Scientific Research Publishing

Journal of Information Security, 2017, 8, 42-55
http://www.scirp.org/journal/jis
ISSN Online: 2153-1242
ISSN Print: 2153-1234

Program Counter Encoding for ARM® Architecture

Seho Park, Yongsuk Lee, Gyungho Lee

Department of Computer Science and Engineering, Korea University, Seoul, Korea
Email: ghlee@korea.ac.kr

How to cite this paper: Park, S., Lee, Y. and Lee, G. (2017) Program Counter Encoding for ARM® Architecture. *Journal of Information Security*, **8**, 42-55.
http://dx.doi.org/10.4236/jis.2017.81004

Received: December 9, 2016
Accepted: January 7, 2017
Published: January 10, 2017

Abstract

ARM® is the prevalent processor architecture for embedded and mobile applications. For the smartphones, it is the processor for which software applications are running, whether the platform is with Apple's iOS or Google's Android. Software operations under these platforms are prone to semantic gap, which refers to potential difference between intended operations described in software and actual operations done by processor. Attacks that compromise program control flows, which result in these mantic gaps, are a major attack type in modern software attacks. Many recent software protection schemes for servers and desktops focus on protecting program control flows, but there are little protection tools available for protecting program control flows of mobile applications for ARM processor architecture. This paper uses a program counter (PC) encoding technique (PC-Encoding) to harden program control flows under ARM processor architecture. The PC-Encoding directly encodes control flow target addresses that will load into the PC. It is simple and intuitive to implement and incur little overhead. Encoding the control flow target addresses can minimize the semantic gap by preventing potential compromises of the control flows. This paper describes our efforts of implementing PC-Encoding to harden portable binaries in ELF (Executable and Linkable Format).

Keywords

Compiler, Control-Flow Integrity, PC-Encoding, Software Security, ARM®

1. Introduction

Software security has become an increasingly important concern with the prevalent use of the Internet. With the looming popularity of the Internet of Things (IoT), this concern is becoming even more prevalent with respect to every aspect of modern life. A variety of security techniques have been researched and devel-

oped at the software level, and some of these have been adopted in practice. There are many researches focusing on control flow hardening for server and desktop environments, but there are relatively fewer studies about hardening the program control flow for embedded and mobile devices, *i.e.*, tend to concentrates on Intel's x86 architecture. Attackers, however, are continually capable of finding vulnerabilities, and while the success of the attacks is, more or less, dependent on the same techniques of the past, the incidence rates that are being documented make a cause for a greater alarm; furthermore, the number of vulnerabilities that are utilized for the attacks is increasing, and the risks inherent in the security vulnerabilities have become even more serious.

A major portion of the vulnerabilities allows a memory overwrite, which in turn causes a program control transfer that is contrary to the intentions of the original programmer. The existence of this semantic gap, *i.e.*, the potential difference between the intended operations described in software and the actual operations performed by a processor, is from the existence of a memory overwrite vulnerability.

There have been a variety of techniques for the prevention of arbitrary memory overwrites, along with schemes for the prevention of the activation of injected attack codes, e.g., the numerous variations of DEP (data-execution prevention) [1] [2] [3] [4]; however, the incidence of software attacks that utilize the memory overwrite is ongoing. ROP (return-oriented programming) [5] is a recent attack method that allows a code-reuse attack without an injection of attack code; the code reuse attack utilizes a memory overwrite to activate the preexistent code in memory to start an attack [6] [7] [8] [9] [10]. Note that a memory overwrite is essential for not only traditional code-injection attacks [11], but also for recent code-reuse attacks that seek to divert program control flows from the intended operations that are described in software [12].

To protect the control flow from the software attacks, we need to enforce the integrity of the control flows through an examination of the destinations of all the control flow transfer instructions to confirm their legitimacy. Conceptually, one can generate a control-flow graph (CFG) for a program under protection and check at runtime to see whether the program execution actually follows the CFG. Even without a contemplation of the accuracy of the CFG, a couple of issues arise from this conceptual control flow validation scenario such as CFG granularity and the representation and storage of the CFG. System-call-level CFG was utilized during the early days of the technology to reduce the overhead of the CFG representation and storage, and also to enable access to the CFG at runtime [13]; however, such a coarse-grain CFG often fails to catch a compromised control flow because many hundreds or thousands of instructions can exist between the system calls.

The CFI (control-flow integrity) [14] [15] [16] is one of the early proposals for checking and validating each control-flow transfer at the fine-grain machine instruction level, whereby the destinations of all the indirect branch instructions are forcibly checked. The full CFG, comprising a pair of IDs, the branch-instruction

address ID and its target address ID, for all the indirect branch instructions is generated by a binary patch, and each control-flow transfer is checked at run time per the CFG. Even if the CFG is limited to statically linked procedures, however, it is still very difficult to draw a fully accurate CFG in practice. Also, there is a not so trivial overhead for representing and accessing the CFG. CFI implementations in general utilize the fact that a code region is safe from a memory overwrite, and the instruction addresses and target addresses are also coalesced together to make the CFG representation compact and fast to access. There have also been some CFI-implementation methods with a looser notion of CFG for the ease of the deployment and a lesser performance overhead [17] [18]; although, the coalesced address ID and the looser notion of the CFG will result in more of the semantic-gap potential, thereby reducing the robustness of the protection [19].

PC-encoding can protect the general indirect-branch instructions as the CFI does, but without the requirement of CFG generation [20] [21] [22]. Instead, PC encoding encodes the target address at its definition and requires a proper decoding of the target address for a legitimate control-flow transfer. PC encoding has little room for the semantic gap because the CFG is not generated as an approximation of what is described in the program, but the program itself acts as the CFG; furthermore, unlike randomizing the instruction set architecture [23] or the program code [24], checking the legitimacy of each control-flow transfer requires only a couple of machine instructions without a need to access the memory for the CFG. PC encoding ensures a minimal performance degradation.

This paper describes our efforts to implement PC-encoding for hardening the portable ELF (executable and linkable format) binary. ARM® processor stores the return address at *lr* (link register) when it uses a *bl* (branch link) instruction to call a function. In the function prologue, PC encoding encrypts the *lr* and *fp* (frame pointer register) before pushing into the stack. At the function epilogue, the *fp* and *pc* (program-counter register) are restored after decrypted from the stack. Our LLVM (low-level virtual machine)-based PC-encoding compiler provides control flow protection without significant overhead for programs that run under the ARM®-processor architecture [25]. Compared to other software based control flow hardening schemes, PC-encoding makes a suitable fit for the relatively simple architectural environment of ARM®.

The remainder of the paper is organized as follows: Section 2 discusses the basics of PC encoding. The process of applying PC encoding for the *ARM-Linux-ELF* binary is described in Section 3. Section 4 presents the performance-test results from the Gem5 simulator. Also discussed is a comparison of PC-encoding with a typical CFI implementation. This paper concludes in Section 5 with a discussion of the limitations of our current PC-encoding implementation for ARM® processors.

2. Backgrounds

A program control flow is dictated by program data that is loaded to the program

counter at runtime, which we call *PC-bound data*. The basic idea of PC encoding involves the checking of the integrity of the PC-bound data. PC encoding ensures the integrity of the program control flow by protecting the PC-bound data; it encodes the PC-bound data at their definition and during its use, decoding is performed with a secret key. PC encoding provides a sound solution for hardening the program control-flow for embedded platforms because it only incurs a small performance penalty. Data encoding/decoding computation can also be simplified, whereby simple encoding/decoding operations that can be completed in one or two cycles may be employed, e.g., exclusive-or. Also, a compatibility issue is non-existent because the memory layout does not need to be changed; moreover, cooperation with a non-hardened binary is feasible.

Software operations are prone to the semantic gap, which refers to potential difference between intended operations described in software and actual operations done by the processor. The semantic gap always exists in a sense that actions done by the processor do not have exactly the same semantics of the program code in high level languages. But only the semantic gaps related with program control flow may cause a critical vulnerability. As an example, consider functions like strcpy() or memcpy() have been used for copying data to a specified variable. Programmer using these functions may intend to copy some data to a specified variable but without a consideration about the actual actions of the processor. A compiled instruction sequence using the functions may not include a logic for checking the size of data to copy, causing a PC-bound data located next to the local variable for the data copy be overwritten. Also, note that the processor executes one instruction at a time, independent from other instructions. As a result, the processor is not aware of the context for each instruction and may execute the instructions out of context if the PC-bound data is compromised.

PC encoding encodes the destinations of indirect jump instructions including return addresses on stack, function addresses on GOT, function pointers, and exception handlers. It is, however, impossible to overwrite the hardcoded destination of a direct jump, and direct jumps are therefore excluded from the scope of PC encoding. **Figure 1(a)** shows the normal code parts of an object file. A memory-overwrite attack can occur during the execution time of {*Some code*}. If an attacker contaminates the PC bound data during the Some-code execution, the "*movpc*, *PC-bound data*" instruction will take the control flow to an illegal location. **Figure 1(b)**, however, shows a hardened case. Due to the key value, the PC-bound data is already in an encoded state during the Some-code execution, so an attacker cannot overwrite the intended-value to PC-bound data properly; the key value must be known by an attacker for a successful attack.

In general, the key must be stored in a recoverable point at the time of verification, and the memory is one of the typical places used for the key storage; however, if the key is stored in the memory, the key itself can be vulnerable to a memory-overwrite attack. This dilemma can be found in a number of protection schemes; for example, *StackShield* copies the return address of the function to a

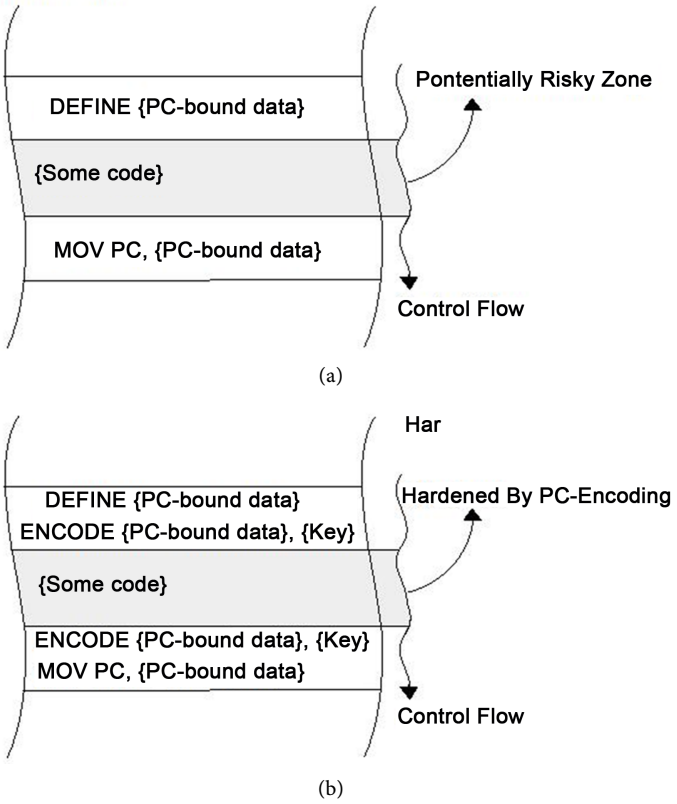

Figure 1. PC-encoding concepts.

specially reserved and managed area called *Global Ret Stackor Shadow RET Stack* [26]. *Stack Shield* then performs an integrity check through a comparison of the two values at the functions epilogue; however, the Shadow RET Stack is also stored in the dynamically-writable memory area, and it can still be a target of a memory-overwrite attack. Some schemes are therefore supported by low-level software like a kernel to ensure a read-only property of the key storage. These solutions maintain the key storage as a read only area during a code execution that is under protection, and they temporarily change the storage permission into a writeable form at the time that the PC-bound data is defined, e.g., the *mprotect()* function can change the memory permission dynamically, but this function can also be the victim of an RTL attack [27] [28] [29] [30] [31]. As a result, the *mprotect()* function can become a more serious vulnerability. Moreover, the frequent changes of the memory permission cause a critical performance penalty.

The key value used in this paper is the *self-address* of the PC-bound data. The self-address is the address of the memory location containing the PC-bound data. The self-address does not require any additional storage, and it is always available as a part of the value-address pair. Therefore, it is possible to extract the key easily at the moment of the decoding or the encoding of the target addresses. Also, it is not possible to compromise the self-address with a memory-overwrite attack because the address is not in the memory. The memory layout for different address spaces can be allocated with some level of randomness in

the ASLR environment [31] [32], and the allocation order of the function frames on the stack space can vary depending on the dynamic execution flow of the process.

One can apply PC-encoding to all indirect calls including calls and returns; however, it is difficult to locate the exact locations of all indirect calls with the exception of a few stylized cases such as GOT entries, function returns, and exception handlers. This paper focuses on a basic implementation under the ARM®-processor environment. In consideration of the fact that call/return pairs are the most frequent indirect branches, this paper focuses on the encoding and decoding of the return addresses for our first realization of the PC-encoding compiler for ARM® processors.

A variety of high-level languages such as C, C++, Fortran, and Ada can benefit from PC-encoding. PC-encoding adds an instrumentation of a few instructions into a binary executable. There are three ways one can add new instructions for the instrumentation into a binary executable. The first is a runtime modification (binary editing); however, with the binary editing at run time, it is possible to damage the functionality of the binary executable. Nevertheless, the technique may have the highest utility in the sense that it is independent of the languages that the code is written in. The second way is a binary patch. A binary patch is also of a high utility because it requires no source code. With this technique, it is possible to insert a protection patch by using only the executable binary without a dependence on the type of high-level language; however, a relocation of the addresses related to the inserted code patch can be a difficult problem to handle, and many clues for an understanding of the programmer's intent disappear from the binary executable. The third way is a compile-time modification. Generating additional code regarding a security mechanism at the compile step has a disadvantage, though, due to a dependency on the type of underlying high-level language. However, it can add the instrumented protection in a relatively reliable way due to its use of an internally-validated library. This leads to a guarantee of the functionality of the instrumented program.

In this paper, we have applied our PC-encoding technique at the compile time under the LLVM compiler infrastructure. The aim of the LLVM is to facilitate developing a compiler in a way that is independent from specific high-level languages and processor architecture. The LLVM *frontend* is separate from the code generation for an elimination of the dependency upon a specific high-level language. This frontend converts a high-level language code into an intermediate language code called the *LLVM IR*, e.g., the *Clang* is a typical frontend for the C language. By utilizing these features, codes that have been written in a variety of high-level-language types can be easily hardened with the PC-encoding technique.

3. Implementation

In the call/return convention under ARM®-processor architecture, ARM® processor stores the return address at *lr* (link register) when it uses a *bl* (branch

link) instruction to call a function. In the functions prologue, the *lr* and *fp* (frame pointer register) are pushed into the stack. At the functions epilogue, the *fp* and *pc* (program-counter register) are restored. Figure 2 shows an implementation overview of generating the hardened ARM-Linux-ELF binary via PC-encoding. This walk through is consisted of three steps. In step 1, the written code is converted from a high-level language to the LLVMIR code. In step 2, the hardened- assembly file is created with an LLVM IR file. The LLC is a tool that can convert LLVM IR files into a specified architecture-assembly file. It is possible to adjust options to make the assembly files of other architectures; for this paper, an assembly file for ARM® processors was constructed. In step 3, it is possible to determine the appropriateness of the patches or to collect the number of added instructions by analyzing the assembly file; then, step 3 converts the assembly file into a binary object file using the ARM®-processor assembler.

In step 2, a modified LLC named LLC.PCE is used for the implementation. The LLC.PCE applies PC-encoding to the LLVM IR file regardless of the high level language used. In the LLVM, the ARM Frame Lowering class is responsible for the code generation of the function frame in the ARM environment. The emit Prologue and the emit Epilogue functions of the ARM Frame Lowering Class were edited for the PC-encoding.

The general prologue and epilogue that the *gcc* compiler generates for the ARM environment are below:

```
Prologue:
pushfp,lr       //Save frame pointer and return
add sp,sp,$n    //Allocate space for local variables
Epilogue:
movsp,fp        //Move stack pointer to stack base
popfp,pc        //Restore frame pointer and return
```

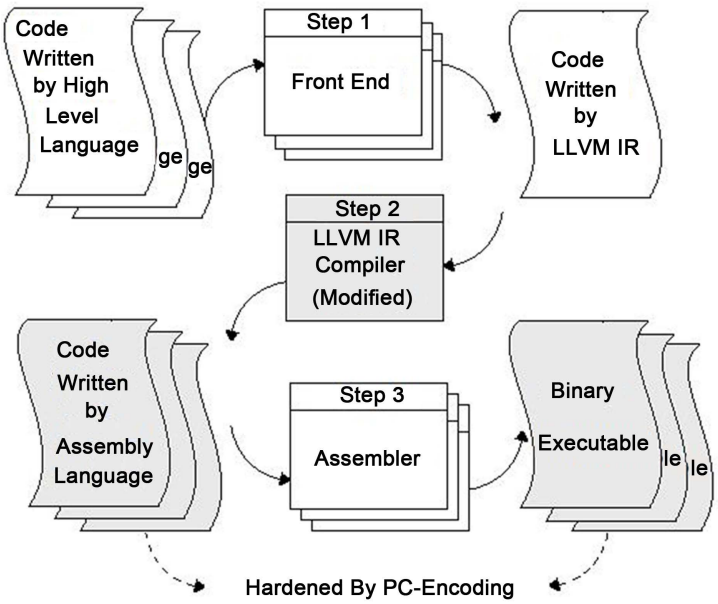

Figure 2. Implementation overview.

In the above code, the *lr* register is stored on the stack in the prologue, suggesting that the *lr* register can be overwritten by a memory-overwrite attack. PC-encoding inserts encoding and decoding instructions for the protection of the *lr*, as follows:

```
Prologue:
eorlr,lr,sp     //Encode return address
pushfp,lr       //Save frame pointer and return
add sp,sp,$n    //Allocate space for local variables
Epilogue:
movsp,fp        //Move stack pointer to stack base
ldrlr,[sp,#4]   //Load return address at lr
eorlr,lr,sp     //Decode return address
strlr,sp        //Save decoded return address
popfp,pc    //Restore frame pointer and return
```

In the above code, four instructions are added for PC-encoding. In this case, it is inevitable to have a memory access at the decode because of "*pop fp, pc*". The return address moves directly to the PC register *p c*, but the PC-encoding must decode the return address before this action. The decode process must therefore be accompanied by a memory access. At the function exit, the main intention of a programmer is just "*return to the caller*": The specific status of the registers and the memory are not always a part of the programmer's consideration. However, it is possible to manipulate the return address by using a frame-pointer overflow. If we follow *gcc* style of prologue and epilogue, compiler and its library do not correct this potential semantic-gap source. Furthermore, since the *fp* register is pushed to the stack, this can also be compromised by a memory overwrite attack. Both the *lr* and *fp* registers must therefore be encoded for the full protection of the return process in the *gcc* version.

To avoid the problem of additional memory access and the issues with the frame pointer, we have used the features in LLVM. Compilers do not always need to make the prologue-epilogue in gcc style as previously shown. LLVM can create a different prologue-epilogue form as follows:

```
Prologue:
pushlr          //save return address
add sp,sp,$n    //Allocate space for local variables
Epilogue:
addsp, sp, $n   //Mov stack pointer to base
poplr           //Restore return address
movpc,lr        //Return to caller
```

In the above code, an *fp* register is not utilized. The offsets are calculated based on the *sp* instead of the *fp* when the local variables or function parameters are accessed; furthermore, the return address is restored at *lr* before it moves into *pc*. As a result, it is possible to remove a memory access during the decode process. PC-encoding implemented without the additional memory access is as follows:

```
Prologue:
eorlr,lr,sp    //Encode return address
pushlr         //Save encoded return address
sub sp,sp,$n   //Allocate space
Epilogue:
add sp,sp,$n   //Move stack pointer to base
poplr          //Restore return encoded address
eorlr,lr,sp    //Decode return address
movpc,lr       //Return to caller
```

The two added *eor* instructions have no memory reference; as a result, we can protect the return with just two register-to-register instructions. Encoding the target addresses can avoid the unintended control transfer because it allows the control transfer to only a target address that is legitimately decoded, avoiding the potential semantic gap.

The PC-encoding implementation depends on a trust worthy low-layer software like the OS kernel. If the code is contaminated by an arbitrary memory-overwrite attack, it is not possible to protect the PC-bound data with PC-encoding; however, this does not mean that PC-encoding needs special functions like *mprotect()*.

4. Performance

By comparing the efficiency of the PC-encoding technique with a typical implementation of the well-known CFI [14], it is possible to illustrate the performance-overhead potential. One can patch the code with the ARM®-processor assembly to enforce the CFI as shown in Figure 3.

When the CFI is implemented on the ARM® processor, the caller needs one instruction and the recipient of the call needs three additional instructions;

```
caller:
blcallee          //Call the callee
b next            //Bypass the ID
[AABBCCDD] //block ID
next:
.
callee:
pushlr            //Save return address
sub sp,sp,$n      //Allocate space for local variables
.
add sp,sp,$n      //Move stack pointer to base
poplr             //Restore return address
ldr r3,[lr,#4]    //Obtain ID of return block
cmp r3,AABBCCDDh //Check validation
bne error label // Error handling
movpc,lr   //Return to caller(next)
```

Figure 3. Example instrumentation for CFI.

regarding the latter, one of the instructions is a memory reference instruction, while another is a conditional branch instruction. A memory reference instruction will take longer than the others, and a conditional branch instruction also may cause more cycles due to branch misprediction than those required by a normal command; consequently, a relatively large performance degradation may incur. **Table 1** shows the number of instructions between the CFI and the PC-encoding.

ROP [28] is one of the latest powerful code-reuse attacks that can bypass the existing defenses such as DEP, ASLR, and ASCII-Armor [30]. Attackers can gather fixed-position code parts that are called *gadgets*, and these fixed-position addresses can be selected without a containing NULL byte. By using these gadgets, the attacker can perform a desired operation without executing any code in the NX (non-executable) area. The overwrite attack and the advanced RTL attack that jumps to fixed PLT entries use some of these ROP gadgets, whereby an attacker can seize the control of a system more easily.

These advanced memory-overwrite attacks typically start from the contamination of one PC-bound data. Canary and ASCII-Armor can be effective against a linear memory-overwrite attack (smashing attack), but there are many ways to overwrite the PC-bound data without smashing. An arbitrary overwrite attack can bypass defenses such as Stack Guard [33] because the guard is separate from the return address. PC-encoding, however, protects the target directly; the target address and its protection are integrated together as a single data. This integrated way provides a useful form of protection regardless of whether the attacker uses a pointer for an indirect overwrite or just performs straight smashing. PC-encoding technique can therefore prevent the first intended jump using the target PC-bound data; meanwhile, the ROP-attack sequence that occurs after the overwriting of the first PC-bound data is blocked by the PC-encoding environment.

PC encoding also provides additional benefits against ROP. PC encoding can eliminate the ROP gadgets by inserting a decode instruction in front of the return instructions. Typical gadgets in a pattern of *pop-pop-ret* will be transformed into a *pop-pop-eor-ret* pattern, and an attacker should therefore be able to guess the key for the use of the gadgets. In the Intel x86 architecture, the insertion of instructions into the gadgets may cause a side effect of unexpected instructions that can be exploited for an ROP attack, because the unaligned instructions that are different from the programmer's intention can be fetched and executed with the Intel x86 architecture. But only the 4 byte aligned instructions are executable with the ARM® processor, meaning that the insertion of instructions

Table 1. The number of instructions in the CFI and PC encoding.

	added instruction	of memory access	conditional branch
CFI	4	1	1
PC-Encoding	2	0	0

for the elimination of a gadget can be a more reliable and clear solution under the ARM® architecture.

Among the potential gadgets-instruction sequences ending with the following indirect branch instructions: *b; bl; bx; blx; bxj* and *pop* {*pc, …}* or *mov pc, {reg/mem},* PC encoding removes the gadgets that use the "*pop* {*pc, …}" *or* "*mov pc, {reg/mem}*" instruction. However, binaries that are compiled by the LLVM rarely have the "*pop {pc,…}*" instruction because the LLVM compiler does not use the "*pop {pc,…}*" instruction as a return instruction. The "*b, bl, bx, blx, bxj*" gadgets are not removed by the current PC-encoding implementation reported in this paper, as the implementation in this paper covers only the return-address cases; therefore, attacks that use other PC-bound data can still succeed. Our future implementation will be able to handle all of the indirect branches in accordance with the use of our PC-encoding compiler under the Intel x86 architecture [21] [22].

The CFI is also capable of protecting the control flow from advanced attacks and can remove ROP gadgets; however, the CFI has a compatibility problem regarding the binary objects that are created separately due to a global ID-matching problem. For example, the CFI can cause a problem when a program tries to jump to the non-hardened block from a hardened block; in this case, the ID of the CFI verification codes regarding a proper jump are mismatched because the non-hardened block has no ID for the block entry. But there are no such issues with the PC-encoding environment because the definitions and verifications of the PC-bound data are usually in the same block; also, there is no need for the PC-encoding technique to maintain global IDs during the process.

Table 2 shows the performance results for SPEC2006 benchmark binaries that have been hardened by PC encoding. The simulations were performed with the default SE mode of the Gem5 simulator [34]. The SE mode of the Gem5 can simulate most Linux system calls, and it shows that the performance overhead is under 2%; also, the numbers of encode/decode instructions that were added by our PC-encoding compiler are shown in Table 2. The numbers of encoding and decoding instructions are not the same because a must-terminate epilogue terminates the process directly without the execution of a return instruction, and

Table 2. Performance results.

Name	Simulated instructions (million)		Overhead (instruction)	Overhead (tick)
	Normal	PC-encoding		
mcf	8953	9128	1.95%	1.93%
sjeng	33,823	34,170	1.02%	1.00%
Name	Input		Added instructions	
			Encode	Decode
bzip2	dryer.jpg		110	120
mcf	test/input.in		24	24
sjeng	test/test.txt		141	144

a function may have multiple epilogues upon the receipt of branch instructions. Unfortunately, the SE mode of Gem5 does not simulate all of the Linux system calls, and only a few of the binaries can be simulated properly on the Gem5. For example, the bzip2software that is used for compression needs to call a *utime()*, which is a system call that is not included in theGem5; therefore, the bzip2 execution cannot be finished properly on the Gem5.

5. Conclusions

This paper presents a practical guide for the implementation of a PC-encoding compiler under the ARM®-processor architecture. The ARM®-processor architecture has become the most popular one for embedded and application processors, and it is widely adopted in embedded devices including smartphones. We used our experience of building the PC-encoding compiler for the Intel x86 architecture [21] [22] as the basis for our implementation of the PC-encoding compiler for the ARM® processor. Our implementation protects the return addresses without inserting a protection code for accessing the memory; therefore, the protection incurs a minimal performance overhead.

Regarding the implementation presented in this paper, the key utilized for the encoding process is the self-address, *i.e.*, the location of the instruction defining PC-bound data. This key value of the self-address allows a low overhead implementation; however, if the degree of ASLR randomization is weak or no re-randomization occurs after a crash, it is possible for attackers to determine the key value using replay attacks and a source-code analysis. For more-secure protection, it may be necessary to utilize cryptographic keys that are more difficult to guess [22].

The PC-encoding implementation presented in this paper does not provide protection for every type of attacks because it has focused only on the return addresses among the many types of PC-bound data. While compromising the return address is the most prevailing software attacks for compromising the control flow integrity, many software attacks that do not rely on the return address exist. To protect other PC-bound data, a few techniques have been proposed. For example, it is possible to insert decoding instructions into the PLT region to prevent GOT-overwrite attacks, whereby an encode instruction can be added to the *dl_resolve()* function [21]. Also, one may utilize the relocation table to find every indirect jump [18]. We are currently in the process of incorporating these techniques into our ARM®-processor PC encoding compiler.

Acknowledgements

This work was supported in part by the National Research Foundation of Korea (NRF 2015R1A2A2A01).

References

[1] Ahn, Y.-J., Lee, Y., Choi, J. and Lee, G. (2014) Countering Code Injection Attack at TLB Miss *IEEE Computer*, **47**, 66-72. https://doi.org/10.1109/MC.2013.228

[2] Microsoft. A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2,Windows XP Tablet PC Edition 2005, and Windows Server 2003. https://support.microsoft.com/en-us/kb/875352

[3] Park, Y.J., Zhang, Z. and Lee, G. (2006) Micro-Architectural Protection against Buffer Overflow Attack. *IEEE Micro*, **26**, 62-71. https://doi.org/10.1109/MM.2006.76

[4] Red Hat. New Security Enhancements in Red Hat Enterprise Linux v.3, update3. http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf

[5] Roemer, R., Buchanan, E., Shacham, H. and Savage, S. (2012) Return-Oriented Programing: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, **15**, 2:1-2:34.

[6] Bletsch, T., Jiang, X., Freeh, V.W. and Liang, Z. (2011) Jump-Oriented Programming: A New Class of Codereuse Attack. *Proceedings of the* 6*th ASIACCS*, pp. 30-40, March. https://doi.org/10.1145/1966913.1966919

[7] Buchanan, E., Roemer, R., Shacham, H. and Savage, S. (2008) When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. *ACM Conference on Computer and Communications Security* (*CCS*), Alexander, 27-31 October 2008, 27-38. https://doi.org/10.1145/1455770.1455776

[8] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H. and Winandy, M. (2010) Return-Oriented Programming without Returns. *ACM Conference on Computer and Communications Security* (*CCS*), Chicago, 4-8 October 2010, 559-572.

[9] Dai Zovi, D. (2010) Practical Return-Oriented Programming. SOURCE Boston.

[10] Davi, L., Dmitrienko, A., Sadeghi, A.R. and Winandy, M. (2010) Return-Oriented Programming without Returns on ARM. Technical Report HGI-TR-2010-002, Ruhr-University Bochum, July 2010. http://www.trust.rub.de/home/publications/DaDmSaWi2010

[11] Ahn, D. and Lee, G. (2015) A Memory Access Validation Scheme against Payload Injection Attacks. *IEEE Transactions on Dependable and Secure Computing*, **12**, 387-399. https://doi.org/10.1109/TDSC.2014.2355844

[12] Ruan, Y., Kalyanasundaram, S. and Zou, X. (2016) Survey of Return-Oriented Programming Defense Mechanisms. *Security and Communication Networks*, **9**, 1247-1265. https://doi.org/10.1002/sec.1406

[13] Forrest, S., Hofmeyr, S. and Somayaji, A. (2008) The Evolution of System-Call Monitoring. *Annual Computer Security Applications Conference*, Anaheim, 8-12 December 2008, 418-430. https://doi.org/10.1109/acsac.2008.54

[14] Abadi, M., Budiu, M., Erlingsson, U. and Ligatti, J. (2009) Control-Flow Integrity: Principles, Implementations, and Applications. *ACM Transactions on Information and System Security*, **15**, 1-40. https://doi.org/10.1145/2240276.2240279

[15] Pewny, J. and Holz, T. (2013) Control-Flow Restrictor: Compiler-Based CFI for iOS. *Proceedings of the* 29*th Annual Computer Security Applications Conference*, New Orleans, 9-13 December 2013, 309-318. https://doi.org/10.1145/2523649.2523674

[16] Zeng, B., Tan, G., and Morrisett, G. (2011) Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. *Proceedings of the* 18*th ACM conference on Computer and Communications Security*, Chicago, 17-21 October 2011, 29-40. https://doi.org/10.1145/2046707.2046713

[17] Zhang, M. and Sekar, R. (2013) Control Flow Integrity for COTS Binaries. 22*nd USENIX Security Symposium*, Washington DC, 14-16 August 2013, 337-352.

[18] Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D. and

Zou, W. (2013) Practical Control Flow Integrity and Randomization for Binary Executables. *Proceedings of the IEEE Security and Privacy Symposium*, San Francisco, 19-22 May 2013, 559-573.

[19] Goktas, E., Athanasopoulos, E., Bos, H. and Portokalidis, G. (2014) Out of Control: Overcoming Control-Flow Integrity. *Proceedings of the* 2014 *IEEE Symposium on Security and Privacy*, San Jose, 18-21 May 2014, 575-589. https://doi.org/10.1109/SP.2014.43

[20] Lee, G. and Tyagi, A. (2000) Encoded Program Counter: Self-Protection from Buffer Overflow Attacks. *International Conference on Internet Computing*, Las Vegas, 26-29 June 2000, 387-394.

[21] Lee, G. and Pyo, C. (2013) Method and Apparatus for Securing Indirect Function Calls by Using Program Counter Encoding. US Patent No. US8583939 B2.

[22] Pyo, C. and Lee, G. (2002) Encoding Function Pointers and Memory Arrangement Checking against Buffer Overflow Attack. 4*th International Conference on Information and Communications Security*, Lecture Notes in Computer Science 2513, Singapore, 9-12 December 2002, 25-36. https://doi.org/10.1007/3-540-36159-6_3

[23] Barrantes, E.G., Ackley, D.H., Forrest, S. and Stefanovic, D. (2005) Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security*, **8**, 3-40. https://doi.org/10.1145/1053283.1053286

[24] Gupta, A., Habibi, J., Kirkpatrick, M.S. and Bertino, E. (2015) Marlin: Mitigating Code Reuse Attacks Using Code Randomization. *IEEE Transactions on Dependable and Secure Computing*, **12**, 326-337. https://doi.org/10.1109/TDSC.2014.2345384

[25] Computer Science Department at the University of Illinois at Urbana-Champaign. The LLVM Compiler Infrastructure. http://llvm.org

[26] Sinnadurai, S., Zhao, Q. and Wong, W. (2008) Transparent Runtime Shadow Stack: Protection against Malicious Return Address Modifications.

[27] Nergal (2001) The Advanced Return-Into-Lib(C) Exploits: PaX Case Study. *Phrack Magazine*, **58**, 54.

[28] Shacham, H. (2007) The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the x86). 14*th ACM Conference on Computer and Communications Security*, Alexandria, 29 October-2 November 2007, 552-561. https://doi.org/10.1145/1315245.1315313

[29] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V. and Ning, P. (2011) On the Expressiveness of Return-Intolibc Attacks. 14*th International Conference on Recent Advances in Intrusion Detection*, Menlo Park, 20-21 September 2011, 121-141. https://doi.org/10.1007/978-3-642-23644-0_7

[30] Black Hat USA Whitepaper (2010) Payload Already Inside: Data Reuse for ROP Exploits.

[31] Pax Team. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt

[32] Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N. and Boneh, D. (2004) On the Effectiveness of Addressspace Randomization. 11*th ACM Conference on Computer and Communications Security*, Washington DC, 25-29 October 2004, 298-307.

[33] Cowan, C., Pu, C., Maier, D., Hintongif, 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. 7*th USENIX Security Symposium*, San Antonio, 26-29 January 1998, 63-78.

[34] Gem5 Simulator. http://www.gem5.org