Code Pointer Masking: Hardening Applications against Code Injection Attacks

Pieter Philippaerts[1], Yves Younan[1], Stijn Muylle[1], Frank Piessens[1], Sven Lachmund[2], and Thomas Walter[2]

[1] DistriNet Research Group
[2] DOCOMO Euro-Labs

Abstract. In this paper we present an efficient countermeasure against code injection attacks. Our countermeasure does not rely on secret values such as stack canaries and protects against attacks that are not addressed by state-of-the-art countermeasures of similar performance. By enforcing the correct semantics of code pointers, we thwart attacks that modify code pointers to divert the application's control flow. We have implemented a prototype of our solution in a C-compiler for Linux. The evaluation shows that the overhead of using our countermeasure is small and the security benefits are substantial.

1 Introduction

A major goal of an attacker is to gain control of the computer that is being attacked. This can be accomplished by performing a so-called *code injection attack*. In this attack, the attacker abuses a bug in an application in such a way that he can divert the control flow of the application to run binary code — known as *shellcode* — that the attacker injected in the application's memory space. The most basic code injection attack is a stack-based buffer overflow that overwrites the return address, but several other — more advanced — attack techniques have been developed, including heap based buffer overflows, indirect pointer overwrites, and others. All these attacks eventually overwrite a *code pointer*, i.e. a memory location that contains an address that the processor will jump to during program execution.

According to the NIST's National Vulnerability Database [1], 9.86% of the reported vulnerabilities are buffer overflows, second to only SQL injection attacks (16.54%) and XSS (14.37%). Although buffer overflows represent less than 10% of all attacks, they make up 17% of the vulnerabilities with a high severity rating.

Code injection attacks are often high-profile, as a number of large software companies can attest to. Apple has been fighting off hackers of the iPhone since it has first been exploited with a code injection vulnerability in one of the iPhone's libraries[3]. Google saw the security of its sandboxed browser *Chrome* breached[4]

[3] CVE-2006-3459.
[4] CVE-2008-6994.

because of a code injection attack. And an attack exploiting a code injection vulnerability in Microsoft's Internet Explorer[5] led to an international row between Google and the Chinese government. This clearly indicates that even with the current widely deployed countermeasures, code injection attacks are still a very important threat.

In this paper we present a new approach, called *Code Pointer Masking (CPM)*, for protecting against code injection attacks. CPM is very efficient and provides protection that is partly overlapping with but also complementary to the protection provided by existing efficient countermeasures.

By efficiently masking code pointers, CPM constrains the range of addresses that code pointers can point to. By setting these constraints in such a way that an attacker can never make the code pointer point to injected code, CPM prevents the attacker from taking over the computer.

In summary, the contributions of this paper are:

- It describes the design of a novel countermeasure against code injection attacks on C code.
- It reports on a prototype implementation for the ARM architecture that implements the full countermeasure. A second prototype for the Intel x86 architecture exists, but is not reported on in this paper because of page limit constraints and because it is still incomplete. It does, however, show that the concepts can be ported to different processor architectures.
- It shows by means of the SPEC CPU benchmarks that the countermeasure imposes an overhead of only a few percentage points and that it is compatible with existing large applications that exercise almost all corners of the C standard.
- It provides an evaluation of the security guarantees offered by the countermeasure, showing that the protection provided is complementary to existing countermeasures.

The paper is structured as follows: Section 2 briefly describes the technical details of a typical code injection attack. Section 3 discusses the design of our countermeasure, and Section 4 details the implementation aspects. Section 5 evaluates our countermeasure in terms of performance and security. Section 6 further discusses our countermeasure and explores the ongoing work. Section 7 discusses related work, and finally Section 8 presents our conclusions.

2 Background: Code Injection Countermeasures

Code injection attacks have been around for decades, and a lot of countermeasures have been developed to thwart them. Only a handful of these countermeasures have been deployed widely, because they succeed in raising the bar for the attacker at only a small (or no) performance cost. This section gives an overview of these countermeasures.

[5] CVE-2010-0249.

Stack Canaries try to defeat stack-based buffer overflows by introducing a secret random value, called a *canary*, on the stack, right before the return address. When an attacker overwrites a return address with a stack-based buffer overflow, he will also have to overwrite the canary that is placed between the buffer and the return address. When a function exits, it checks whether the canary has been changed, and kills the application if it has.

ProPolice [2] is the most popular variation of the stack canaries countermeasure. It reorders the local variables of a function on the stack, in order to make sure that buffers are placed as close to the canary as possible. However, even ProPolice is still vulnerable to information leakage [3], format string vulnerabilities [4], or any attack that does not target the stack (for example, heap-based buffer overflows). It will also not emit the canary for every function, which can lead to vulnerabilities[6].

Address Space Layout Randomization (ASLR, [5]) randomizes the base address of important structures such as the stack, heap, and libraries, making it more difficult for attackers to find their injected shellcode in memory. Even if they succeed in overwriting a code pointer, they will not know where to point it to.

ASLR raises the security bar at no performance cost. However, there are different ways to get around the protection it provides. ASLR is susceptible to information leakage, in particular buffer-overreads [3] and format string vulnerabilities [4]. On 32-bit architectures, the amount of randomization is not prohibitively large [6], enabling an attacker to correctly guess addresses. New attacks also use a technique called heap-spraying [7]. Attackers pollute the heap by filling it with numerous copies of their shellcode, and then jump to somewhere on the heap. Because most of the memory is filled with their shellcode, there is a good chance that the jump will land on an address that is part of their shellcode.

Non-executable memory is supported on most modern CPUs, and allows applications to mark memory pages as non-executable. Even if the attacker can inject shellcode into the application and jump to it, the processor would refuse to execute it. There is no performance overhead when using this countermeasure, and it raises the security bar quite a bit. However, some processors still do not have this feature, and even if it is present in hardware, operating systems do not always turn it on by default. Linux supports non-executable memory, but many distributions do not use it, or only use it for some memory regions. A reason for not using it, is that it breaks applications that expect the stack or heap to be executable.

But even applications that use non-executable memory are vulnerable to attack. Instead of injecting code directly, attackers can inject a specially crafted fake stack. If the application starts unwinding the stack, it will unwind the fake stack instead of the original calling stack. This allows an attacker to direct the processor to arbitrary functions in libraries or program code, and choose which

[6] CVE-2007-0038

parameters are passed to these functions. This type of attack is referred to as a *return-into-libc* attack [8]. A related attack is called *return-oriented programming* [9], where a similar effect is achieved by filling the stack with return addresses to specifically chosen locations in code memory that execute some instructions and then perform a return. Other attacks exist that bypass non-executable memory by first marking the memory where they injected their code as executable, and then jumping to it [10].

Control Flow Integrity (CFI, [11]) is not a widely deployed countermeasure, but is discussed here because it is the countermeasure with the closest relation to CPM. CFI determines a program's control flow graph beforehand and ensures that the program adheres to it. It does this by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. CFI has been formally proven correct. Hence, under the assumptions made by the authors, an attacker will never be able to divert the control flow of an application that is protected with CFI.

CFI is related to CPM in that both countermeasures constrain the control flow of an application, but the mechanisms that are used to enforce this are different. The evaluation in Section 5 shows that CFI gives stronger guarantees, but the model assumes a weaker attacker and its implementation is substantially slower.

3 Code Pointer Masking

Existing countermeasures that protect code pointers can be roughly divided into two classes. The first class of countermeasures makes it hard for an attacker to change specific code pointers. An example of this class of countermeasures is Multistack [12]. In the other class, the countermeasures allow an attacker to modify code pointers, but try to detect these changes before any harm can happen. Examples of such countermeasures are stack canaries [13], pointer encryption [14] and CFI [11]. These countermeasures will be further explained in Section 7.

This section introduces the *Code Pointer Masking (CPM)* countermeasure, located somewhere between those two categories of countermeasures. CPM does not prevent overwriting code pointers, and does not detect memory corruptions, but it makes it hard or even impossible for an attacker to do something useful with a code pointer.

3.1 General Overview

CPM revolves around two core concepts: *code pointers* and *pointer masking.* A code pointer is a value that is stored in memory and that at some point in the application's lifetime is copied into the program counter register. If an attacker can change a code pointer, he will also be able to influence the control flow of the application.

CPM introduces masking instructions to mitigate the effects of a changed code pointer. After loading a (potentially changed) code pointer from memory into a register, but before actually using the loaded value, the value will be sanitized by combining it with a specially crafted and pointer-specific bit pattern. This process is called *pointer masking.*

By applying a mask, CPM will be able to selectively set or unset specific bits in the code pointer. Hence, it is an efficient mechanism to limit the range of addresses that are possible. Any bitwise operator (e.g. AND, OR, BIC (bit clear — AND NOT), ...) can be used to apply the mask on the code pointer. Which operator should be selected depends on how the layout of the program memory is defined. On Linux, using an AND or a BIC operator is sufficient. Even though an application may still have buffer overflow vulnerabilities, it becomes much harder for the attacker to exploit them in a way that might be useful.

The computation of the mask is done at link time, and depends on the type of code pointer. For instance, generating a mask to protect the return value of a function differs from generating a mask to protect function pointers. An overview of the different computation strategies is given in the following sections. The masks are not secret and no randomization whatsoever is used. An attacker can find out the values of the different masks in a target application by simply compiling the same source code with a CPM compiler. Knowing the masks will not aid the attacker in circumventing the masking process. It can, however, give the attacker an idea of which memory locations can still be returned to. But due to the narrowness of the masks (see Section 5.1), it is unlikely that these locations will be interesting for the attacker.

3.2 Assumptions

The design of CPM provides protection even against powerful attackers. It is, however, essential that two assumptions hold:

1. *Program code is non-writable.* If the attacker can arbitrarily modify program code, it is possible to remove the masking instructions that CPM adds. This defeats the entire masking process, and hence the security of CPM. Non-writable program code is the standard nowadays, so this assumption is more than reasonable.
2. *Code injection attacks overwrite a code pointer eventually.* CPM protects code pointers, so attacks that do not overwrite code pointers are not stopped. However, all known attacks that allow an attacker to execute arbitrary code overwrite at least one code pointer.

3.3 Masking the Return Address

The return address of a function is one of the most popular code pointers that is used in attacks to divert the control flow. Listing 1.1 shows the sequence of events in a normal function epilogue. First, the return address is retrieved from the stack and copied into a register. Then, the processor is instructed to jump to the address in the register. Using for instance a stack based buffer overflow, the attacker can overwrite the return address on the stack. Then, when the function executes its epilogue, the program will retrieve the modified address from the stack, store it into a register, and will jump to an attacker-controlled location in memory.

Listing 1.1. A normal function epilogue.

```
[get return address from stack]
[jump to this address]
```

CPM mitigates this attack by inserting a masking instruction inbetween, as shown in Listing 1.2. Before the application jumps to the code pointer, the pointer is first modified in such a way that it cannot point to a memory location that falls outside of the code section.

Listing 1.2. A CPM function epilogue.

```
[get return address from stack]
[apply bitmask on address]
[jump to this masked address]
```

The mask is function-specific and is calculated by combining the addresses of the different return sites of the function using an OR operation. In general, the quality of a return address mask is proportional to the number of return sites that the mask must allow. Hence, fewer return sites results on average in a better mask. As the evaluation in Section 5.2 shows, it turns out that most functions in an application have only a few callers.

However, the quality is also related to how many bits are set in the actual addresses of the return sites, and how many bits of the different return addresses overlap. Additional logic is added to the compiler to move methods around, in order to optimize these parameters.

Example Assume that we have two methods M1 and M2, and that these methods are the only methods that call a third method M3. Method M3 can return to a location somewhere in M1 or M2. If we know during the compilation of the application that these return addresses are located at memory location 0x0B3E (0000101100111110) for method M1 and memory location 0x0A98 (0000101010011000) for method M2, we can compute the mask of method M3 by ORing the return sites together. The final mask that will be used is mask 0x0BBE (0000101110111110).

By ANDing this generated mask and the return address, the result of this operation is limited to the return locations in M1 and M2, and to a limited number of other locations. However, most of the program memory will not be accessible anymore, and all other memory outside the program code section (for example, the stack, the heap, library memory, ...) will be completely unreachable.

3.4 Masking Function Pointers

It is very difficult to statically analyze a C program to know beforehand which potential addresses can be called from some specific function pointer call. CPM solves this by overestimating the mask it uses. During the compilation of the program, CPM scans through the source code of the application and detects for which functions the address is taken, and also detects where function pointer calls are located. It changes the masks of the functions that are called to ensure that they can also return to any return site of a function pointer call. In addition, the masks that are used to mask the function pointers are selected in such a way that they allow a jump to all the different functions whose addresses have been taken somewhere in the program. As Section 5.1 shows, this has no important impact on the quality of the masks of the programs in the benchmark.

The computation of the function pointer mask is similar to the computation of the return address masks. The compiler generates a list of functions whose addresses are taken in the program code. These addresses are combined using an OR operation into the final mask that will be used to protect all the function pointer calls.

A potential issue is that calls of function pointers are typically implemented as a *JUMP <register>* instruction. There is a very small chance that if the attacker is able to overwrite the return address of a function and somehow influence the contents of this register, that he can put the address of his shellcode in the register and modify the return address to point to this *JUMP <register>* instruction. Even if this jump is preceded by a masking operation, the attacker can skip this operation by returning to the JUMP instruction directly. Although the chances for such an attack to work are extremely low (the attacker has to be able to return to the JUMP instruction, which will in all likelihood be prevented by CPM in the first place), CPM specifically adds protection to counter this threat.

The solutions to this problem depends from architecture to architecture. For example, CPM can reserve a register that is used exclusively to perform the masking of code pointers. This will make sure that the attacker can never influence the contents of this register. The impact of this particular solution will differ from processor to processor, because it increases the register pressure. However, as the performance evaluation in Section 5.1 shows, on the ARM architecture this *is* a good solution. And because both the Intel 64 and AMD64 architectures sport additional general purpose registers, a similar approach can be implemented here as well.

3.5 Masking the Global Offset Table

A final class of code pointers that deserves special attention are entries in the *global offset table (GOT)*. The GOT is a table that is used to store offsets to objects that do not have a static location in memory. This includes addresses of dynamically loaded functions that are located in libraries.

At program startup, these addresses are initialized to point to a helper method that loads the required library. After loading the library, the helper method modifies the addresses in the GOT to point to the library method directly. Hence, the second time the application tries to call a library function, it will jump immediately to the library without having to go through the helper method.

Overwriting entries in the GOT by means of indirect pointer overwriting is a common attack technique. By overwriting addresses in the GOT, an attacker can redirect the execution flow to his shellcode. When the application unsuspectedly calls the library function whose address is overwritten, the attacker's shellcode is executed instead.

Like the other code pointers, the pointers in the GOT are protected by masking them before they are used. Since all libraries are loaded into a specific memory range (e.g. 0x4NNNNNNN on 32-bit Linux), all code pointers in the GOT must either be somewhere in this memory range, or must point to the helper method (which is located in the program code memory). CPM adds instructions that ensure this, before using a value from the GOT.

3.6 Masking Other Code Pointers

CPM protects all code pointers in an application. This section contains the code pointers that have not been discussed yet, and gives a brief explanation of how they are protected.

On some systems, when an application shuts down it can execute a number of so-called destructor methods. The *destructor table* is a table that contains pointers to these methods, making it a potential target for a code injection attack. If an attacker is able to overwrite one of these pointers, he might redirect it to injected code. This code will then be run when the program shuts down. CPM protects these pointers by modifying the routine that reads entries from the destructor table.

Applications might also contain a *constructor table*. This is very similar to the destructor table, but runs methods at program startup instead of program shutdown. This table is not of interest to CPM, because the constructors will have already executed before an attacker can start attacking the application and the table is not further used.

The C standard also offers support for *long jumps*, a feature that is used infrequently. A programmer can save the current program state into memory, and then later jump back to this point. Since this memory structure contains the location of where the processor is executing, it is a potential attack target. CPM protects this code pointer by adding masking operations to the implementation of the longjmp method.

4 Implementation

This section describes the implementation of the CPM prototype for the ARM architecture. It is implemented in gcc-4.4.0 and binutils-2.20 for Linux. For GCC, the machine descriptions are changed to emit the masking operations during the conversion from RTL[7] to assembly. The implementation provides the full CPM protection for return addresses, function pointers, GOT entries, and the other code pointers.

4.1 Function Epilogue Modifications

Function returns on ARM generally make use of the `LDM` instruction. `LDM`, an acronym for 'Load Multiple', is similar to a `POP` instruction on x86. But instead of only popping one value from the stack, `LDM` pops a variable number of values from the stack into multiple registers. In addition, the ARM architecture also supports writing directly to the program counter register. Hence, GCC uses a combination of these two features to produce an optimized epilogue. Listing 1.3 shows what this epilogue looks like.

Listing 1.3. A function prologue and epilogue on ARM.

```
stmfd   sp!, {<registers>, fp, lr}
...
ldmfd   sp!, {<registers>, fp, pc}
```

The `STMFD` instruction stores the given list of registers to the address that is pointed to by the *sp* register. <*registers*> is a function-specific list of registers that are modified during the function call and must be restored afterwards. In addition, the frame pointer and the link register (that contains the return address) are also stored on the stack. The exclamation mark after the *sp* register means that the address in the register will be updated after the instruction to reflect the new top of the stack. The 'FD' suffix of the instruction denotes in which order the registers are placed on the stack.

Similarly, the `LDMFD` instruction loads the original values of the registers back from the stack, but instead of restoring the *lr* register, the original value of this register is copied to *pc*. This causes the processor to jump to this address, and effectively returns to the parent function.

Listing 1.4 shows how CPM rewrites the function epilogue. The `LDMFD` instruction is modified to not pop the return address from the stack into PC. Instead, the return address is popped off the stack by the subsequent `LDR` instruction into the register *r9*. We specifically reserve register *r9* to perform all

[7] RTL or *Register Transfer Language* is one of the intermediate representations that is used by GCC during the compilation process.

Listing 1.4. A CPM function prologue and epilogue on ARM.

```
stmfd   sp!, {<registers>, fp, lr}
...
ldmfd   sp!, {<registers>, fp}
ldr     r9, [sp], #4
bic     r9, r9, #0xNN000000
bic     r9, r9, #0xNN0000
bic     r9, r9, #0xNN00
bic     pc, r9, #0xNN
```

the masking operations of CPM. This ensures that an attacker will never be able to influence the contents of the register, as explained in Section 3.4.

Because ARM instructions cannot take 32-bit operands, we must perform the masking in multiple steps. Every bit-clear (`BIC`) operation takes an 8-bit operand, which can be shifted. Hence, four `BIC` instructions are needed to mask the entire 32-bit address. In the last `BIC` operation, the result is copied directly into *pc*, causing the processor to jump to this address.

The mask of a function is calculated in the same way as explained in Section 3.3, with the exception that it is negated at the end of the calculation. This is necessary because our ARM implementation does not use the `AND` operator but the `BIC` operator.

Alternative function epilogues that do not use the `LDM` instruction are protected in a similar way. Masking is always done by performing four `BIC` instructions.

4.2 Procedure Linkage Table Entries

As explained in Section 3.5, applications use a structure called *the global offset table* in order to enable dynamically loading libraries. However, an application does not interact directly with the GOT. It interacts with a jump table instead, called *the Procedure Linkage Table (PLT)*. The PLT consists of PLT entries, one for each library function that is called in the application. A PLT entry is a short piece of code that loads the correct address of the library function from the GOT, and then jumps to it.

Listing 1.5. A PLT entry that does not perform masking.

```
add     ip, pc, #0xNN00000
add     ip, ip, #0xNN000
ldr     pc, [ip, #0xNNN]!
```

Listing 1.5 shows the standard PLT entry that is used by GCC on the ARM architecture. The address of the GOT entry that contains the address of the library function is calculated in the *ip* register. Then, in the last instruction, the address of the library function is loaded from the GOT into the *pc* register, causing the processor to jump to the function.

CPM protects addresses in the GOT by adding masking instructions to the PLT entries. Listing 1.6 shows the modified PLT entry.

Listing 1.6. A PLT entry that performs masking.

```
add     ip, pc, #0xNN00000
add     ip, ip, #0xNN000
ldr     r9, [ip, #0xNNN]!
cmp     r9, #0x10000
orrge   r9, r9, #0x40000000
bicge   pc, r9, #0xB0000000
bic     r9, r9, #0xNN000000
bic     r9, r9, #0xNN0000
bic     r9, r9, #0xNN00
bic     pc, r9, #0xNN
```

The first three instructions are very similar to the original code, with the exception that the address stored in the GOT is not loaded into *pc* but in *r9* instead. Then, the value in *r9* is compared to the value 0x10000.

If the library *has not* been loaded yet, the address in the GOT will point to the helper method that initializes libraries. Since this method is always located on a memory address below 0x10000, the `CMP` instruction will modify the status flags to 'lower than'. This will force the processor to skip the two following `ORRGE` and `BICGE` instructions, because the suffix 'GE' indicates that they should only be executed if the status flag is 'greater or equal'. The address in *r9* is subsequently masked by the four `BIC` instructions, and finally copied into *pc*.

If the library *has* been loaded, the address in the GOT will point to a method loaded in the 0x4NNNNNNN address space. Hence, the `CMP` instruction will set the status flag to 'greater than or equal', allowing the following `ORRGE` and `BICGE` instructions to execute. These instructions will make sure that the most-significant four bits of the address are set to 0x4, making sure that the address will always point to the memory range that is allocated for libraries. The `BICGE` instruction copies the result into *pc*.

4.3 Protecting Other Code Pointers

The protection of function pointers is similar to the protection of the return address. Before jumping to the address stored in a function pointer, it is first masked with four `BIC` operations, to ensure the pointer has not been corrupted. Register *r9* is also used here to do the masking, which guarantees that an attacker cannot interfere with the masking, or jump over the masking operations.

The *long jumps* feature of C is implemented on the ARM architecture as an `STM` and an `LDM` instruction. The behavior of the longjmp function is very similar to the epilogue of a function. It loads the contents of a memory structure into a number of registers. CPM modifies the implementation of the longjmp function in a similar way as the function epilogues. The `LDM` instruction is changed that it does not load data into the program counter directly, and four `BIC` instructions are added to perform the masking and jump to the masked location.

4.4 Limitations of the Prototype

In some cases, the CPM prototype cannot calculate the masks without additional input. The first case is when a function is allowed to return to library code. This happens when a library method receives a pointer to an application function as a parameter, and then calls this function. This function will return back to the library function that calls it.

The prototype compiler solves this by accepting a list of function names where the masking should not be done. This list is program-specific and should be maintained by the developer of the application. In the SPEC benchmark, only one application has one method where masking should be avoided.

The second scenario is when an application generates code or gets a code pointer from a library, and then tries to jump to it. CPM will prevent the application from jumping to the function pointer, because it is located outside the acceptable memory regions. A similar solution can be used as described in the previous paragraph. None of the applications in the SPEC benchmark displayed this behavior.

5 Evaluation

In this section, we report on the performance of our CPM prototype, and discuss the security guarantees that CPM provides.

5.1 Compatibility, Performance and Memory Overhead

To test the compatibility of our countermeasure and the performance overhead, we ran the SPEC benchmark [15] with our countermeasure and without. All tests were run on a single machine (ARMv7 Processor running at 800MHz, 512Mb RAM, running Ubuntu Linux with kernel 2.6.28).

All C programs in the SPEC CPU2000 Integer benchmark were used to perform these benchmarks. Table 1 contains the runtime in seconds when compiled with the unmodified GCC on the ARM architecture, the runtime when compiled with the CPM countermeasure, and the percentage of overhead.

Most applications have a performance hit that is less than a few percent, supporting our claim that CPM is a highly efficient countermeasure. There are no results for VORTEX, because it does not work on the ARM architecture.

SPEC CPU2000 Integer benchmarks					
Program	**GCC (s)**	**CPM (s)**	**Overhead**	**Avg. Mask size**	**Jump surface**
164.gzip	808	824	+1.98%	10.4 bits	2.02%
175.vpr	2129	2167	+1.78%	12.3 bits	1.98%
176.gcc	561	573	+2.13%	13.8 bits	0.94%
181.mcf	1293	1297	+0.31%	8.3 bits	1.21%
186.crafty	715	731	+2.24%	13.1 bits	3.10%
197.parser	1310	1411	+7.71%	10.7 bits	1.18%
253.perlbmk	809	855	+5.69%	13.2 bits	1.51%
254.gap	626	635	+1.44%	11.5 bits	0.57%
256.bzip2	870	893	+2.64%	10.9 bits	3.37%
300.twolf	2137	2157	+0.94%	12.9 bits	3.17%

Table 1. Benchmark results of the CPM countermeasure on the ARM architecture

Running this application with an unmodified version of GCC results in a memory corruption (and crash).

The memory overhead of CPM is negligible. CPM increases the size of the binary image of the application slightly, because it adds a few instructions to every function in the application. CPM also does not allocate or use memory at runtime, resulting in a memory overhead of practically 0%.

The SPEC benchmark also shows that CPM is highly compatible with existing code. The programs in the benchmark add up to a total of more than 500,000 lines of C code. All programs were fully compatible with CPM, with the exception of only one application where a minor manual intervention was required (see Section 4.4).

5.2 Security Evaluation

As a first step in the evaluation of CPM, some field tests were performed with the prototype. Existing applications and libraries that contain vulnerabilities[8] were compiled with the new countermeasure. CPM did not only stop the existing attacks, but it also raised the bar to further exploit these applications. However, even though this gives an indication of some of the qualities of CPM, it is not a complete security evaluation.

The security evaluation of CPM is split into two parts. In the first part, CPM is compared to the widely deployed countermeasures. Common attack scenarios are discussed, and an explanation is given of how CPM protects the application in each case. The second part of the security evaluation explains which security guarantees CPM provides, and makes the case for CPM by using the statistics we have gathered from the benchmarks.

[8] CVE-2006-3459 and CVE-2009-0629

	ProPolice	**ASLR**[1]	**NX**[2]	**Combination**[3]
Stack-based buffer overflow	IL	HS, IL	RiC	IL+RiC
Heap-based buffer overflow	N/A	HS, IL	RiC	IL+RiC, HS+RiC
Indirect pointer overwrite	N/A	HS, IL	RiC	IL+RiC, HS+RiC
Dangling pointer references	N/A	HS, IL	RiC	IL+RiC, HS+RiC
Format string vulnerabilities	N/A	HS, IL	RiC	IL+RiC, HS+RiC

[1] = This assumes the strongest form of ASLR, where the stack, the heap, and the libraries are randomized. On Linux, only the stack is randomized.
[2] = This assumes that all memory, except code and library memory, is marked as non-executable. On Linux, this depends from distribution to distribution, and is often not the case.
[3] = This is the combination of the ProPolice, ASLR and No-Execute countermeasures, as deployed in modern operating systems.

Table 2. An overview of how all the widely deployed countermeasures can be broken by combining different common attack techniques: Heap spraying (HS), Information leakage (IL) and Return-into-libc/Return-oriented programming (RiC).

CPM versus Widely Deployed Countermeasures Table 2 shows CPM, compared in terms of security protection to widely deployed countermeasures (see Section 2). The rows in the table represent the different vulnerabilities that allow code injection attacks, and the columns represent the different countermeasures.

Each cell in the table contains the different (combinations of) attack techniques (see Section 2) that can be used to break the security of the countermeasure(s). The different techniques that are listed in the table are *return-into-libc/return-oriented programming (**RiC**)*, *information leakage (**IL**)*, and *heap spraying (**HS**)*. CPM is the only countermeasure that offers protection against all different combinations of common attack techniques, albeit not a provably perfect protection.

Applications that are protected with the three widely deployed countermeasures can be successfully attacked by using a combination of two common attack techniques. If the application leaks sensitive information [3], the attacker can use this information to break ASLR and ProPolice, and use a Return-into-libc attack, or the newer but related Return-oriented Programming attacks, to break No-Execute. If the application does not leak sensitive data, the attacker can use a variation of a typical heap spraying attack to fill the heap with a fake stack and then perform a Return-into-libc or Return-oriented Programming attack.

CPM protects against Return-into-libc attacks and Return-oriented Programming attacks [9] by limiting the amount of return sites that the attacker can return to. Both attacks rely on the fact that the attacker can jump to certain interesting points in memory and abuse existing code (either in library code memory or application code memory). However, the CPM masks will most likely not give the attacker the freedom he needs to perform a successful attack. In particular, CPM will not allow returns to library code, and will only allow returns to a limited part of the application code. Table 1 shows for each application the jump surface, which represents the average surface area of the program code memory that an attacker can jump to with a masked code pointer (without CPM, these values would all be 100%).

Protection against spraying shellcode on the heap is easy for CPM: the masks will never allow an attacker to jump to the heap (or any other data structure, such as the stack), rendering this attack completely useless. An attacker can still spray a fake stack, but he would then have to perform a successful return-into-libc or return-oriented programming attack, which is highly unlikely as explained in the previous paragraph.

CPM can also not be affected by information that an attacker obtained through memory leaks, because it uses no secret information. The masks that are calculated by the compiler are *not* secret. Even if an attacker knows the values of each individual mask, this will not aid him in circumventing the CPM masking process. It can give him an idea of which memory locations can still be returned to, but due to the narrowness of the masks it is unlikely that these locations will be interesting.

Like many other compiler-based countermeasures, all libraries that an application uses must also be compiled with CPM. Otherwise, vulnerabilities in these libraries may still be exploited. However, CPM is fully compatible with unprotected libraries, thus providing support for linking with code for which the source may not be available.

CPM was designed to provide protection against the class of code injection attacks, but other types of attacks might still be feasible. In particular, data-only attacks [16], where an attacker overwrites application data and no code pointers, are not protected against by CPM.

CPM Security Properties The design of CPM depends on three facts that determine the security of the countermeasure.

CPM masks all code pointers. Code pointers that are not masked are still potential attack targets. For the ARM prototype, we mask all the different code pointers that are described in related papers. In addition, we looked at all the code that GCC uses to emit jumps, and verified whether it should be a target for CPM masking.

Masking is non-bypassable. All the masking instructions CPM emits are located in read-only program code. This guarantees that an attacker can never modify the instructions themselves. In addition, the attacker will not be able to skip the masking process. On the ARM architecture, we ensure this by reserving register *r9* and using this register to perform all the masking operations and the computed jumps.

The masks are narrow. How narrow the masks can be made differs from application to application and function to function. Functions with few callers will typically generate more narrow masks than functions with a lot of callers. The assumption that most functions have only a few callers is supported by the statistics. In the applications of the SPEC benchmark, 27% of the functions had just one caller, and 55% of the functions had three callers or less. Around 1.20% of the functions had 20 or more callers. These functions are typically library functions such as *memcpy*, *strncpy*, ... To improve the masks, the compiler shuffles functions around and sprinkles a small amount of padding in-between the functions. This is to ensure that return addresses contain as many 0-bits as possible. With this technique, we can reduce the number of bits that are set to 1 in the different function-specific masks. Without CPM, an attacker can jump to any address in memory (2^{32} possibilities on a 32-bit machine). Using the techniques described here, the average number of bits per mask for the applications in the SPEC benchmark can be brought down to less than 13 bits. This means that by using CPM for these applications, the average function is limited to returning to less than 0.0002% of the entire memory range of an application.

CPM has the same high-level characteristics as the CFI countermeasure, but it defends against a somewhat stronger attack model. In particular, non-executable data memory is not required for CPM. If the masks can be made so precise that they only allow the correct return sites, an application protected with CPM will never be able to divert from the intended control flow. In this case, CPM offers the exact same guarantees that CFI offers. However, in practice, the masks will not be perfect. Hence, CPM can be seen as an efficient approximation of CFI.

The strength of protection that CPM offers against diversion of control flow depends on the precision of the masks. An attacker can still jump to any location allowed by the mask, and for some applications this might still allow interesting attacks. As such, CPM offers fewer guarantees than CFI. However, given the fact that the masks are very narrow, it is extremely unlikely that attackers will be able to exploit the small amount of room they have to maneuver. The SPEC benchmark also shows that CPM offers a performance that is much better than CFI[9]. This can be attributed to the fact that CPM does not access the memory in the masking operations, whereas CFI has to look up the labels that are stored in the memory. Finally, CPM offers support for dynamically linked code, a feature that is also lacking in CFI.

6 Discussion and Ongoing Work

CPM overlaps in part with other countermeasures, but also protects against attacks that are not covered. Vice versa, there are some attacks that might work on CPM (i.e. attacks that do not involve code injection, such as data-only attacks), which might not work with other countermeasures. Hence, CPM is complementary to existing security measures, and in particular can be combined with popular countermeasures such as non-executable memory, stack canaries and ASLR[10]. Adding CPM to the mix of existing protections significantly raises

[9] CFI has an overhead of up to 45%, with an average overhead of 16% on the Intel x86 architecture. Results for CPM are measured on the ARM architecture.

[10] As implemented in current operating systems, where only the stack and the heap are randomized.

the bar for attackers wishing to perform a code injection attack. One particular advantage of CPM is that it offers protection against a combination of different attack techniques, unlike the current combination of widely deployed countermeasures.

When an attacker overwrites a code pointer somewhere, CPM does not detect this modification. Instead it will mask the code pointer and jump to the sanitized address. An attacker can still crash the application by writing rubbish in the code pointer. The processor would jump to the masked rubbish address, and will very likely crash at some point. But most importantly, the attacker will not be able to execute his payload. CPM can be modified to detect any changes to the code pointer, and abort the application in that case. This functionality can be implemented in 7 ARM instructions (instead of 4 instructions), but does temporarily require a second register for the calculations.

The mechanism of CPM can be ported to other architectures. A second prototype exists for the x86 architecture, but is not reported on in this paper because of page limit constraints and because it is still incomplete. However, protection of the largest class of code pointers — the return address — works, and its performance is comparable to the performance on the ARM architecture.

A promising direction of future work is processor-specific enhancements. In particular, on the ARM processor, the conditional execution feature may be used to further narrow down the destination addresses that an attacker can use to return to. Conditional execution allows almost every instruction to be executed conditionally, depending on certain status bits. If these status bits are flipped when a return from a function occurs, and flipped again at the different (known) return sites in the application, the attacker is forced to jump to one of these return addresses, or else he will land on an instruction that will not be executed by the processor.

7 Related work

Many countermeasures have been designed to protect against code injection attacks. In this section, we briefly highlight the differences between our approach and other approaches that protect programs against attacks on memory error vulnerabilities. For a more complete survey of code injection countermeasures, we refer the reader to [17].

Bounds checkers Bounds checking [18] is a better solution to buffer overflows, however when implemented for C, it has a severe impact on performance and may cause existing code to become incompatible with bounds checked code. Recent bounds checkers [19, 20] have improved performance somewhat, but still do not protect against dangling pointer vulnerabilities, format string vulnerabilities, and others.

Probabilistic countermeasures Many countermeasures make use of randomness when protecting against attacks. Many different approaches exist when using randomness for protection. Canary-based countermeasures [13] use a secret random number that is stored before an important memory location: if the random number has changed after some operations have been performed, then an attack has been detected. Memory-obfuscation countermeasures [14] encrypt important memory locations using random numbers. Memory layout randomizers [21] randomize the layout of memory by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers [22] encrypt the instructions while in memory and will decrypt them before execution.

While these approaches are often efficient, they rely on keeping memory locations secret. Different attacks exist where the attacker is able exploit leaks to read the memory of the application [3]. Such memory leaking vulnerabilities can allow attackers to bypass this type of countermeasure.

Separation and replication of information Countermeasures that rely on separation or replication of information will try to replicate valuable control-flow information or will separate this information from regular data [23]. These countermeasures are easily bypassed using indirect pointer overwriting where an attacker overwrites a different memory location instead of the return address by using a pointer on the stack. More advanced techniques try to separate all control-flow data (like return addresses and pointers) from regular data [12], making it harder for an attacker to use an overflow to overwrite this type of data.

While these techniques can efficiently protect against buffer overflows that try to overwrite control-flow information, they do not protect against attacks where an attacker controls an integer that is used as an offset from a pointer.

Another widely deployed countermeasure distinguishes between memory that contains code and memory that contains data. Data memory is marked as non-executable [21]. This simple countermeasure is effective against direct code injection attacks (i.e. attacks where the attacker injects code as data), but provides no protection against indirect code injection attacks such as return-to-libc attacks. CPM can provide protection against both direct and indirect code injection.

Software Fault Isolation Software Fault Isolation (SFI) [24] was not developed as a countermeasure against code injection attacks in C, but it does have some similarities with CPM. In SFI, data addresses are masked to ensure that untrusted code cannot (accidentally) modify parts of memory. CPM on the other hand masks code addresses to ensure that control flow can not jump to parts of memory.

Execution monitors Some existing countermeasures monitor the execution of a program and prevent transferring control-flow which can be unsafe.

Program shepherding [25] is a technique that monitors the execution of a program and will disallow control-flow transfers that are not considered safe. Existing implementations have a significant performance impact for some programs, but acceptable for others.

Control-flow integrity, as discussed in Section 5.2, is also a countermeasure that is classified as an execution monitor.

8 Conclusion

The statistics and recent high-profile security incidents show that code injection attacks are still a very important security threat. There are different ways in which a code injection attack can be performed, but they all share the same characteristic in that they all overwrite a code pointer at some point.

CPM provides an efficient mechanism to strongly mitigate the risk of code injection attacks in C programs. By masking code pointers before they are used, CPM imposes restrictions on these pointers that render them useless to attackers.

CPM offers an excellent performance/security trade-off. It severely limits the risk of code injection attacks, at only a very small performance cost. It seems to be well-suited for handheld devices with slow processors and little memory, and can be combined with other countermeasures in a complementary way.

Acknowledgements This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

1. National Institute of Standards and Technology, “National vulnerability database statistics.” http://nvd.nist.gov/statistics.cfm.
2. H. Etoh and K. Yoda, “Protecting from stack-smashing attacks,” tech. rep., IBM Research Divison, June 2000.
3. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, “Breaking the memory secrecy assumption,” in *Proceedings of the European Workshop on System Security (Eurosec)*, (Nuremberg, Germany), Mar. 2009.
4. K. S. Lhee and S. J. Chapin, “Buffer overflow and format string overflow vulnerabilities,” *Software: Practice and Experience*, vol. 33, pp. 423–460, April 2003.
5. S. Bhatkar, D. C. Duvarney, and R. Sekar, “Address obfuscation: An efficient approach to combat a broad range of memory error exploits,” in *Proceedings of the 12th USENIX Security Symposium*, USENIX Association, August 2003.
6. H. Shacham, M. Page, B. Pfaff, E. J. Goh, N. Modadugu, and D. Boneh, “On the Effectiveness of Address-Space Randomization,” in *Proceedings of the 11th ACM conference on Computer and communications security*, October 2004.
7. F. Gadaleta, Y. Younan, and W. Joosen, “Bubble: A javascript engine level countermeasure against heap-spraying attacks,” in *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSOS)*, 2010.
8. R. Wojtczuk, “Defeating solar designer non-executable stack patch.” Posted on the Bugtraq mailinglist, February 1998.
9. H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM conference on Computer and communications security*, (Washington, D.C., U.S.A.), pp. 552–561, ACM, ACM Press, October 2007.

10. skape and Skywing, "Bypassing windows hardware-enforced data execution prevention," *Uninformed*, vol. 2, Sept. 2005.
11. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, (Alexandria, Virginia, U.S.A.), pp. 340–353, ACM, November 2005.
12. Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *Proceedings of the Twenty-Second Annual Computer Security Applications Conference (ACSAC '06)*, pp. 429–438, IEEE Press, December 2006.
13. C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 7th USENIX Security Symposium*, (San Antonio, Texas, U.S.A.), USENIX Association, January 1998.
14. C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium*, pp. 91–104, USENIX Association, August 2003.
15. J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, pp. 28–35, July 2000.
16. U. Erlingsson, "Low-level software security: Attacks and defenses," Tech. Rep. MSR-TR-2007-153, Microsoft Research, 2007.
17. Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against c and c++ programs," *ACM Computing Surveys*, 2010.
18. Y. Oiwa, T. Sekiguchi, E. Sumii, and A. Yonezawa, "Fail-safe ANSI-C compiler: An approach to making C programs secure: Progress report," in *Proceedings of International Symposium on Software Security 2002*, November 2002.
19. P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th USENIX Security Symposium*, (Montreal, QC), Aug. 2009.
20. Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "Paricheck: An efficient pointer arithmetic checker for c programs," in *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, (Bejing, China), ACM, Apr. 2010.
21. The PaX Team, "Documentation for the PaX project."
22. E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pp. 281–289, ACM, October 2003.
23. T. Chiueh and F. H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, (Phoenix, Arizona, USA), pp. 409–420, IEEE Computer Society, IEEE Press, April 2001.
24. S. Mccamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proceedings of the 15th USENIX Security Symposium*, (Vancouver, British Columbia, Canada), USENIX Association, August 2006.
25. V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *Proceedings of the 11th USENIX Security Symposium*, (San Francisco, California, U.S.A.), USENIX Association, August 2002.