a moderate cost.

The following experiment aims to discover the overhead of the current canary-based stack protector in GCC 5.4.0. The experiment was reproduced from [30]. The experiment consisted of incrementing the variable i 2,000,000,000 times without any compiler optimization. The experiment was executed ten times and the arithmetical mean time of the ten executions was taken. The base case does no function calls and simply increments

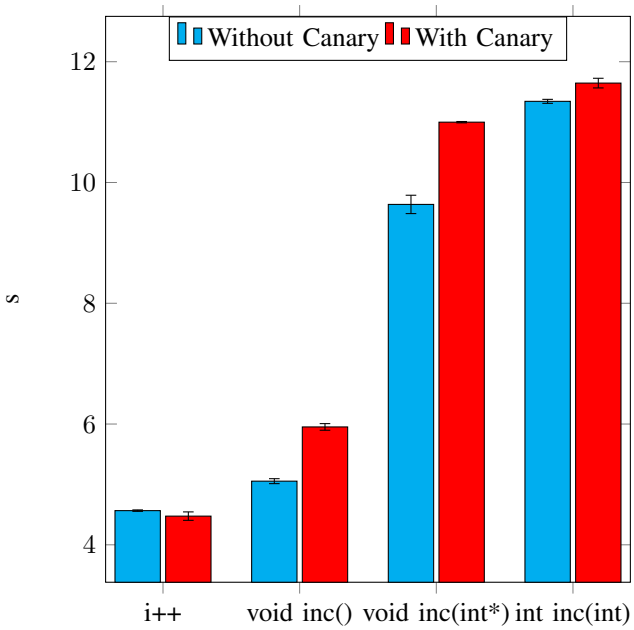

Fig. 4. In the plot there the different increment methods with their runtime and the confidence intervals with a confidence level of 95% are depicted. The red bar representing the run-time in seconds without the use of a canary word and the blue bars representing the run-time with canary word protection.

the variable i the $++$ operator. In the second case the function `void inc()` does the $i++$ operation on the global variable i. In the third case, the function `void inc(int*)` takes an integer variable and increments it as a side effect. In the fourth case, the function `int inc(int)` takes an integer argument and returns the value $+1$. In Figure III-C the run-times of the different scenarios are shown. The experiment was executed on an Intel Core i7-4510U with 2.0GHz and 16 GB of main memory and GCC version 5.4.0.

1) Defeating StackGuard: StackGuard is not able to mitigate every attack. If Eve is aware of the StackGuard protection she could be able to defeat it.

One method for defeating the StackGuard is to, e.g., overwrite the destination address of a `strcpy(dst, buf)` function [41]. If `dst` points to the address of the *return address*, `buf` is written into the *return address* without touching the canary word. This method is depicted in Figure 5.

D. PointGuard

PointGuard is a method for protecting programs against pointer corruption attacks [42]. This is done, by encrypting the pointer each time a pointer is initialized or modified, and decrypting it right before it is read. This can be done by choosing a predetermined key value, which can be a random or a specific value, and XOR it with the pointer. Encryption and decryption can also use another scheme, but it has to be reasonably fast.

When using no pointer protection, a function pointer can be corrupted to point at a malicious function. Eve can, e.g., exploit a buffer overflow vulnerability and overwrite a function

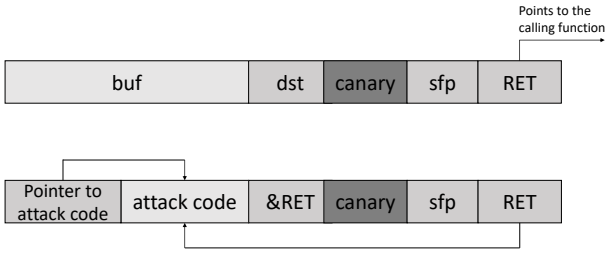

Fig. 5. The Figure shows how the canary can be defeated. The top element depicts how the stack looks like immediately after the creation of the stack frame. Eve now places the a pointer to the attack code and the attack code into the buffer. Furthermore she replaces `dst` with the address of the function's return address using a buffer overflow vulnerability. When the `stycpy(dst,buf)` function gets called, it writes the pointer to the attack code into the return address, because `dst` was overwritten with the address of the return address. Now when the function returns, the instruction pointer will be set to the attack code, and Eve is able to execute arbitrary code.

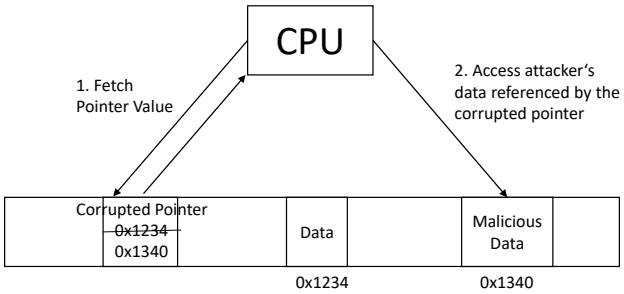

Fig. 6. The Figure shows how the corruption of a pointer can easily lead to controlling the program's behaviour by being able to execute arbitrary code. The benign pointer, pointing to the legitimate data at address 0x1234 gets overwritten. The pointer is now corrupted and points to malicious code at address 0x1340. (Redrawn from [42])

pointer. In Figure 6 a pointer corruption attack is illustrated. The PointGuard method of encrypting the pointers helps to mitigate any attack technique, which seeks to modify the program pointers (e.g. the return address or a function pointer). If a pointers gets modified, the compiler inserted code will decrypt it. Decrypting an unencrypted address will result in a random address. Thus the program will access a random memory location, instead of the memory location containing the attack code. This scenario, depicted in Figure 7, will most likely cause the program to crash and not allowing Eve to cause any particular behaviour.

1) Defeating PointGuard: PointGuard uses XOR encryption to encrypt the pointers, instead of a more complex non-linear operation. This is making it weak. Any bit that is not overwritten will decrypt correctly. When a few bits are modified in the cryptext, then the modification only affects these same bits in the plaintext after decryption. An attacker will now attempt to redirect the pointer towards a memory

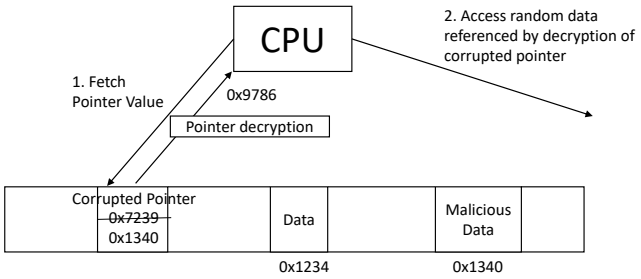

Fig. 7. The Figure shows how that corrupting a pointer does not lead to controlling the program's behaviour. Eve has changed the value of the pointer as before to 0x1340. When decrypting that value becomes a random value and does not match the malicious code's address anymore. The CPU tries to access a random memory location which will most likely result in a crash. a program crash is in this case a desired outcome because it prevents Eve taking over control of the victim program. (Redrawn from [42])

location, where the first few bytes remain the same. The remaining bits could be figured out, e.g., via brute force. This takes much less effort than brute forcing the whole number of bits of the address. A buffer overflow attack could be used to overwrite the last few bits on an little-endian architecture. [43].

IV. Conclusion

In the beginning, the gap between formally correct compiler optimizations and security were introduced. To prove that point, some examples of how compiler optimization techniques violated security guarantees made by the developer were shown. Formally correct proven compilers were optimizing away security relevant code, rendering it more vulnerable. Discrepancies between the source code and the compiled code are typically due to bugs in the compiler, rendering the compiler-introduced security violations surprising. Standard compiler correctness proofs do not consider any of the presented attack scenarios, as they do not include the state of the underlying machine. Several possible solutions to the correctness-security gap problem, which could be implemented, were presented.

On the contrary the compiler can also enhance program security. Compiler extensions like StackGuard, and PointGuard are able to detect, and mitigate buffer overflows on the stack.

Buffer overflow mitigation techniques always introduce a trade off between the effectiveness of the attack mitigation, and the run-time. Other than StackGuard and PointGuard, Array Bound Checking is harder to defeat, when trying to exploit a buffer overflow vulnerability. However there overhead rises drastically.

It can be concluded that, on one hand, compiler optimization are deteriorating the security of code while enhancing its memory, and time requirements. On the other hand, compiler extensions are enhancing the security of the program, but are always tied to a certain overhead. Future requirements for compilers comprise enhancing the efficiency and effectiveness of security enhancing compiler extensions, and building possibilities for secure optimizing compiler techniques.

References

[1] (February 2017) Definition of: compiler. [Online]. Available: https://www.pcmag.com/encyclopedia/term/40105/compiler

[2] V. D'Silva, M. Payer, and D. Song, "The Correctness-Security Gap in Compiler Optimization," in *Security and Privacy Workshops (SPW), 2015 IEEE*. IEEE, 2015, pp. 73–87.

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

[4] J. McCarthy and J. Painter, "Correctness of a Compiler for Arithmetic Expressions," *Mathematical aspects of computer science*, vol. 1, 1967.

[5] K. Thompson, "Reflections on Trusting Trust," *Communications of the ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[6] J. M. Boyle, R. D. Resler, and V. L. Winter, "Do You Trust Your Compiler?" *Computer*, no. 5, pp. 65–73, 1999.

[7] X. Leroy, "Formal Verification of a Realistic Compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[8] N. Benton, "Simple Relational Correctness Proofs for Static Analyses and Program Transformations," in *ACM SIGPLAN Notices*, vol. 39, no. 1. ACM, 2004, pp. 14–25.

[9] X. Leroy, "Formal Certification of a Compiler back-end or: Programming a Compiler with a Proof Assistant," in *ACM SIGPLAN Notices*, vol. 41, no. 1. ACM, 2006, pp. 42–54.

[10] (October 2017) "peephole optimization compiler design". [Online]. Available: http://freestudy9.com/peephole-optimization/

[11] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding And Understanding Bugs in C Compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[12] M. J. Voss and R. Eigemann, "High-level Adaptive Program Optimization with ADAPT," in *ACM SIGPLAN Notices*, vol. 36, no. 7. ACM, 2001, pp. 93–102.

[13] E. Eide and J. Regehr, "Volatiles Are Miscompiled, And What To Do About It," in *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008, pp. 255–264.

[14] "securezeromemory function". [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/aa366877(v=vs.85).aspxml

[15] J. Wolf, *C von A bis Z: das umfassende Handbuch;[das Lehr-und Nachschlagewerk; für Einsteiger, Umsteiger und Profis; zum aktuellen Standard C99; inkl. CD-ROM mit Openbooks und Referenzkarte mit wichtigen Befehlen]*. Galileo Press, 2011.

[16] T. Newsham, "Format String Attacks," 2000.

[17] "iso/iec 9899:1990, programming languages – c". [Online]. Available: https://www.iso.org/standard/17782.html

[18] J. Goll and M. Dausmann, *C als erste Programmiersprache: Mit den Konzepten von C11*. Springer-Verlag, 2014.

[19] B. W. Kernighan, A. T. Schreiner, E. Janich, and D. M. Ritchie, *Programmieren in C mit dem C-Reference Manual, in deutscher Sprache*. Hanser, 1990.

[20] D. M. Ritchie, "The Development Of The C Language," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 201–208, 1993.

[21] C. Lattner. (2011) "what every c programmer should know about undefined behaviour 1". [Online]. Available: http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

[22] ——. (2011) "what every c programmer should know about undefined behaviour 2". [Online]. Available: http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html

[23] ——. (2011) "what every c programmer should know about undefined behaviour 3". [Online]. Available: http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_21.html

[24] D. F. YongBin Zhou, "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing," *State Key Laboratory of Information Security*, vol. 1, 2006.

[25] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Annual International Cryptology Conference*. Springer, 1996, pp. 104–113.

[26] B. B. Brumley and N. Tuveri, "Cache-timing Attacks and Shared Contexts," in *Proceedings of the 2nd International Workshop on Constructive Side-Channel Analysis and Secure Design, COSADE*, 2011, pp. 233–242.

[27] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proceedings of*

the 2012 ACM conference on Computer and communications security. ACM, 2012, pp. 305–316.

[28] K. D. Cooper, L. T. Simpson, and C. A. Vick, "Operator Strength Reduction," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 5, pp. 603–625, 2001.

[29] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 260–275.

[30] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[31] R. W. Jones and P. H. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, no. 001. Linköping University Electronic Press, 1997, pp. 13–26.

[32] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2. IEEE, 2000, pp. 119–129.

[33] H. Orman, "The Morris worm: A fifteen-year perspective," *IEEE Security & Privacy*, vol. 99, no. 5, pp. 35–43, 2003.

[34] "The security expert benjamin kunz-mejri from security firm vulnerability lab discovered a remote zero-day stack buffer overflow vulnerability in skype." https://securityaffairs.co/wordpress/60507/hacking/skype-buffer-overflow.html, accessed: 2018-04-25.

[35] T.-c. Chiueh and F.-H. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks," in *Distributed Computing Systems, 2001. 21st International Conference on.* IEEE, 2001, pp. 409–417.

[36] J. Erickson, *Hacking: The Art of Exploitation*. No starch press, 2008.

[37] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the Expressiveness of Return-into-libc Attacks," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 121–141.

[38] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A Fast Address Sanity Checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[39] "Propolice: Gcc extension for protecting applications from stack-smashing attacks."

[40] "gcc, the gnu compiler collection". [Online]. Available: https://gcc.gnu.org/

[41] G. Richarte *et al.*, "Four Different Tricks to Bypass Stackshield and Stackguard Protection," *World Wide Web*, vol. 1, 2002.

[42] S. C. Cowan, S. R. Arnold, S. M. Beattie, and P. M. Wagle, "Pointguard: Method And System For Protecting Programs Against Pointer Corruption Attacks," Jan. 14 2014, uS Patent 8,631,248.

[43] S. Alexander, "Defeating Compiler-Level Buffer Overflow Protection," *The USENIX Magazine; login*, 2005.