CPM: Masking Code Pointers to Prevent Code Injection Attacks

PIETER PHILIPPAERTS, YVES YOUNAN, STIJN MUYLLE and FRANK PIESSENS,
DistriNet Research Group, University of Leuven
SVEN LACHMUND and THOMAS WALTER, DOCOMO Euro-Labs

Code Pointer Masking (CPM) is a novel countermeasure against code injection attacks on native code. By enforcing the correct semantics of code pointers, CPM thwarts attacks that modify code pointers to divert the application's control flow. It 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. This paper reports on two prototype implementations on very distinct processor architectures, showing that the idea behind CPM is portable. The evaluation also shows that the overhead of using our countermeasure is very small and the security benefits are substantial.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Security, Algorithms

Additional Key Words and Phrases: code injection,code pointer,countermeasure,masking

ACM Reference Format:
Philippaerts, P., Younan, Y., Muylle, S., Piessens, F., Lachmund, S., Walter, T. 2012. CPM: Masking Code Pointers to Prevent Code Injection Attacks. ACM Trans. Info. Syst. Sec. 0, 0, Article 0 (2012), 27 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Being able to silently break into a computer system and taking it over without the legitimate user realizing this is the most optimal scenario for any hacker. The attacker can spy on the user, abuse the available resources (in a botnet for example), and use the computer to anonymize illicit activities. If the user of the system does not notice any of this, he is of course less likely to try and interfere with the attacker. To obtain this objective, a hacker usually exploits a bug in software that leads to a *code injection attack*. There are different types of code injection attacks, but in this paper we will use the term to refer to code injection attacks that exploit bugs in so-called native code.

In these 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. Several other — more advanced — attack techniques have also been developed, including heap based buffer overflows, indirect pointer overwrites, and others. All these attacks eventually overwrite a *code pointer*, e.g. a memory location that contains an address that the processor will jump to during program execution.

According to the NIST's National Vulnerability Database [National Institute of Standards and Technology], 9.86% of the reported vulnerabilities are buffer overflows, com-

Author's addresses: DistriNet Research Group, University of Leuven, Celestijnlaan 200A, B-3001 Leuven, Belgium; DOCOMO Euro-Labs, Landsberger Strasse 312, 80687 Munich, Germany.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

ing 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[1]. Google saw the security of its sandboxed browser *Chrome* breached[2] because of a code injection attack. And an attack exploiting a code injection vulnerability in Microsoft's Internet Explorer[3] 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. Contrary to other highly efficient countermeasures, CPM's security does not rely on secret data of any kind, and so cannot be bypassed if the attacker can read memory [Strackx et al. 2009; Lhee and Chapin 2003].

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 two prototype implementations for the ARM and x64 architectures that implement the full countermeasure.
— 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 work in this paper is an evolution of the work in [Philippaerts et al. 2011]. In particular, an entire new prototype on x64 has been implemented and evaluated. This shows that the idea behind CPM can be ported to vastly different processor architectures, and that its excellent performance is not specific to a single processor type.

The paper is structured as follows: Section 2 briefly describes the technical details of a typical code injection attack and gives an overview of common countermeasures. 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 ATTACKS

Code injection attacks occur when an attacker can successfully divert the processor's control flow to a memory location whose contents is controlled by an attacker. The only way an attacker can influence the control flow of the processor is by overwriting locations in memory that store so-called *code pointers*. A code pointer is a variable that

[1]CVE-2006-3459.
[2]CVE-2008-6994.
[3]CVE-2010-0249.

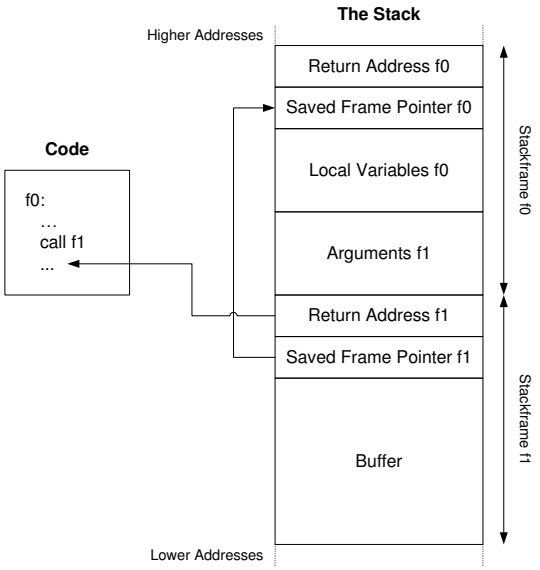

Fig. 1. A typical stack layout with two functions *f0* and *f1*

contains the memory address of a function or some other location in the application code where the processor will at some point jump to. Well-known code pointers are the return address and function pointers.

In this section, we briefly describe the most basic type of code injection attack, which occurs by writing outside the bounds of a buffer on the stack and overwriting the return address. This type of attack is not exploitable anymore in most cases, due to the deployment of various countermeasures. However, it is very easy to explain, and thus serves as a perfect illustration of the basics of a code injection attack. We then briefly discuss some widely deployed countermeasures, and also explain more advanced attack techniques that can be used to get around these countermeasures.

2.1. Stack-Based Buffer Overflows

When an array is declared in C, space is reserved for it and the array is manipulated by means of a pointer to the first byte. No information about the array size is available at runtime, and most C-compilers will generate code that will allow a program to copy data beyond the end of an array, overwriting adjacent memory space. If interesting information is stored somewhere in the adjacent memory space, it can be possible for an attacker to overwrite it. On the stack this is usually the case: it stores the addresses to resume execution after a function call has completed its execution, i.e., the return address.

For example, on the ARM and x64 architectures the stack grows down (i.e., newer function calls have their variables stored at lower address than older ones). The stack is divided into stackframes. Each stackframe contains information about the current function: arguments of the called function, registers whose values must be stored across function calls, local variables and the return address. This memory layout is shown in Figure 1. An array allocated on the stack will usually be located in the section of local variables of a stackframe. If a program copies data past the end of this array, it will overwrite anything else stored before it and thus will overwrite other data stored on the stack, like the return address.

If an attacker can somehow get binary code in the application's memory space, then he can use the above technique to overwrite a return address and divert the control flow to his binary code that is stored somewhere in the application's memory. This binary code is called *shellcode*, and is typically a very short code fragment that seeks to allow the attacker to execute arbitrary instructions with the same privilege level as the application. For instance, shellcode can be used to steal sensitive data or install a backdoor on the system. A common way of getting this shellcode in the memory space of the application is by giving it as input to the application. This input is then typically copied to the stack or the heap, where the attacker can then divert the control flow to.

The stack-based buffer overflow attack described earlier is the oldest and best-known code injection attack. However, more advanced attack techniques follow a similar pattern in the sense that at some point a code pointer gets overwritten. Our countermeasure applies to all these attacks, as we will discuss in Section 5.

2.2. Countermeasures and Advanced Attacks

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.

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.

The initial implementations of stack canaries were foiled by using indirect pointer overwrite attacks, where an attacker overwrites an unprotected pointer and integer value on the stack. If the application code later dereferences the pointer and overwrites the value with the integer, the attacker can write a random value anywhere in memory by simply manipulating the pointer and the integer. This allows an attacker to write any value over the return address on the stack without having to overwrite the canary first.

ProPolice [Etoh and Yoda 2000] 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 [Strackx et al. 2009], format string vulnerabilities [Lhee and Chapin 2003], 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[4].

Address Space Layout Randomization (ASLR, [Bhatkar et al. 2003]) 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 [Strackx et al. 2009] and format string vulnerabilities [Lhee and Chapin 2003]. On 32-bit architectures, the amount of randomization is

[4]CVE-2007-0038

not prohibitively large [Shacham et al. 2004], enabling an attacker to correctly guess addresses. New attacks also use a technique called heap-spraying [Gadaleta et al. 2010]. 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 some 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 parameters are passed to these functions. This type of attack is referred to as a *return-into-libc* attack [Wojtczuk 1998]. A related attack is called *return-oriented programming* [Shacham 2007], 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 [Anisimov ; skape and Skywing 2005].

Control Flow Integrity (CFI, [Abadi et al. 2005]) is not a widely deployed countermeasure, but it 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 the 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 [Younan et al. 2006]. 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 [Cowan et al. 1998], pointer encryption [Cowan et al. 2003] and CFI [Abadi et al. 2005]. These countermeasures will be further explained in Section 7.

This section introduces the *Code Pointer Masking (CPM)* countermeasure, located 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*. Even though an application may still have memory management vulnerabilities, it becomes much harder for the attacker to exploit them in a way that might be useful.

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.

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. In a prototypical function epilogue, the return address is first 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.

CPM mitigates this attack by inserting a masking instruction in between. 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.

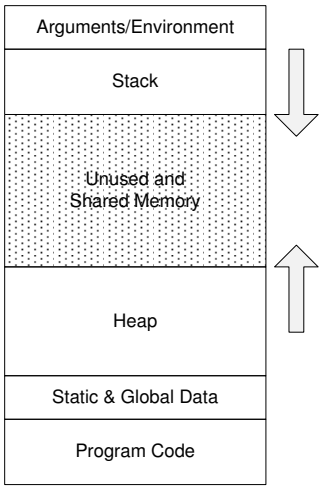

Fig. 2. Stack, heap and program code memory layout for a Linux application

Example The following example illustrates address masking for the Linux operating system. It should be noted that on other operating systems, the countermeasure may need different masking operations than those used here, but the concept remains the same on any system.
As shown in Figure 2, program data, heap and stack are located above the program code in memory. For illustrative purposes, the program code is assumed to range from 0x00000000 to 0x0000FFFF, thus stack and heap are located on memory addresses larger than 0x0000FFFF.
For each function in the application, the epilogue is changed from fetching the return address and jumping to it, to fetching the return address, performing an AND operation with the mask 0x0000FFFF on the return address, and then jumping to the result. Memory addresses that point to the stack or heap will have at least one bit of the two most significant bytes set. These bits will be cleared, however, because of the AND operation. As a result, before the memory address reaches the JUMP instruction, it will be properly sanitized to ensure that it can only point to a location within the code segment.

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. If the attacker is able to modify the return address of the function, he is only able to jump to existing program code.

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 can be 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 depend on the processor 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 and x64 architectures this *is* a good solution.

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 systems, 0x7FNNNNNNNNNN on 64-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 been executed already 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.

Like many other countermeasures, CPM has to be applied to the entire program to ensure maximum protection. If there are unprotected code pointers left in the program code (for example, by statically or dynamically linking with a library that has not been protected with CPM), an attacker might still find a way to exploit these code pointers. However, even in the presence of unprotected code, CPM will still greatly

increase the security of the application by significantly reducing the attack surface of the application.

4. IMPLEMENTATION

This section describes the implementation of the CPM prototype for the ARM and x64 architecture. The ARM prototype is implemented in gcc-4.4.0 and binutils-2.20 for Linux, whereas the x64 prototype is implemented in gcc-4.6.0 and binutils-2.21 for Linux.

For GCC, the machine descriptions are changed to emit the masking operations during the conversion from RTL[5] to assembly. The implementations provide the full CPM protection for return addresses, function pointers, GOT entries, and the other code pointers.

4.1. General Overview

The prototypes are implemented as a multistage transformation process on the program binary code. The process consists of five steps.

In the first step, the source code of the application is parsed and interpreted using the 'C Intermediate Language (CIL)[6]' tool. Using this tool, the code is searched for *address of* operators on functions. When an application wants to use a function pointer, it must first acquire a pointer to the function it wants to call. This is done using these *address of* operators, so if we know all the functions for which the address is taken, we know all possible destinations for function pointers. In addition, the tool also creates for every function a list of caller functions. These lists are needed to calculate the return address masks.

The second step uses a modified GCC compiler to compile the application into binary code. This step emits masking instructions for a number of code pointers, such as the return address and function pointer calls. Section 4.2 shows how the code from function epilogues is instrumented to protect return addresses. Note, however, that after this step the masking instructions still use dummy masks that in essence do nothing.

The third step links the object files together and emits a binary file. The linker also inserts the PLT code as described in Section 4.3, which ensures that the entries in the GOT will always point to the library code section.

Step four is where the actual CPM magic happens. The binary produced in the previous step is disassembled and inspected. A script searches for the dummy instructions that were inserted in step two, calculates the applicable masks and inserts the mask into the executable. After this step, the resulting binary is almost fully protected.

In the final step, step five, the protection for long jumps is added. In the prototypes we addressed this by building our own libraries with a protected longjmp implementation. When the binaries are being run, we use the LD_PRELOAD environment variable to force the application to use our longjmp implementation instead of the one from the standard C library.

The above steps can be consolidated into the compiler and linker. However, this is a non-trivial task and does not offer any advantage except better usability of the prototypes.

4.2. 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/x64. But instead of only pop-

[5]RTL or *Register Transfer Language* is one of the intermediate representations that is used by GCC during the compilation process.
[6]http://cil.sourceforge.net/

ping 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 shows what this epilogue looks like.

Listing 1: 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 2: 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
```

Listing 2 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 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.

The x64 prototype uses a similar approach, but uses an AND instruction instead. Listing 3 shows the replacement instructions for the return instruction. The return address is popped from the stack and masked using an `AND` instruction. The processor is then instructed to jump to the result. On the x64 prototype, register *r10* is used exclusively to perform CPM masking operations.

Listing 3: A CPM function epilogue on x64.

```
popq    %r10
andq    $0xNNNNNNNNNNNNNNNN, %r10
jmpq    *%r10
```

4.3. Procedure Linkage Table Entries

As explained in Section 3.5, applications use a structure called *the global offset table* in order to enable the dynamic loading of 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 4: A PLT entry on ARM that does not perform masking.

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

Listing 4 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 5 shows the modified PLT entry.

Listing 5: A PLT entry on ARM 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 library memory range (0x4NNNNNNN on 32-bit Linux systems). 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*.

The x64 implementation of a PLT entry is shown in Listing 6. It consists of a jump that reads the requested address from the GOT and jumps to it. The values in the GOT are initialized in such a way that they point to the `PUSHQ` instruction following the jump. This makes sure that when the corresponding library has not been loaded into memory yet, the jump simply falls through to the next instruction.

The `PUSHQ` instruction pushes a value onto the stack that uniquely identifies the PLT entry that is being executed. After the push, a jump is performed to the method that loads the library from disk. This method uses the index on the stack to determine which library should be loaded.

Listing 6: An unprotected PLT entry on x64.

```
jmpq   *name@GOTPC(%rip)
pushq  0xNNNNNNNN
jmp    0xNNNNNNNN
```

Instead of immediately jumping to the address in the GOT, PLT entries that are used in the x64 implementation of CPM will first make sure that the address points to the library address space. On 64-bit Linux, libraries are loaded in the 0x00007FNNNNNNNNNN memory region. Listing 7 shows a modified PLT entry. The first instruction copies the address in the GOT into the reserved register *r10*. The address is immediately copied into scratch register *r11*. The value in *r11* is then shifted right for 40 bits, preserving the 8 most significant bits of the address[7]. These bits are compared to the value 0x7F. If the address in the GOT points to a function in a library, the eight most significant bits of the address should be equal to 0x7F. If this is not the case, we assume that the library has not been loaded into memory yet. If the `CMP` instruction sets the *equals* flag, the `JMPQ` instruction is executed and the processor jumps to the address stored in *r10*. If this flag is not set, the processor jumps to the `PUSHQ` instruction, and the library loading process continues as in an unmodified PLT entry.

One problem with 64-bit Linux versions is that the stack and libraries share the common address space 0x00007FNNNNNNNNNN. In order to make sure that attackers cannot inject code on the stack, the stack must be moved to another address (which

[7]In current x64 processor implementations, only 48 bits are addressable. In user space, the top 16 bits will always be 0.

Listing 7: A protected PLT entry on x64.

```
mov     name@GOTPC(%rip), %r10
mov     %r10,%r11
shr     $0x28,%r11
cmp     $0x7f,%r11
jne     .stub
jmpq    *%r10
.stub:
pushq   0xNNNNNNNN
jmp     0xNNNNNNNN
```

is a small change in the existing ASLR implementation that already changes the memory range of the stack). Alternatively, making the stack non-executable also solves the problem; this last option is already implemented in many operating systems.

4.4. 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. On ARM, 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.

On x64, the implementation of longjmp is modified to move the stored return address into reserved register *r10* instead of *rdx* and the jump at the end of the function is preceded with a masking `AND` instruction.

4.5. 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. In the SPEC benchmark, only one application had one method with this behavior. The method was used to serve as a comparison function for the quicksort implementation of libc.

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.

The second scenario is when an application generates code (e.g. JIT compilers) or gets a code pointer from a library (e.g. by using a method like *GetProcAddress* on Windows), 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, where function pointers could be marked to alter their masking behavior. For example, a function pointer that is used to jump to JITted code can be marked to only allow jumps to the heap. Likewise, a function pointer that is used to jump to libraries can be marked to

only allow jumps to the library address space (similar to the protection of PLT entries, as discussed in Section 4.3). 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 and the performance overhead of our ARM prototype, we ran the SPEC2000 benchmark [Henning 2000] with our countermeasure and without. All tests were run on a single machine with an ARMv7 processor running at 800MHz, 512Mb RAM, running Ubuntu Linux with kernel 2.6.28. The x64 prototype was benchmarked using the SPEC2006 benchmark on a machine with an Intel Core2 Duo processor running at 2.4GHz, 2Gb RAM, running Ubuntu Linux with kernel 2.6.35.

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 I: Benchmark results of the CPM countermeasure on the ARM architecture

SPEC CPU2006 Integer benchmarks			
Program	**GCC (s)**	**CPM (s)**	**Overhead**
400.perlbench	514	524	+1.95%
401.bzip2	698	716	+2.58%
403.gcc	531	550	+3.58%
429.mcf	646	647	+0.15%
445.gobmk	670	707	+5.52%
456.hmmer	576	587	+1.91%
458.sjeng	739	775	+4.87%
462.libquantum	1178	1175	-0.25%
464.h264ref	994	1015	+2.11%

Table II: Benchmark results of the CPM countermeasure on the x64 architecture

All C programs in the SPEC CPU2000 and CPU2006 Integer benchmarks were used to perform these benchmarks. Table I and Table II contain the runtime in seconds when compiled with the unmodified GCC, 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. Table I does not contain results for VORTEX, because it does not work 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 benchmarks add up to a total of about 1,500,000 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 4.5).

5.2. Security Evaluation

As a first step in the evaluation of CPM, two field tests were performed with the prototype. Existing applications and libraries that contain vulnerabilities[8] were compiled with the new countermeasure. For demo purposes, we wrote exploits for the original vulnerable code and concluded that CPM successfully mitigated the attacks by raising the bar to 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.

5.2.1. CPM versus Widely Deployed Countermeasures. This section compares CPM with the widely deployed countermeasures that were introduced in Section 2.2. Some functionality of CPM overlaps with parts of these existing defenses, but CPM also protects scenarios where (the combination of) these countermeasures do not protect applications. This section starts by discussing the similarities in protection, and then goes on to explain how CPM protects against scenarios that are not protected by the current state of practice.

Stack canaries were introduced to protect return addresses against stack-based buffer overflows. They are successful against this particular scenario where an attacker overwrites the return address, but as explained in Section 2.2 there are a number of ways to get around this protection. CPM overlaps with this countermeasure, in the sense that it also protects return addresses. However, unlike stack canaries, CPM protects return addresses against *any* overwrite, including indirect pointer overwriting.

Address Space Layout Randomization makes it more difficult for attackers to guess addresses of interesting data structures, such as the stack, heap and libraries. CPM overlaps in part, because it protects against code injection attacks on the stack or heap. However, CPM prevents them by disallowing an application to jump to an address on one of these data structures instead of obfuscating the address. CPM also ensures that normal application functions cannot return to the library address space.

Finally, non-executable memory marks data structures as non-executable, preventing attackers from directly injection code into these structures. CPM also protects

[8]CVE-2006-3459 and CVE-2009-0692

	ProPolice	ASLR	NX[1]	Combination[2]
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 that all memory, except code and library memory, is marked as non-executable. On Linux, this depends on the distribution, and is often not the case.
[2] = This is the combination of the ProPolice, ASLR and No-Execute countermeasures, as deployed in modern operating systems.

Table III: 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).

against these attacks, but does it by disallowing an application to jump to an address in data space.

The aforementioned countermeasures suffer from a number of problems that still make them vulnerable to exploitation. Table III shows how each of these countermeasures (and also their combination) can be broken by using multiple attack techniques. The rows in the table represent the different vulnerabilities that allow code injection attacks, and the columns represent the countermeasures.

Each cell in the table contains the (combinations of) attack techniques (see Section 2.2) that can be used to break the security of the countermeasure(s). The 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 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 [Strackx et al. 2009], 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 [Shacham 2007] 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 I 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%). Table IV shows average bit mask sizes and jump surface results of an additional set of popular C applications that were randomly selected from SourceForge.net and Github.com. The average mask size is similar to the mask sizes that were seen in the SPEC benchmark applications, but because many of these applications are quite large, the average jump surface decreases significantly in some cases compared to the SPEC applications. Due

Program	Avg. Mask size	Jump surface
git	16.31 bits	3.93%
httpd	10.09 bits	0.13%
vlc	12.56 bits	0.53%
quakespasm	11.13 bits	0.23%
xbmc	11.51 bits	0.02%
flex	14.78 bits	4.54%
wireshark	11.07 bits	0.14%

Table IV: Average bit mask sizes and jump surface results of an additional set of applications.

to the increasing importance of return-oriented programming, Section 5.2.4 will give a detailed evaluation of CPM against RoP attacks.

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 unlikely as explained in the previous paragraph and Section 5.2.4.

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 [Erlingsson 2007], where an attacker overwrites application data and no code pointers, are not protected against by CPM.

5.2.2. Combining CPM with Other Countermeasures. CPM can be combined with existing countermeasures. In particular, stack canaries and non-executable memory are 100% compatible with the prototype. CPM does not depend for its security on one of these countermeasures, but because CPM — like the other widely deployed countermeasures — does not offer a 100% security guarantee, combining all these countermeasures together is preferred from a defensive point of view.

The story for ASLR is a little bit more complicated. The CPM prototype is compatible with ASLR, as long as everything except the program code itself is randomized. The prototype precomputes the masks at link time, so the memory location of methods cannot be altered later on. This implies that the executables must always be loaded at the same location in memory (and thus counters the premise of ASLR). However, the stack, heap and libraries *can* be randomized without breaking the protections of CPM, so combining CPM with a weaker version of ASLR is still possible.

This begs the question whether the prototype can be modified to support a full ASLR implementation (with randomized program code). We leave the implementation to fu-

ture work, but yes, such an implementation is possible. The biggest change would be to compute the masks at runtime instead of at link time.

The steps described in Section 4.1 will not change, but in the implementation that supports randomized code step four will have to be done last. Instead of calculating the masks right after the binary has been linked, the masks will have to be calculated by the loader. The loader first decides where the program code will be loaded in memory (using a randomized location), will then load the code into memory, and then a new step should be added where the loader modifies the code to enable the CPM protections. These modifications are the same modifications that the current prototype does in step four: find the dummy instructions in the binary, calculate the applicable masks and insert the mask into the executable.

It is important to note that even through the loading of the executable may be slowed down by this process (in the order of maybe a few milliseconds), it will not have an impact on the performance of the application when it is executing. All mask updates are done *before* the application starts executing.

5.2.3. 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 [Younan et al. 2012; Abadi et al. 2005]. 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. We found that the list of types of code pointers we introduced in Section 3 was complete.

Masking is non-bypassable. Every computed jump that is emitted by the compiler will be masked, but the masking process should not be subverted. 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. We ensure this by reserving a dedicated register 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*, etc. 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. As the numbers for 'jump surface' in Table I show, this is a significant improvement over unprotected code.

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 guaran-

tees 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.

5.2.4. CPM and Return-oriented Programming. A relatively new attack technique called *return-oriented programming (RoP)* has received a lot of attention from the research community in recent years. The basic idea of this technique is that an attacker will first identify a set of short instruction sequences in the program code that end in a computed jump (called *gadgets*). In a second step, the attacker will then combine these gadgets to perform useful computations (and eventually exploit the application). Shacham proved in [Shacham 2007] that the set of gadgets obtained from libc is Turing Complete.

Although CPM is not specifically tailored for protection against RoP attacks, the attacks do rely on modifying code pointers and thus will encounter at least one masking instruction.

First of all, it is important to note that RoP is an attack technique, not a vulnerability. Hence, it can only be used *after* an attacker has successfully compromised an application. Therefore, the attacker will need an exploit to jump to the first gadget, but this (computed) jump will always be masked by CPM. However, due to the nature of the masking process, there is a chance that the attacker might still be able to jump to a gadget.

For CPM on the ARM architecture (for which it was originally developed), this is not a major problem. Because gadgets end in a computed jump, every gadget will be masked. This means that, even if the attacker passes the first hurdle to get to the first gadget, he will then need to circumvent the masking again to go to the second gadget, etc. As the number of gadgets that are required in the exploit code grows, it rapidly becomes improbable that the attack will be possible.

Attackers on architectures that support executing unaligned instructions (like x64 or x86) can also make use of unaligned gadgets. These gadgets start with one or more instructions that are executed in an unaligned fashion (essentially by jumping into the middle of an aligned instruction). This allows attackers to use instruction sequences that the compiler never specifically intended to embed in the program code. This is important for CPM, because the compiler might emit unintended computed jumps that are not preceded by masking instructions. If an attacker can reach one of these gadgets, he might be able to start a sequence of gadgets that bypasses the masks.

While it is true that CPM on ARM offers better security guarantees against RoP attacks than CPM on x64, it must be noted that even on the x64 architecture the bar against RoP attacks is raised (albeit less than on ARM). The attacker will never be able to avoid the first masking operation when he wants to jump to the first gadget.

[9]CFI has an overhead of up to 45%, with an average overhead of 16% on the Intel x86 architecture. The results are difficult to compare, because our results are measured on the ARM and x64 architectures, however because CFI does a memory lookup for every computed jump it encounters, where CPM does not, it is fair to say that CPM will be much faster on any architecture.

Hence, there is a good chance that the attacker will not be able to reach any of the gadgets he needs to circumvent the masking operations. All the aligned gadgets are still protected by masking operations, so he loses a substantial amount of freedom to write a successful exploit. Furthermore, an approach as defined in [Onarlioglu et al. 2010] can be adopted to enforce aligned execution with only a minor impact on performance.

Exploit writers have built tools to automatically search for usable gadgets in binaries. In order to obtain some statistics, we tested one such tool on a large application. A number of free tools exist for the x86 architecture, but we could not immediately find similar tools for the x64 or the ARM architecture. That is why we implemented a third partial CPM prototype for x86, with the sole purpose of gathering these statistics. The prototype supports masking return addresses.

The freely available ROPGadget[10] tool was used to retrieve all the gadgets from the binary of the Apache web server version 2.4.2. We chose the Apache web server, because it is commonly used, has a large attack surface, a significant code base, and is written in pure C.

ROPGadget identified 4238 gadgets in the binary. Out of all the functions that were present in the Apache binary, 57.75% were prevented to return to *any* gadget by the CPM masking procedure. Over 70% of functions could return to only 10 or less gadgets. Interestingly (but somewhat expected), there is a spike on the other side of the spectrum, with 23 functions (2.33% of the total) being able to jump to any gadget.

These statistics tell us that CPM significantly raises the bar. In fact, for more than half of the functions, the mask is perfect (from a defense point of view). However, a small fraction of the functions have the worst possible mask. These functions are functions that are used throughout the entire application, and consist mostly of memory management functions. In Section 6, we discuss future work that can be used in this case to improve the masks and make them arbitrarily precise.

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 (e.g. 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 stack canaries, non-executable memory and ASLR[11]. Adding CPM to the mix of existing protections significantly raises 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 multiple architectures. Two prototypes have been implemented on vastly different processor architectures, that differ in ar-

[10] `http://shell-storm.org/project/ROPgadget/`

[11] When everything except the program code is randomized. Support for randomized program code is future work.

chitecture type (RISC vs. CISC), instruction set (ARM vs. x64), and word size (32-bit vs. 64-bit). Similar performance characteristics are observed.

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.

We are also investigating a new mask optimization technique where we detect when a function's mask it too imprecise, and duplicate the function code in the binary. Imprecise masks are typically caused by functions that have many callers, so by duplicating the function we can distribute the callers over two (or more) different copies of the function. In this way, we can arbitrarily increase the mask precision at the cost of using more memory to store program code. An initial prototype where functions were duplicated for every caller (essentially generating the perfect mask for every function) proved very promising. The SPEC benchmark showed that it had an average overhead of almost 0%, at the cost of a substantial code increase.

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 [Younan et al. 2010a].

Bounds checkers Bounds checking [Kendall 1983; Steffen 1992; Austin et al. 1994; Jones and Kelly 1997; Lhee and Chapin 2002; Oiwa et al. 2002; Patil and Fischer 1997] 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 [Akritidis et al. 2009; Younan et al. 2010b] have improved performance somewhat, but still do not protect against dangling pointer vulnerabilities, format string vulnerabilities, and others.

Safe languages Safe languages are languages where it is generally not possible for any known code injection vulnerability to exist as the language constructs prevent them from occurring. A number of safe languages are available that will prevent these kinds of implementation vulnerabilities entirely. There are safe languages [Jim et al. 2002; Necula et al. 2002; Larus et al. 2004; Kowshik et al. 2002] that remain as close to C or C++ as possible, and are generally referred to as safe dialects of C. While some safe languages [Condit et al. 2003; Xu et al. 2004] try to stay compatible with existing C programs, use of these languages may not always be practical for existing applications.

Probabilistic countermeasures Many countermeasures make use of randomness when protecting against attacks. Many different approaches exist when using randomness for protection. Canary-based countermeasures [Cowan et al. 1998; Etoh and Yoda 2000; Krennmair 2003; Robertson et al. 2003] 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 (or memory corruption) has been detected. Memory-obfuscation countermeasures [Cowan et al. 2003; Bhatkar and Sekar 2008] encrypt important memory locations using random numbers. Mem-

ory layout randomizers [The PaX Team ; Bhatkar et al. 2003; Xu et al. 2003; Bhatkar et al. 2005] randomize the layout of memory by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers [Barrantes et al. 2003; Kc et al. 2003] 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 [Strackx et al. 2009]. 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. This makes it harder for an attacker to overwrite this information using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function [Chiueh and Hsu 2001]. 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 [Younan et al. 2006], 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 differentiates between memory that contains code and memory that contains data. Data memory is marked as non-executable [The PaX Team]. 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.

Removal of Gadgets A number of countermeasures are aimed specifically at the prevention of return-oriented programming attacks. Early work focussed on protecting the stack [Davi et al. 2011] or keeping track of the ratio of return instructions [Chen et al. 2009; Davi et al. 2009]. However, these solutions do not adequately protect against RoP attacks; in particular, none of these countermeasures prevents an attacker from using gadgets that do not use the return instruction.

Other solutions modify the compilation process to ensure that no (usable) gadgets are present in the resulting binary. G-Free [Onarlioglu et al. 2010] replaces so-called free-branch instructions by a small set of instructions that verify the code pointer. However, G-Free uses a secret value to protect code pointers, and can be exploited by information leakage vulnerabilities [Strackx et al. 2009]. Furthermore, the performance of G-Free is unclear. The authors report a performance close to that of CPM, however their benchmarks for application-wide overhead consist of IO-bounded applications that are not control flow intensive.

Work by Li [Li et al. 2010] prevents the existence of gadgets in the kernel. Due to the fact that this countermeasure is geared towards kernel-level protection and that it only protects against return-oriented programming attacks, it is difficult to compare it with CPM. However, the authors report benchmark results (with full protection) between 5.78% and 17.32%, well over the performance overhead of CPM.

Software Fault Isolation Software Fault Isolation (SFI) [Wahbe et al. 1993; Mccamant and Morrisett 2006] 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 [Kiriansky et al. 2002] is a technique that monitors the execution of a program and will disallow control-flow transfers [12] that are not considered safe. An example of a use for shepherding is to enforce return instructions to only return to the instruction after the call site. The proposed implementation of this countermeasure is done using a runtime binary interpreter. As a result, the performance impact of this countermeasure is significant for some programs, but acceptable for others.

Control-flow integrity (CFI, [Abadi et al. 2005]), as discussed in Section 5.2.3, is also a countermeasure that is classified as an execution monitor. Control-Flow Locking (CFL, [Bletsch et al. 2011]) is an interesting variation of CFI. By relaxing the constraint that an attack should immediately be detected, CFL optimizes the performance of CFI. CFL limits itself to the protection against return-oriented programming. It would not, for example, stop a direct code injection attack on the stack or heap, like CPM would[13]. Although the performance of CFL is good, CPM has less than half the overhead of CFL and is in some cases over 10x faster.

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.

ACKNOWLEDGMENTS

The work reported on in this paper builds on joint research performed with and supported by DOCOMO Euro-Labs in 2008-2009.

The authors wish to thank Raoul Strackx for his interesting input during the development of the x64 prototype.

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund K.U.Leuven, and by the EU-funded FP7-project NESSoS.

[12] Such a control flow transfer occurs when e.g., a *call* or *ret* instruction is executed.

[13] The authors of CFL assume the stack and heap will always be non-executable, which would stop a direct code injection attack, but it is our opinion that having multiple defenses in place against an attack is a better option.

REFERENCES

ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, Alexandria, Virginia, U.S.A., 340–353.

AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*. Montreal, QC.

ANISIMOV, A. Defeating Microsoft Windows XP SP2 heap protection and DEP bypass.

AUSTIN, T. M., BREACH, S. E., AND SOHI, G. S. 1994. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM.

BARRANTES, E. G., ACKLEY, D. H., FORREST, S., PALMER, T. S., STEFANOVIĆ, D., AND ZOVI, D. D. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*. ACM, 281–289.

BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association.

BHATKAR, S. AND SEKAR, R. 2008. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*. Lecture Notes in Computer Science Series, vol. 5137. Springer.

BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*. USENIX Association, Baltimore, MD.

BLETSCH, T., JIANG, X., AND FREEH, V. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 353–362.

CHEN, P., XIAO, H., SHEN, X., YIN, X., MAO, B., AND XIE, L. 2009. Drop: Detecting return-oriented programming malicious code. *Information Systems Security*, 163–177.

CHIUEH, T. AND HSU, F. H. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*. IEEE Computer Society, IEEE Press, Phoenix, Arizona, USA, 409–420.

CONDIT, J., HARREN, M., MCPEAK, S., NECULA, G. C., AND WEIMER, W. 2003. CCured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM, San Diego, California, U.S.A., 232–244.

COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. 2003. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*. USENIX Association, 91–104.

COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, San Antonio, Texas, U.S.A.

DAVI, L., SADEGHI, A., AND WINANDY, M. 2009. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*. ACM, 49–54.

DAVI, L., SADEGHI, A., AND WINANDY, M. 2011. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 40–51.

ERLINGSSON, U. 2007. Low-level software security: Attacks and defenses. Tech. Rep. MSR-TR-2007-153, Microsoft Research.

ETOH, H. AND YODA, K. 2000. Protecting from stack-smashing attacks. Tech. rep., IBM Research Divison. June.

GADALETA, F., YOUNAN, Y., AND JOOSEN, W. 2010. Bubble: A javascript engine level countermeasure against heap-spraying attacks. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSOS)*.

HENNING, J. L. 2000. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer 33,* 7, 28–35.

JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*. USENIX Association, Monterey, California, U.S.A., 275–288.

JONES, R. W. M. AND KELLY, P. H. J. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*. Number 009-02 in Linköping Electronic Articles in Computer and Information Science. Linköping University Electronic Press.

KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*. ACM, Washington, D.C., U.S.A., 272–280.

KENDALL, S. C. 1983. Bcc: Runtime checking for C programs. In *Proceedings of the USENIX Summer 1983 Conference*. USENIX Association, Toronto, Ontario, Canada, 5–16.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. 2002. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, San Francisco, California, U.S.A.

KOWSHIK, S., DHURJATI, D., AND ADVE, V. 2002. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers Architecture and Synthesis for Embedded Systems*. Grenoble, France, 288–297.

KRENNMAIR, A. 2003. ContraPolice: a libc extension for protecting applications from heap-smashing attacks.

LARUS, J. R., BALL, T., DAS, M., DELINE, R., FÄHNDRICH, M., PINCUS, J., RAJAMANI, S. K., AND VENKATAPATHY, R. 2004. Righting software. *IEEE Software 21,* 3, 92–100.

LHEE, K. S. AND CHAPIN, S. J. 2002. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, San Francisco, California, U.S.A., 81–90.

LHEE, K. S. AND CHAPIN, S. J. 2003. Buffer overflow and format string overflow vulnerabilities. *Software: Practice and Experience 33,* 5, 423–460.

LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. 2010. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*. ACM, 195–208.

MCCAMANT, S. AND MORRISETT, G. 2006. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*. USENIX Association, Vancouver, British Columbia, Canada.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. National vulnerability database statistics. http://nvd.nist.gov/statistics.cfm.

NECULA, G., MCPEAK, S., AND WEIMER, W. 2002. CCured: Type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, Portland, Oregon, U.S.A., 128–139.

OIWA, Y., SEKIGUCHI, T., SUMII, E., AND YONEZAWA, A. 2002. Fail-safe ANSI-C compiler: An approach to making C programs secure: Progress report. In *Proceedings of International Symposium on Software Security 2002*.

ONARLIOGLU, K., BILGE, L., LANZI, A., BALZAROTTI, D., AND KIRDA, E. 2010. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 49–58.

PATIL, H. AND FISCHER, C. N. 1997. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice and Experience 27,* 1.

PHILIPPAERTS, P., YOUNAN, Y., MUYLLE, S., PIESSENS, F., LACHMUND, S., AND WALTER, T. 2011. Code pointer masking: Hardening applications against code injection attacks. In *Proceedings of the Eighth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. Amsterdam, The Netherlands.

ROBERTSON, W., KRUEGEL, C., MUTZ, D., AND VALEUR, F. 2003. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*. USENIX Association.

SHACHAM, H. 2007. 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*. ACM, ACM Press, Washington, D.C., U.S.A., 552–561.

SHACHAM, H., PAGE, M., PFAFF, B., GOH, E. J., MODADUGU, N., AND BONEH, D. 2004. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*.

SKAPE AND SKYWING. 2005. Bypassing windows hardware-enforced data execution prevention. *Uninformed 2*.

STEFFEN, J. L. 1992. Adding run-time checking to the portable C compiler. *Software: Practice and Experience 22,* 4, 305–316.

STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. 2009. Breaking the memory secrecy assumption. In *Proceedings of the European Workshop on System Security (Eurosec)*. Nuremberg, Germany.

THE PAX TEAM. Documentation for the PaX project.

WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. 1993. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles*. ACM, Asheville, North Carolina, U.S.A.

WOJTCZUK, R. 1998. Defeating solar designer non-executable stack patch. Posted on the Bugtraq mailinglist.

XU, J., KALBARCZYK, Z., AND IYER, R. K. 2003. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*. IEEE Computer Society, IEEE Press, Florence, Italy.

XU, W., DUVARNEY, D. C., AND SEKAR, R. 2004. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, ACM Press, 117–126.

YOUNAN, Y., JOOSEN, W., AND PIESSENS, F. 2010a. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Computing Surveys*.

YOUNAN, Y., JOOSEN, W., AND PIESSENS, F. 2012. Runtime countermeasures for code injection attacks against c and c++ programs. *ACM Computing Surveys 44,* 3, 17:1–17:28.

YOUNAN, Y., PHILIPPAERTS, P., CAVALLARO, L., SEKAR, R., PIESSENS, F., AND JOOSEN, W. 2010b. Paricheck: An efficient pointer arithmetic checker for c programs. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, Bejing, China.

YOUNAN, Y., POZZA, D., PIESSENS, F., AND JOOSEN, W. 2006. Extended protection against stack smashing attacks without performance loss. In Proceedings of the Twenty-Second Annual Computer Security Applications Conference (ACSAC '06). *Proceedings of the Twenty-Second Annual Computer Security Applications Conference (ACSAC'06)*, 429–438.

Received January 2012; revised May 2012, September 2012; accepted January 2013