Control Flow Hardening with Program Counter Encoding for ARM® Processor Architecture

Seho Park, Jongmin Lee, Yongsuk Lee, and Gyungho Lee

Abstract—Software operations 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 results in the semantic gap, are a major attack type in modern software attacks. Many recent software protection schemes focus on protecting program control flows. But even without challenging the protection scope of the schemes, most of them suffer not only from the deployment difficulties but also from critical performance issues. 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 and to implement and incurs 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 binary in ELF (Executable and Linkable Format). Our LLVM (low level virtual machine) based PC-Encoding compiler provides control flow protection with little overhead for programs running under ARM® processor architecture.

Keywords—Key-Words: Compiler, Control Flow Integrity, PC-Encoding, Software Security, ARM®.

I. INTRODUCTION

Software security has become increasingly important concern with the prevalent use of the Internet. With a looming popularity of the Internet of Things (IoT), the concern becomes more prevalent in every aspect of modern life. Various security techniques have been researched and developed at the software level, and some of these have actually been adopted in practice. However, attackers are able to continue to find vulnerabilities, and the attacks succeed more or less in the same way as used to be but with more alarming rates. The number of vulnerabilities utilized for attacks is increasing, and the risks inherent in the security vulnerabilities have become even more serious.

A major portion of the vulnerabilities allows memory overwrite, which in turn causes program control transfer in a way the programmer did not intend. This semantic gap, i.e., potential difference between intended operations described in software and actual operations done by processor can exist as long as there is a vulnerability allowing a memory overwrite.

There have been various techniques for preventing arbitrary memory overwrite along with schemes for preventing the activation of injected attack code, e.g., many variations of DEP (Data Execution Prevention) [13], [18]. However, software attacks utilizing memory overwrite keep appearing. A recent sophisticated attack method is ROP (Return Oriented Programming), a code reuse attack that does not need to inject attack code but needs a memory overwrite to start an attack with the code already existing in the memory [2], [3], [4], [7], [8]. Note that a memory overwrite is essential not only in traditional code injection attacks but also in recent code reuse attacks to divert program control flows from the intended operations described in software.

In order to preserve the control flow from software attacks, we need to enforce the integrity of control flows by examining the destinations of all control flow transfer instructions to see whether they are legitimate or not. Conceptually one can generate a control flow graph (CFG) for a program under protection and check at run time to see if program execution actually follows the CFG. Even without contemplating the accuracy of the CFG, a couple of issues arise from this conceptual control flow validation scenario in terms of the granularity of the CFG along with representing and storing the CFG. System call level CFG has been utilized in early days for the sake of reducing the overhead of representing and storing the CFG and also of accessing the CFG at run time [9]. However, such a coarse grain CFG often fails to catch a compromised control flow because there may exist many hundred or thousand instructions exist between the calls.

The CFI (Control flow integrity) [1], [16], [23] is one of the early proposals to check and validate each control flow transfer at fine grain machine instruction level; force the destinations of all indirect branch instructions to be checked. The full CFG with a pair of IDs for all indirect branch instructions, branch instruction address ID and its target address ID, is generated by binary patch, and each control flow transfer is checked at run time per the CFG. However, even if the CFG is limited to statically linked procedures, it is very difficult to draw a full complete CFG in practice and will cause a serious performance overhead for representing and accessing the CFG. Basic
Section III. Section IV presents the performance test results of applying the PC-Encoding for degradation. Environment of ARM® processors with low performance from Gem5 simulator. This paper closes in Section V with a discussion on the limitations of our current implementation of PC-Encoding compiler provides control flow protection with little overhead for programs running under ARM® processor architecture [5]. Our implementation suggests that the PC-Encoding fits well for a relatively simple architecture environment of ARM® processors with low performance degradation.

This paper describes our efforts to implement the PC-Encoding to harden portable binary in ELF (Executable and Linkable Format). Our LLVM (low level virtual machine) based PC-Encoding compiler provides control flow protection with little overhead for programs running under ARM® processor architecture [5]. Our implementation suggests that the PC-Encoding fits well for a relatively simple architecture environment of ARM® processors with low performance degradation.

Section II discusses the basics of PC-Encoding. The process of applying the PC-Encoding for ARM-Linux-elf binary is in Section III. Section IV presents the performance test results from Gem5 simulator. This paper closes in Section V with a discussion on the limitations of our current implementation of PC-Encoding for ARM® processors.

II. BACKGROUNDS

Program control-flow is dictated by program data loaded to the program counter at runtime, which we call the PC-bound data. The basic idea of the PC-Encoding is to check the integrity of the PC-bound data. The PC-Encoding ensures the integrity of program control flow by protecting the PC-bound data. The PC-Encoding encodes the PC-bound data at their definition and decode at their use with a secret key. The PC-Encoding makes a good solution to harden program control flow for embedded platforms because it has little performance penalty. Data encoding/decoding computation can be made to be simple. Simple encoding/decoding operation that can be done in one or two cycles may be employed, e.g. exclusive-or. Also, there is no compatibility issue. It doesn’t need to change memory layout. Moreover it can cooperate well with not hardened binary.

The PC-Encoding encodes destination of indirect jump instructions such as return addresses on stack, function addresses on GOT, function pointers, or exception handlers. But it is impossible to overwrite hard coded destination in direct jump. Therefore, direct jumps are excluded from scope of the PC-Encoding. Fig. 1(a) shows normal code parts in object file. Memory overwrite attack can occur during execution time of the Some code. If an attacker contaminates the PC-bound data during the Some code execution, the “mov pc, {PC-bound data}” instruction will take the control flow to illegal location. However, Fig. 1(b) shows hardened case. The PC-bound data is already in encoded state by the Key during the Some code execution, so attacker can’t overwrite PC-bound data with intended value. Consequently, the Key is a value that the attacker must know for a successful attack.

In general, the key must be stored in a recoverable point at the time of verification. Typical places that can be mentioned as a point of storage is memory. However, if the key is stored in memory, the key itself can be vulnerable from the memory overwrite attack. This dilemma can be found in some protection schemes. e.g., StackShield copies the return address of the function to special purpose area, called Global Ret Stack or Shadow RET Stack [21]. And then it performs integrity check by comparing the two values at the epilogue of functions. However, the Shadow Ret Stack is also dynamically writable memory area. And it can still be a target of the memory overwrite attack. So some schemes are supported by low level software like kernel to ensure read-only property of the key storage. These solutions maintain the key storage as read-only area during a code execution under protection, and temporarily change storage permission to writeable at the time of defining PC-bound data, e.g., mprotect() function can change memory
permission dynamically, but this function also can be the victim of RTL attack[14],[19],[22],[26]. As a result, mprotect() function can become a more serious vulnerability. Moreover, frequent dynamic change of memory permission will causes critical performance penalty.

The value used as the key 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 points a permanent space that is programmable but requires no additional storage to store it. And it is always maintained as a part of the value-address pair. Therefore it is possible to extract the key easily at the moment of decoding or encoding the target addresses. Also it is not possible to compromise the self-address with the memory overwrite attack, because address is not in the memory. Memory layout for different address spaces can be allocated with some level of randomness in ASLR environment [15],[20]. And the allocation order of function frames on the stack space can vary depending on the dynamic execution flow of the process.

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

Various high level languages such as C, C++, Fortran and Ada can benefit from PC-Encoding. PC-Encoding adds a few instructions into a binary executable in order to harden the binary executable. There are three ways one can add new instructions into a binary executable. The first is a runtime modification (binary editing). With the binary editing at runtime it is possible to damage the functionality of the binary executable. Nevertheless, it may have the highest utility in the sense that it is independent from the languages the code is written. The second is a binary patch. Binary patch also is of a high utility, because it requires no source code. It is possible to insert a protection patch by using only the executable binary without depending on the type of high-level languages. However, relocation of the addresses due to the inserted code patch can be a difficult problem, because many clues for understanding the programmer's intent would have disappeared in the binary executable. Third is the modification at the compile time. Adding additional code of security mechanism at the compile step has a disadvantage, i.e., dependency on the type of high-level languages. In the worst case, it needs to modify compilers manually as many as the number of high-level languages used to extend the coverage of adoption. However, it can add a protection in a relatively reliable way by using an internally validated compiler library. This property leads to a guarantee of the functionality of the modified program.

In this paper, we have applied our PC-Encoding at the compile time. A compiler infrastructure, LLVM, was used to complement for the limitations mentioned above. The LLVM aims to operate independently from high level language and the architecture. The LLVM is with the frontend separate from code generation to eliminate the dependency with a specific high level language. The 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 written in various types of high level language can be easily hardened with the PC-Encoding.

In order to understand this paper, it is necessary to know the Arm® processor Architecture. The Arm® processor stores the return address at lr(Linked Register) when it calls a function by bl(Branch Link) instruction. In the prologue of functions, lr and fp (frame pointer Register) are pushed into the stack. At the epilogue of functions, fp and pc(program counter register) are restored.

III. IMPLEMENTATIONS

Fig.2 is implementation overview about generating hardened ARM-Linux-elf binary. This walkthrough consists of three steps. In step 1, it converts the code written in high level language to LLVM IR code. In step2, it creates the hardened Assembly file with LLVM IR file. The LLC is a tool that can convert LLVM IR files into a specified architecture assembly file. Also it is possible to adjust options to make other architecture’s assembly file. In this paper it make assembly file for ARM® processors. In Step 3, it is possible to determine whether the appropriate patches or collect the number of added instructions by analyzing the assembly file. And then, Step 3 convert assembly file to binary object file through ARM® processor Assembler.
In step 2, the implementation uses a modified LLC, named LLC.PCE. The LLC.PCE will apply the PC-Encoding to LLVM IR file regardless of the high level language. Among the LLVM source code, the ARMFrameLowering class is responsible for code generation of the function frame in the ARM® environment. The emitPrologue and the emitEpilogue functions in the ARMFrameLowering Class were edited for the PC-Encoding.

Prologue and epilogue that the gcc compiler generates in general for the ARM® environment is in the below.

Prologue:

```
push {fp,lr}  //Save frame pointer and return
add sp,sp,$n //Allocate space for local variables
```

Epilogue:

```
mov sp,fp   //Move stack pointer to stack base
pop {fp,pc} //Restore frame pointer and return
```

In above code, the lr register is stored at stack in prologue. It is mean that lr register can be overwritten by memory overwrite attack. The PC-Encoding can insert encode and decode instructions for protect lr.

Prologue:

```
eor lr,lr,sp    //Encode return address
push lr     //Save encoded return address
sub sp,sp,$n //Allocate space
```

Epilogue:

```
add sp,sp,$n //Move stack pointer to base
pop lr     //Restore return address
mov pc,lr   //Return to caller
```

In this code, there is no fp register. Offsets are calculated based on the sp instead of using fp when accessing local variables or function parameters. And the return address is restored at lr before it moves into pc. By using this moment, it is possible to remove a memory access in decode process. The PC-Encoding example implemented in this consideration is as follows.

Prologue:

```
eor lr,lr,sp //Encode return address
push lr     //Save encoded return address
sub sp,sp,$n //Allocate space
```

Epilogue:

```
add sp,sp,$n //Move stack pointer to base
pop lr     //Restore return encoded address
eor lr,lr,sp //Decode return address
mov pc,lr   //Return to caller
```

Two added eor instructions have no memory reference. As a result, we are able to protect return process of a function with just two of register-to-register instructions. Encoding the target addresses can avoid unintended control transfer causing the semantic gap because it allows the control transfer only to a target address legitimately decoded.

In fact, at the function exit, the main intention of a programmer is just a "return to the caller". The status of the registers and the memory are not always in programmer’s consideration. However, it is possible to manipulate the return address using a frame pointer overflow. Compilers and its library do not perform action to correct this potential semantic gap source. Since fp register is also pushed to stack, it can be exploited. So both lr and fp registers must be encoded for the full protection of return process in the gcc version.

The PC-Encoding implementation depends on trustworthy low layer software like the OS kernel. If the code were contaminated by an arbitrary memory overwrite attack, it is not possible to protect the PC-bound data with PC-Encoding. But it does not mean that the PC-Encoding needs special functions like the mprotect().

IV. PERFORMANCE

Comparing the efficiency of PC-Encoding with the well-known CFI [1] is illustrating to see the performance overhead potential. One can patch the code with ARM® processor assembly to have the CFI enforced as in the below.

Prologue:

```
add sp, sp, $n //Mov stack pointer to base
pop lr     //Restore return address
mov pc,lr   //Return to caller
```

In this code, there is no fp register. Offsets are calculated based on the sp instead of using fp when accessing local variables or function parameters. And the return address is restored at lr before it moves into pc. By using this moment, it is possible to remove a memory access in decode process. The PC-Encoding example implemented in this consideration is as follows.

Prologue:

```
eor lr,lr,sp // Encode return address
push lr     // Save encoded return address
sub sp,sp,$n //Allocate space
```

Epilogue:

```
add sp,sp,$n //Move stack pointer to base
pop lr     //Restore return encoded address
eor lr,lr,sp //Decode return address
mov pc,lr   //Return to caller
```

Two added eor instructions have no memory reference. As a result, we are able to protect return process of a function with just two of register-to-register instructions. Encoding the target addresses can avoid unintended control transfer causing the semantic gap because it allows the control transfer only to a target address legitimately decoded.

In fact, at the function exit, the main intention of a programmer is just a "return to the caller". The status of the registers and the memory are not always in programmer’s consideration. However, it is possible to manipulate the return address using a frame pointer overflow. Compilers and its library do not perform action to correct this potential semantic gap source. Since fp register is also pushed to stack, it can be exploited. So both lr and fp registers must be encoded for the full protection of return process in the gcc version.

The PC-Encoding implementation depends on trustworthy low layer software like the OS kernel. If the code were contaminated by an arbitrary memory overwrite attack, it is not possible to protect the PC-bound data with PC-Encoding. But it does not mean that the PC-Encoding needs special functions like the mprotect().

IV. PERFORMANCE

Comparing the efficiency of PC-Encoding with the well-known CFI [1] is illustrating to see the performance overhead potential. One can patch the code with ARM® processor assembly to have the CFI enforced as in the below.

Prologue:

```
push lr     //save return address
add sp,sp,$n //Allocate space for local variables
```

Epilogue:

```
bl callee  //Call the callee
b next     //Bypass the ID
```
The PC-Encoding provides additional benefits against ROP. The PC-Encoding can eliminate the ROP gadgets by inserting 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 thus attackers should be able to guess the key in order to use the gadgets. In Intel x86 architecture, inserting instruction into the gadgets may cause a side effect of unexpected instructions that can be exploited for ROP attack, because unaligned instructions different from the programmer’s intention can be fetched and executed with Intel x86 architecture. But only the 4byte aligned instructions are allowed to execute with ARM® processor. It means that inserting instructions to eliminate the gadget can be more reliable and clear solution with ARM® architecture.

This paper assumes the following instruction sequences as potential gadgets.

\[
\{{\text{Several instructions}}}; \ \text{mov pc, } \{{\text{reg/mem}}\};
\]

\[
\{{\text{Several instructions}}}; \ \text{bx}, \ \text{bl}, \ \text{bxj};
\]

\[
\{{\text{Several instructions}}}; \ \text{pop} \ {\{\text{pc, ...}}\};
\]

Binaries compiled by the LLVM rarely have the "pop \{pc, ...}" instruction, because the LLVM compiler does not use "pop \{pc, ...}" instruction as a return instruction. Therefore, the PC-Encoding removes most of the gadgets that use the "mov pc, \{reg/mem\}" instruction. However, "b, bl, bx, bxj" gadgets are not eliminated by the current PC-Encoding implementation reported in this paper. The implementation in this paper covers only return address case. So attacks using other PC-bound data can still valid. Also this narrow protection scope causes un-elimination of “b, bl, bx, bxj” gadgets. But it is an implementation problem rather than a fundamental disadvantage inherent in the concept of PC-Encoding. Our future implementation will be able to handle all the indirect branches as done in our PC-Encoding compiler for Intel x86

---

Table 1 Number of instructions in CFI and PC-Encoding

<table>
<thead>
<tr>
<th></th>
<th>added instruction</th>
<th>of memory access</th>
<th>conditional branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFI</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PC-Encoding</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

Table 2 Performance result

<table>
<thead>
<tr>
<th>name</th>
<th>Simulated instructions (million)</th>
<th>Overhead (instruction)</th>
<th>Overhead (tick)</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcf</td>
<td>8953</td>
<td>1.95%</td>
<td>1.93%</td>
</tr>
<tr>
<td>sjeng</td>
<td>33823</td>
<td>1.02%</td>
<td>1.00%</td>
</tr>
</tbody>
</table>

---

Table 2 Performance result

<table>
<thead>
<tr>
<th>name</th>
<th>Input</th>
<th>Added instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Encode</td>
<td>Decode</td>
</tr>
<tr>
<td>bzip2</td>
<td>dryer.jpg</td>
<td>110</td>
</tr>
<tr>
<td>mcf</td>
<td>test/input.in</td>
<td>24</td>
</tr>
<tr>
<td>sjeng</td>
<td>test/test.txt</td>
<td>141</td>
</tr>
</tbody>
</table>

---

When implementing the CFI on ARM® processor, the caller needs one and the callee needs three additional instructions. One of them is a memory reference instruction, and the other one is conditional branch instruction. Memory reference instruction will take longer than the others, and the conditional branch instruction also requires more cycles than a normal command requires. Consequently, it will causes the relatively large performance degrade. Table.1 shows the number of instructions between the CFI and the PC-Encoding.

ROP (Return Oriented Programming) [19] is a latest powerful code reuse attack that can bypass the defenses such as DEP, ASLR and ASCII-Armor [26]. Attackers can gather fixed position code parts, called gadget. These fixed position address can be selected with not containing NULL byte. The Attacker can performs a desired operation without executing the code on the NX(Non-Executable) area using these gadgets. The overwrite attack and advanced RTL attack that jumps to fixed PLT entries use some the ROP gadgets, and it allows the attacker seize control of the system more easily.

First of all, these advanced memory overwrite attack should start from the contamination of one PC-bound data. Canary and ASCII-Armor can take effect against the linear memory overwrite attack (smashing attack). But there are many ways to overwrite the PC-bound data. Arbitrary overwrite attack can bypass the defense such as StackGuard [6] because the guard is separate from the return address. The PC-encoding protects the target directly, i.e., the target address and its protection is integrated together as a single data: it is still a useful protection whether the attacker uses a point for indirect overwrite or does straight smashing. Thus, the PC-Encoding can prevent the first intended jump using the target PC-bound data. ROP attack sequence occurring after overwriting the first PC-bound data is blocked by the PC-Encoding environment.
processors. For example, it is possible to inserting decoding jumps as in CCFIR [25]. We are currently in the process of with the encode instruction added to instructions into PLT region to prevent GOT overwrite attacks randomization is weak, it is possible for attackers to easily figure out the key via replay attacks and source code analysis. If the degree of ASLR's implementation. However, if the key is guessed in a low overhead. Therefore, the protection is with minimal performance overhead.

In our implementation presented in this paper, the key for the PC-Encoding is the self-address, i.e., the location of the instruction defining PC-bound data. It allows a low overhead implementation. However, if the degree of ASLR’s randomization is weak, it is possible for attackers to easily figure out the key via replay attacks and source code analysis. For more tight protection we may want to resort on utilizing harder to guess cryptographic keys [17].

The PC-encoding implementation presented in this paper is not able to prevent every type of attacks, because it focuses only on the return addresses among many types of the PC-bound data. Many software attacks not resorting on the return address exist, and a few techniques have been proposed to solve this problem. For example, it is possible to insert decoding instructions into PLT region to prevent GOT overwrite attacks with the encode instruction added to dl_resolve() function [11]. Also, one may utilize the relocation table to find every indirect jumps as in CCFIR [25]. We are currently in the process of incorporating these into our PC-Encoding compiler for ARM® processors.

V. CONCLUSION

This paper has shown a guide to make practical PC-Encoding compiler implementation under Arm® processor architecture. Arm® processors have become the most popular embedded and application processor architecture, widely adopted in embedded devices including the smart-phones. Based on our experience of building PC-Encoding compiler for Intel x86 architecture [11], [17], we are implementing PC-Encoding compiler for ARM® processor. By utilizing the features of Arm® processors, our implementation protects the return addresses without inserting a protection code accessing the memory. Therefore, the protection is with minimal performance overhead.

REFERENCES


