석 사 학 위 논 문
Master's Thesis

RecurFI: 윈도우 바이너리에 대한 효율적 제어흐름 무결성 검사 시스템

RecurFI: Practical Coarse-Grained CFI on Windows Binary Code

2017

홍 진 아 (洪 鎭 娥 Hong, Jina)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

RecurFI: 윈도우 바이너리에 대한 효율적 제어흐름 무결성 검사 시스템

2017

홍 진 아

한 국 과 학 기 술 원

전산학부 (정보보호대학원)

RecurFI: 윈도우 바이너리에 대한 효율적 제어흐름 무결성 검사 시스템

홍 진 아

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2016년 12월 16일

심사위원장 김 광 조 (인)

심 사 위 원 차 상 길 (인)

심 사 위 원 김 용 대 (인)

RecurFI: Practical Coarse-Grained CFI on Windows Binary Code

Jina Hong

Major Advisor: Kwangjo Kim
Co-Advisor: Sang Kil Cha

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science (Information Security)

Daejeon, Korea
December 16, 2016

Approved by

Kwangjo Kim
Professor of Graduate School of Information Security

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MIS
20153708

홍진아. RecurFI: 윈도우 바이너리에 대한 효율적 제어흐름 무결성 검사 시스템. 전산학부 (정보보호대학원) . 2017년. 25+ii 쪽. 지도교수: 김광조, 차상길. (영문 논문)
Jina Hong. RecurFI: Practical Coarse-Grained CFI on Windows Binary Code. School of Computing (Graduate School of Information Security) . 2017. 25+ii pages. Advisor: Kwangjo Kim, Sang Kil Cha. (Text in English)

초 록

갈수록 진화하는 해킹 공격에 대응하기 위해 등장한 제어흐름 무결성 기술은 강력한 소프트웨어 방어의 패러다임을 제시하였지만, 막중한 오버헤드로 인해 실제로 적용하는 데에는 많은 한계가 있었다. 최근 들어 이러한 제어흐름 무결성 기술의 안전성과 성능을 절충하는 다양한 실용적 기술이 등장하였는데, 그 한 예로 MS가 2012년도에 공개한 EMET(Enhanced Mitigation Experience Toolkit)가 있다. 하지만 이를 위협하는 새로운 공격기법들 또한 등장하고 있으며, 이러한 공격을 효율적으로 방어하는 바이너리 기반의 시스템은 아직까지 실용화되지 못하였다. 본 논문에서는 제어흐름 무결성 기술을 우회하는 최신의 공격 기법을 효과적으로 방어할 수 있는 바이너리 기반의 새로운 공격 탐지 시스템인 RecurFI를 제시한다. RecurFI는 어떠한 어플리케이션에도 적용이 가능하며, SPEC 벤치마크를 통해 그 효율성(평균 2%미만의 오버헤드)을 입증하였다.

핵 심 낱 말 보안, 실시간 모니터링, 취약점

Abstract

Although Control-Flow Integrity (CFI) presents a powerful software defense paradigm, it requires a significant runtime overhead in practice. The traditional coarse-grained CFI solutions such as Microsoft's Enhanced Mitigation Experience Toolkit (EMET) address this issue by compromising the security enforced by the original CFI. Unfortunately, though, due to its relaxed policy, new types of exploits have been emerging lately. In this thesis, we present RecurFI, a novel coarse-grained CFI solution that proposes a new sweet spot between security and performance. We define the CFI policies by capturing the general pattern of normal execution and examine recursively whether execution of the protected process satisfy the CFI policies or not. We show that RecurFI can detect all the exploits in our experiment with less than 2% runtime overhead on average.

Keywords Security, Runtime Monitoring, Vulnerabilities

Contents

List of Tables

List of Figures

Chapter 1. Introduction

Control-Flow Integrity (CFI) [1] presents a robust software defense paradigm that primarily defends against control-flow hijack attacks. The CFI incapacitates various conventional hacking attacks based on code injection, or code-reuse, because the CFI permits the execution of a program only through a control-flow that has been allowed in advance. A typical example of such an attack is a Return-Oriented Programming (ROP) attack, [2] which executes a gadget (a code block having a `RET` instruction at the end) through a control flow that may not be generated by a normal program. CFI can effectively prevent control-flow hijacking attacks, but there are two fundamental limitations in applying CFI to real programs. First, it generates a huge overhead, because instrumentation is required for all indirect branches of a target program. Second, it is difficult to apply CFI to binary codes directly. Control-flow Graph (CFG) is necessary to apply the CFI to a program, however, perfectly restoring the CFG from binary codes is impossible.

Recently, coarse-grained CFIs has been proposed to practically solve these inherent problems. To reduce the limit of CFI, coarse-grained CFIs do not perform instrumentation for all indirect branch statements and do not utilize the CFG. To do this, the relaxed CFI has a low overhead that can be applied to commercial products as well. For example, Compliers such as the GNU C Compiler (GCC) and Low-Level Virtual Machine (LLVM) have employed a source code-based coarse-grained CFI enforcement [3]. Microsoft has also released a practical CFI named ROPGuard [4] through a tool called the Enhanced Mitigation Experience Toolkit (EMET) [5]. When a `RET` instruction is executed for a specific Windows Application Programming Interface (API) call, the EMET verifies whether the return address corresponds to the address following the `CALL` instruction. This function of the EMET is similar to a shadow stack [6] , but the defense mechanism of the EMET is a more relaxed type. It has been developed as a technique to efficiently perform coarse-grained CFI using hardware [7].

The emergence of coarse-grained CFI has compromised safety and performance, resulting in a new type of security threat. One example of security threat is an attack that bypasses Microsoft's EMET. An attacker uses a gadget that has no other branch statement between the `CALL` instruction and the `RET` instruction in order to circumvent EMET.

The CFI studies we have seen clearly demonstrate the tradeoff between performance and security. The first proposed CFI [1] can prevent many types of control-flow hijacking attacks, but its performance is poor. On the other hands, the coarse-grained CFI [4, 7, 3] is good enough to be applied to commercial products, but it can be bypassed. It is still an open problem to find a way to prevent attacks completely and practically.

In this thesis, we present a new conceptual system, RecurFI, which effectively defends an evolved ROP attack that circumvents the coarse-grained CFI. In contrast to the complier-based CFIs, RecurFI can be directly applied to binary codes without a source code. Also, in contrast to the EMET, RecurFI effectively defends against recent exploits. The idea underlying RecurFI is intuitive. While most ROP codes are executed by several gadgets chained together, which makes it difficult for all gadgets used in an ROP attack to have a general control-flow. Therefore, we recursively analyze all return addresses on the call stack when the API is called, and check that each return address points to a valid address after the `CALL` instruction. If there is at least one return address referring to an invalid address, the stack is judged as having been falsified and RecurFI generates an exception and notifies the user. The current system is based on Windows, but it is similarly applicable to Linux.

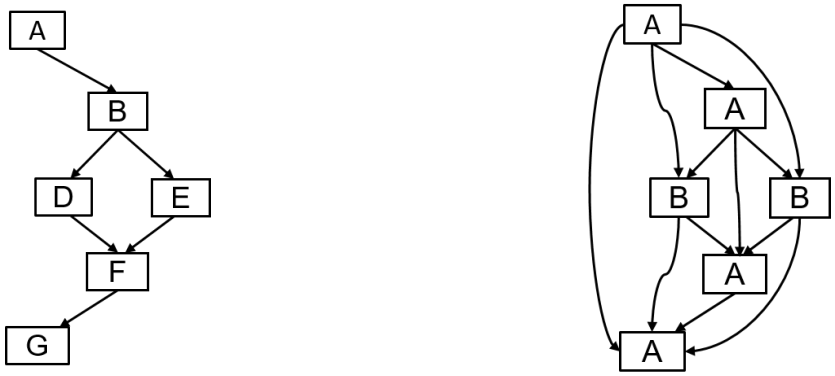
1.1 Thesis Outline

This thesis consists of as follows: While Chapter 1 outlines CFI, we describe in more detail existing coarse-grained CFI solutions and their inefficiency approaches in Chapter 2. Starting from Chapter 3, we delve into the new technique we propose in this thesis, called RecurFI. Evaluation is followed to present the effectiveness and performance of our method. The remaining chapters include discussion of our work, limitation, and future work along with the conclusion.

Chapter 2. Related Work

2.1 Coarse-grained Control-Flow Integrity (CFI)

The CFI security policy[1] dictates that software execution must follow a path of a pre-defined Control-Flow Graph(CFG) determined ahead of time. To ensure a program follows a valid path in the CFG, CFI gives labels at the beginning of destinations basic blocks for all branch instructions. At runtime, CFI security policy checks destination IDs before taking the branch to ensure that the control-flow follows a legitimate path of the CFG. Any deviation from the CFG leads to a CFI exception and process is terminated. CFI is theoretically a perfect detection mechanism. However, there are two fundamental limitations to applying it to a real program. First, CFI is slow. This is because the instrumentation is required for all indirect branch statements in the target program. CFI gives an ID to all destination BBLs and, on indirect jumps, an average of 21% overhead to ensure that the ID matches the source and destination [1]. Second, CFI technology is hard to apply directly to binary code. To apply CFI to a program, the control flow graph (CFG) of the program is necessary, but it is impossible to restore the complete CFG from the binary code.



(a) CFI

(b) coarse-grained CFI

Figure 2.1: The difference between CFI and coarse-grained CFI

Recently, coarse-grained CFI has been developed to solve inherent problems practically. The coarse-grained CFI has a heuristic-based security policy and a more relaxed security policy than the original CFI. Coarse-grained CFIs supported by compilers such as GNU C Compiler (GCC) and Low-Level Virtual

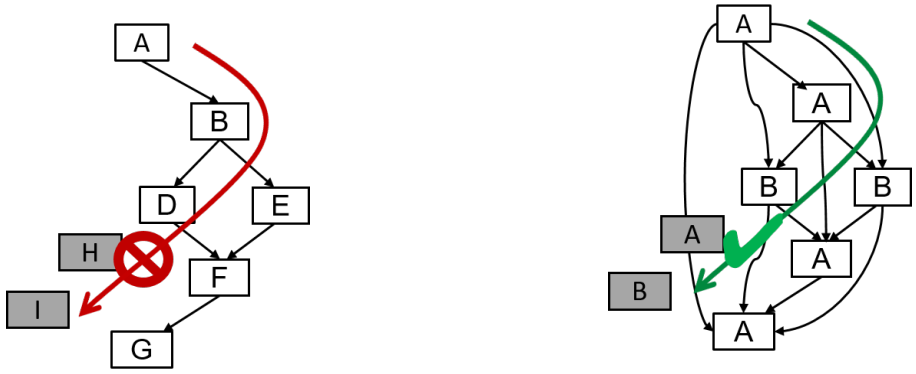
Machine (LLVM) [3] and Control Flow Guard (CFG) [8] which is a coarse-grained CFI applied to internet explorer, have been applied to commercial products on the strength of having lower overhead. These two techniques, a source code-based coarse-grained CFI is applied by creating at the time of compiling and linking a trampoline saving the destination address of all indirect jumps and the destination address of the vtable (virtual function table) to perform a target check before an indirect branching. If an indirect branch occurs during program execution, an address on the trampoline and a target of an indirect branch are compared to check whether the target is valid. If the target address is on the trampoline, the indirect branch considers it as a normal indirect branch. However, if the target address is not on the trampoline, it is judged as an abnormal execution flow. However, these technologies have the limitations that the CFI is not applicable to legacy products because they are executed at a source code level.

A binary-based coarse-grained CFI defines a security policy based on the heuristic of execution flow. One of the heuristic characteristics is defined a valid return address. The return address in a normal function call is the address of the instruction to be executed after calling the function with the `CALL` instruction. Therefore, the return address must have the address value of the instruction immediately following the `CALL` instruction. Considering this, Ivan Fratric's proposed ROPGuard [4] defines a valid return address as a return address with the address value of the instruction immediately following the call instruction, and the normal execution flow always has a valid return address.

A heuristic in which a short gadget is executed with multiple indirect branches differentiates a normal execution flow from an ROP. Kbouncer [7] and ROPecker [9] detect an ROP attack by this feature. In their studies, we analyzed how many instructions the ROP gadget has and how many consecutive indirect branches should be executed. Based on this analysis, their studies set optimal thresholds to minimize false. In Kbouncer [7], if fewer than eight instructions are executed continuously as more than 20 indirect branches during program execution, this execution determines ROP attack. On the other hand, the ROPecker [9] considers that the ROP attack is performed when the instruction of fewer than 11 instructions is executed as six or more indirect branches consecutively.

Besides, CFI examines unique IDs on every indirect branch, but since coarse-grained CFI has a heuristic-based security policy, it takes less time to check the integrity of the destination. Numerically, the overhead of 20% of the existing CFI was reduced to 0.48 – 2 % overhead in the coarse-grained CFI and the coarse-grained CFI is applied to commercial products in the industry[8, 5].

2.2 Inefficiency of Coarse-grained CFI



(a) CFI blocks sophisticated exploit

(b) Coarse-grained CFI cannot block sophisticated exploit

Figure 2.2: Inefficiency of Coarse-grained CFI

The loose security policy of the coarse-grained CFI has resulted in a new type of security problem. The coarse-grained CFI allows more paths than CFG, an attacker can perform sophisticated attacks that perform malicious actions within the paths allowed by the coarse-gained CFI. In various studies [10, 11, 12, 13], it has been shown that ROP attacks using the following three type of gadgets can bypass existing coarse-grained CFIs.

Listing 2.1: Example of a call gadget

```
LEA eax,[ebp-34h];
PUSH eax;
CALL esi;
RET;
```

First, a call gadget [10] is a gadget that can call the Windows API. Conventional ROP attack executes a Windows API by jumping to an address of a Windows API. Such type of attack can be detected by existing coarse-grained CFI, because the `RET` instruction does not refer to a value immediately after a `CALL` instruction. To circumvent this detection, the attacker uses a gadget that calls a Windows API by using a `CALL` instruction. In this way, the existing coarse-grained CFI consider the return address as a valid one, because the return address is immediately preceded by a `CALL` instruction. Such an evolved ROP using a particular gadget may have the same semantic as the previous attack, but can perform the

ROP attack without being detected by the coarse-grained CFI. Second, a call preceded gadget [11] refers to a gadget that starts with an instruction immediately following a `CALL` instruction and does not include another branch between the `CALL` instruction and a `RET` instruction. If a call-preceded gadget is used in an ROP attack, the existing coarse-grained CFI recognizes the return address as a valid return address because the return address has the address of the instruction immediately after the call instruction. The existing coarse-grained CFI is judged to be a normal execution flow if the execution flow has a certain number of consecutive valid return addresses. Therefore, if more than a certain number of call-preceded gadgets are used in ROP attack, coarse-grained CFI judges ROP attack as normal control flow. A Turing-complete call-preceded gadget may be found in shell32.dll [10]. Lastly, a Long-NOP gadget [10, 12, 13] refers to a long, meaningless gadget that does not cause a side effect. Many approaches [7, 4, 5] have been conducted to determine the optimal length of a gadget and the optimal gadget-chain length to detect ROP attacks by the ROP feature that short gadgets are carried out by many indirect branches. However, existing coarse-grained CFI which is the technologies based on the heuristic can be circumvented if there is at least one NOP gadget included in an ROP chain because The ROP gadget breaks assuming that the ROP gadget is short.

All the heuristics-based security policies of recently developed coarse-grained CFI can be evaded by using the three types of gadgets. It has been shown that the gadgets satisfying the conditions mentioned above may be found in a Turing-complete way, but not in all registers. Therefore, the attack performed by attackers is limited. Also, since the coarse-grained CFI are evaded only by modifying the ROP gadget, the ROP attacks still have an abnormal control flow executed by a return address.

Chapter 3. RecurFI

The system proposed in this thesis, RecurFI, can effectively defend against ROP attacks employing the evolving techniques for circumventing the coarse-grained CFI and minimize the increase in performance caused by the defense. The attacks to be detected by RecurFI are limited to ROP attacks. Control-flow hijacking attacks that do not use return addresses such as Jump-Oriented Programming [14] and Printf-Oriented Programming [15] are not the scope of our research. Also, the attacker model [16] with strong assumption such as attack using register spilling when the attacker can adjust arbitrary memory to an arbitrary value is beyond the scope of this thesis. Figure 3.1 shows the overall structure

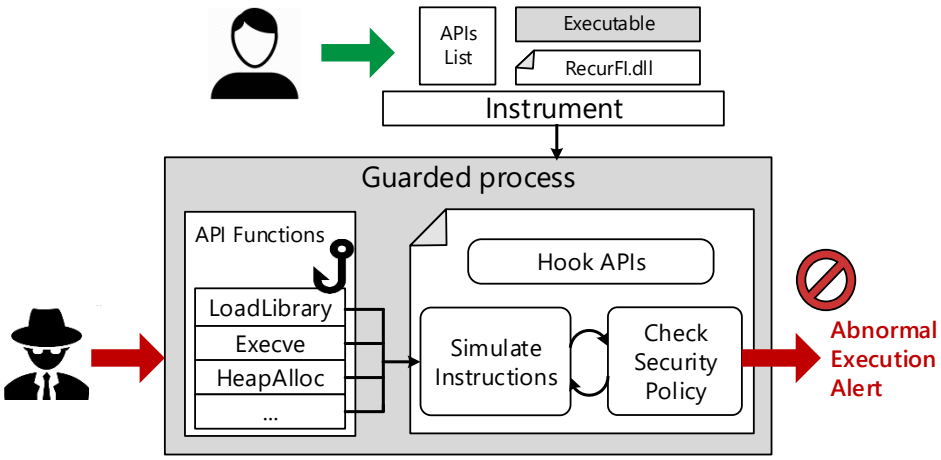


Figure 3.1: Design overview of RecurFI

of RecurFI which receives an executable file to be protected from the user as input and then executed. RecurFI.dll, which protects the control flow of the program, is then injected into the process. The injected DLL first hooks APIs related to the system resources that are necessary for an attacker to control the system. When the hooked API is called during execution, the control of the process is transferred to RecurFI.dll, where the instruction is analyzed, and the return is checked to determine whether the API has normally been called.

We defined the behavioral heuristics by capturing general patterns of normal execution flow. For example, an evolved ROP attack can bypass a coarse-grained CFI using specific gadgets mentioned in

Section 2.2, but it is difficult for all gadgets used in an attack to have a normal control flow. Based on this, we defined the behavioral heuristics by capturing general patterns of normal execution flow and recursively examined whether all the gadgets satisfy the behavioral heuristics. If the execution flow is determined to be a normal execution flow, the RecurFI.dll is returned to the hooked API, and the hooked API is resumed. On the contrary, if the execution flow is determined as an ROP attack, RecurFI notifies the user of the abnormal execution flow and terminates the process. This system flow has the advantage of reducing performance overhead by performing program analysis only when certain APIs are called.

3.1 Runtime Monitoring

The RecurFI system observes the programs that the user wants to protect at runtime. In RecurFI, when a specific Windows API is called, security policy is applied to check whether an execution is normally performing at every time.

For example, an attacker can invoke the `virtualProtect` function to execute injected code by making the injected code into an executable memory. In addition, the Windows APIs that create and allocate heap such as `HeapCreate`, `VirtualAlloc` or `writeProcessMemory` functions are often used to load shellcode into memory, and the `LoadLibrary` function is used to inject DLLs with attack code into the process. The shellcode injected into the memory uses the Windows API associated with the process and file to create a malicious process on the victim or to create another malicious file. RecurFI hooked up such a Windows API and enforced our security policy only when the API is called. An attacker should inevitably call the Windows APIs related to system resources, such as socket related APIs or file system related APIs, to control the computer of a victim perfectly by downloading a dropper on the victim's computer or by executing a code that downloads a malicious execution file. Therefore, in RecurFI, when the system resource related Windows API and the user specified function are called, we need to check whether the function was normally called. This has the advantage that the existing CFI has less overhead than instrumenting all indirect branches. The Windows APIs specified in RecurFI are shown in Table 3.1, which are functions related to memory, process, library, thread, and file system.

Table 3.1: Windows APIs commonly used for malware

```
kernel32.dll:VirtualProtect
kernel32.dll:VirtualProtectEx
kernel32.dll:VirtualAlloc
kernel32.dll:VirtualAllocEx
kernel32.dll:HeapCreate
ntdll.dll:RtlCreateHeap
kernel32.dll:CreateProcessA
kernel32.dll:CreateProcessW
kernel32.dll:CreateProcessInternalA
kernel32.dll:CreateProcessInternalW
kernel32.dll:LoadLibraryA
kernel32.dll:LoadLibraryW
kernel32.dll:LoadLibraryExA
kernel32.dll:LoadLibraryExW
kernel32.dll:CreateRemoteThread
kernel32.dll:WriteProcessMemory
kernel32.dll:CreateFileA
kernel32.dll:CreateFileW
kernel32.dll:WriteFile
kernel32.dll:WriteFileEx
```

3.2 Static Analysis

After the Windows API is called, program execution control passes to RecurFI.dll and static analysis is performed. Since all gadgets used in ROP attacks that bypass coarse-grained CFIs have difficulty in having a general control flow, we focus on this heuristic characteristic. In addition, we extend the static analysis method of existing coarse-grained CFI to check "all gadgets". Inspecting all gadget help us to effectively detect the advanced ROP attack. RecurFI.dll disassembles the code after the PC (Program Counter) register value at the time the windows API was hooked. Because we analyze it statically, we can know what instruction to execute after the hooked function is returned without executing the program. However, disassembling the instruction is limited to find all instructions to be executed after API call. In particular, since the return address executed after the `RET` instruction is statically unknown information, we use the stack pointer and call stack information when the API is called to find the return address. **Algorithm 1** is to calculate the change of stack pointer value when knowing call stack information and stack pointer. First, we disassemble the binary code and then infer how the instruction changes the stack pointer. The instruction to change the stack pointer among the instruction exist in x86 is as follows: The `LEAVE` instruction sets a stack pointer value by adding 4 byte to a base stack

Algorithm 1: Resolving StackPointer Algorithm

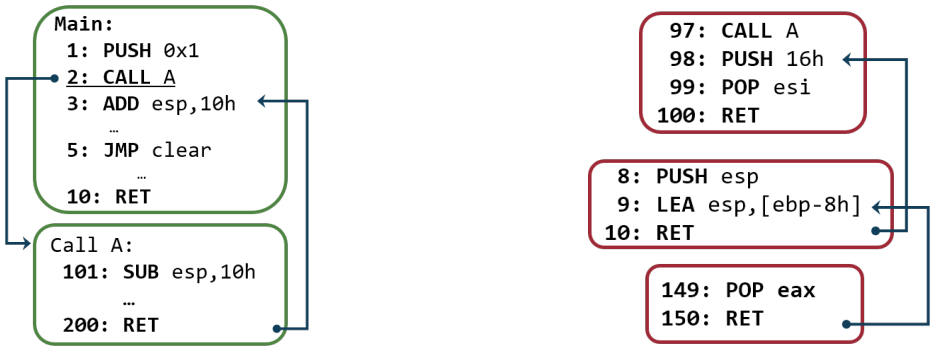
```
1 Function simluateInstruction(pc , stackPtr, FramePtr):
2     opcode, pc ← disassemble(pc);
3     if opcode is push then
4         stackPtr ← stackPtr - 4;
5     if opcode is leave then
6         stackPtr ← FramePtr;
7         stackPtr ← stackPtr + 4;
8     ...
9     return opcode, pc, stackPtr, FramePtr
```

value. The `LEAVE` instruction requires a stack base pointer value, and the following instructions perform an operation between a specific constant and a stack pointer. The `CALL` instruction subtracts 4 byte by pushing a return address. The `RET` instruction adds 4 byte by popping a return address. The "`ADD` esp, const" instruction increments the stack pointer by the const value, and the "`SUB` esp, const" instruction decrements the stack pointer by the const value. `PUSH` and `PUSHA` reduce the stack pointer by 4 byte and 32 byte respectively. `POP` and `POPA` increase the stack pointer by 4 byte and 32 byte, respectively. The `MOV` and `LEA` instruction turn an esp value to an operand value.

The stack pointer is calculated when the specific instruction is encountered. In particular, when the `RET` instruction is encountered, the return address is stored in the pointer address obtained by adding 4 to the stack pointer. We can obtain the value of return address with the value of stack pointer and the call stack information when the hooked Windows API is called. If **Algorithm 1** is called with the found return address as the start address, the next gadget can be analyzed in the same way. Therefore, we can recursively call **Algorithm 1** to keep track of the return address and get all the gadgets.

3.3 CFI Policy

Heuristic observation by the general patterns of normal execution, the differences between a normal execution and an ROP attack were analyzed to determine an appropriate security policy. First, a normal control flow is differentiated from a control flow of ROP attack concerning the presence of a valid return address. A valid return address which is proposed in ROPGuard [4], satisfies the following two conditions. First, a valid return address should be an executable address. If a return address is a memory address that does not allow the code execution, the information included in the stack must have



(a) In normal function call, return addresses target vaild (b) In ROP attack, return address is valid or invalid

Figure 3.2: Differences between normal function call and ROP attack

been corrupted by a memory corruption bug. The security policy that prevents a particular memory area from execution is supported by Data Execution Prevention(DEP) and RˆW(Write XOR Execute) about the hardware. However, a software-based security policy was added to because a particular memory area may become executable by many attacks using the `virtualProtect` function for evading the mitigation. Second, a return address stored in the stack must have the address of the preceded instruction immediately after the `CALL` instruction. In a general control flow, a return address is the address that will be executed after the function is called by a `CALL` instruction, and thus a return address refers to an instruction that immediately follows a `CALL` instruction. Based on this heuristic, the call-preceded policy defined by the heuristic regulates that an instruction of a return address should be an instruction that immediately follows a `CALL` instruction. Thus, the return address that satisfies the call-preceded policy is a valid return address.

A general control flow may be differentiated from an ROP concerning the call-preceded policy. An ROP attack includes a code address that the attack wants to execute as a return address and does not contain a valid return address in general. However, to make the difference undetected, an attacker includes a valid return address by using a gadget that starts with an instruction that immediately follows a `CALL` instruction, as described in Section 2.2. Therefore, the execution flow of an ROP attack can have valid or invalid return addresses. On the contrary, a general control flow must not include an invalid return address and must include only valid return addresses. This difference between a normal control flow and an ROP attack can be used to define a security policy.

Other difference between the normal execution and the ROP attack can be seen by the existence of the `JMP` instruction. A normal program converted by an assembler includes many `JMP` instruction. On the

contrary, an ROP attack, a return-based program, does not contain branches in the gadgets except the `RET` instructions. Therefore, if the `JMP` instruction exists while the program is running, it can be regarded as a normal execution flow. If the `JMP` instruction does not exist, it can be guessed by the ROP gadget.

Security policy Most ROP attacks work by chaining several gadgets. The NOP-gadget used in the evolved ROP attack is not related to the control flow, so it is easy for the attacker to bypass the existing coarse-grained CFI even if only one NOP-gadget is used. On the other hands, since there are not many call-preceded gadgets, an attacker can use this gadget to circumvent the coarse-grained CFI to some extent. However, it is difficult to use an appropriate gadget to perform an attacker's behavior, and there is a limitation in using a general gadget. Because there is a realistic premise that it is difficult for an attacker to build an ROP chain only with a gadget that starts with an instruction immediately after the `CALL` instruction in an ROP attack, examination of all gadgets will reveal a fundamental difference between normal execution flow and ROP. Based on this, we *recursively* check whether "all gadgets" satisfy the normal control flow. We define two security policies based on the difference between the normal execution flow and the ROP, and apply the following security policies to all gadgets.

- First, a normal control flow must include a `JMP` instruction.
- Second, the abnormal control flow must have an invalid return address.

The first security policy is that the `JMP` instruction frequently appears in the normal execution, but not

Algorithm 2: RecurFI Enforcement Algorithm

```
1  Function simulateRetn(pc , stackPtr, FramePtr):
2      pc ← stackPtr + 4 ;
3      while true do
4          opcode, pc, stackPtr, FramePtr ← simulateInstruction(pc, stackPtr, FramePtr);
5          if opcode is return then
6              if checkPolicyViolation(stackPtr) then return false;
7              else return simulateRetn(pc, stackPtr, FramePtr);
8          else if opcode is jump then return true;
9  Function checkPolicyViolation(returnAddress):
10     if IsAddressExecutable(returnAddress) then return false;
11     else if PrecededByCall(returnAddress) then return true;
```

in the ROP attack. The second security policy is defined as a valid return address is a property that can have both normal control flow and ROP, but an invalid return address is a property with ROP only. **Al-**

gorithm 2 is to recursively check all gadgets and apply the security policy. the `simulateInstruction` function in Line 4 in **Algorithm 2** is called to disassemble the instruction and calculate how the instruction changes the stack pointer. This function is called continuously in the while statement, by disassembling the instructions one by one (Lines 3 and 4 in **Algorithm 2**).

Analyzes gadget instructions sequentially and compares and analyzes the instructions as `JMP` instructions. When the `JMP` instruction is encountered, the execution flow analyzed according to the first security policy is determined to be a normal execution flow, not an execution flow executed by the gadget.

When the `JMP` instruction is encountered, the `JMP` instruction is an instruction that is not likely to be included in the gadget. Therefore, the execution flow to be analyzed now is determined to be a normal execution flow, not an execution flow to be executed by the gadget. RecurFI terminates the `SimulateRetn` function and continues the target program (line 8 in **Algorithm 2**). On the other hand, when the gadget's instruction is analyzed sequentially and the `RET` instruction, which means the end of the gadget, is encountered (Line 6 in **Algorithm 2**), the `checkPolicyViolation` function checks whether the return address is a valid return address.

If the return address is not valid according to the second security policy, the stack is corrupted to determine that the ROP attack is in progress, and an exception is generated within the `checkPolicyViolation` function to notify the user. However, even if the return address is valid, it is impossible to know whether the execution flow currently analyzed is the normal execution flow or the execution flow executed by the gadget. This is because the ROP can also have a valid return address in the normal control flow. We call the `simulateRetn` function again with the return address as the start address. Recursively check all gadgets until it is determined to be a normal execution flow or ROP. Table 3.2 is Source code of the SimulateRetn fuction which describes in **Algorithm 3.2**.

Table 3.2: Source code of the SimulateRetn fuction

```
int SimulateRetn(unsigned long FramePtr,unsigned long stackPtr, unsigned long
    stackIncrement)
{
  unsigned long returnAddress;
  unsigned long eip;
  int simret;
  returnAddress = *((unsigned long *)stackPtr);
  stackPtr += stackIncrement + 4;

  eip = returnAddress;
```

```
  while(1) {
    simret = SimulateStackInstruction(NULL, &eip,&FramePtr, &stackPtr, &
        stackIncrement);
    if(!simret) return 1;
    if(simret == 2) {
      returnAddress = *((unsigned long *)stackPtr);
      if(!CheckReturnAddress(returnAddress)) return 0;
      else return SimulateRetn(NULL,stackPtr,0);
    }
  }
}
```

Chapter 4. Evaluation

The current prototype of RecurFI can only protect 32-bit processes in the Windows operating system. But its protection mechanism is equally applicable to 64-bit processes and Linux as well.
The RecurFI consists of two components: 1) instrument, 2) dll with security policy. Based on ROPGuard's instrument and static analysis, we analyze all gadgets and apply extended security policy.
We performed experiments on Intel Xeon E3-1231 v3 CPU, 2GB RAM and Microsoft Windows 7 SP1 32-bit environment to measure the performance and security of RecurFI.

4.1 Performance Evaulation

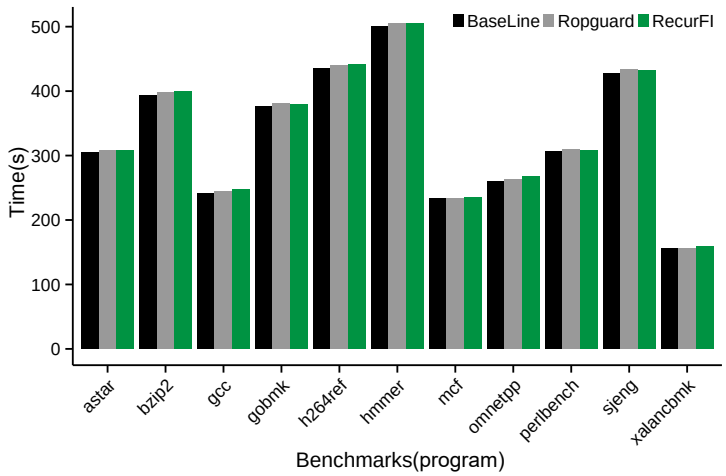


Figure 4.1: Performance evaluation of both ROPGuard which is the coarse-grained CFI of Microsoft's EMET and RecurFI

To assess the practicality of the RecurFI system, the performance of 11 benchmarks of the SPEC CINT2006 was measured and analyzed in the cases where RecurFI was applied or not. Figure 4.1 shows the experimental results. The black bars in Figure 4.1 represent the benchmarks to which the RecurFI system was not applied. The gray bars represent the benchmarks to which ROPGuard, coarse-grained CFI of Microsoft's EMET, was used. The green bars represent the benchmarks to which the RecurFI system was applied. The experiment was performed three times in the same environment. The total duration of the experiment was 9.2 hours.

Table 4.1: Execution time with ROPGuard and RecurFI

Bechmarks	Baseline Execution time (s)	ROPGuard Execution time (s)	RecurFI Execution time (s)
astar	304.6	307.6	307.6
bzip2	393.6	398.3	399.6
gcc	241.3	244.6	248.0
gobmk	376.0	380.3	379.6
h264ref	435.3	440.0	441.6
hmmer	500.0	504.6	505.3
mcf	234.0	233.3	234.6
omnetpp	260.6	263.3	267.0
perlbench	306.3	309.3	308.0
sjeng	428.0	433.6	432.6
xalancbmk	155.6	156.6	158.6

The execution times of the baseline, ROPGuard, and RecurFI in the 11 benchmarks are summarized in Table 4.1. By subtracting the time of ROPGuard and REcurFI from the baseline, we can observe the time overhead imposed on each system. As a result, the time overhead of the RecurFI system ranged from a maximum of 2.6% to a minimum of 0.5% concerning the native overhead. The average overhead of the 11 benchmarks was 1.34%. The average overhead of ROPGuard was 0.9%. Experimental results show that RecurFI can detect ROP attacks that can bypass existing coarse-gained CFI with a time overhead of 0.4%. For the three benchmarks, RecurFI took less time than ROPGuard. This is because the security policies are different. ROPGuard checks the integrity of return addresses in 10 instructions. However, RecurFI does not limit the number of instructions to be analyzed and checks the integrity of the program by using the `JMP` instruction or an invalid return address. Therefore, we confirmed that RecurFI could check program integrity more quickly if the `JMP` instruction or invalid return address comes before ten instructions.

4.2 Security Evaulation

To evaluate the effectiveness of RecurFI, we examined whether it could prevent ROP attacks that could bypass existing coarse-grained CFI. In the experimental environment, we have installed the Internet Explorer 8, EMET, which only checks the validity of the return address, and RecurFI, on the Windows 7 operating system. EMET and RecurFI are runtime monitoring that prevents Windows application

Table 4.2: ROP gadget

DLL name	address	call gadget	address	call-preceded gadget
MSVCR71.dll	0x7c3528dd	CALL virtualprotect; LEA ESP,[EBP-58h]; POP EDI;POP ESI; POP EBX; LEAVE; RETN;	0x7c341555	RETN;
MSVCR100.dll	0x78aef0df	CALL virtualprotect; LEAVE; RETN;	0x78b04220	RETN;

including Internet Explorer (IE) from being exploited. We also made three exploits that exploit the three vulnerabilities of IE 8 (CVE-2012-4969[17], CVE-2013-1347[18], CVE-2013-2551[19]) in IE 8 using the evolved ROP.

The attack code utilized the memory corruption vulnerability to write the ROP gadget and shellcode into memory. Since the stack or heap memory area is usually a memory area that is not executable, shellcode cannot be executed in memory. However, if the attacker can turn the shell code area to an RWX mode by calling the `virtualProtect` API, then a shell code area may be executed, the control flow is set to the shell code area by using the ROP gadget. The shellcode executes `calc.exe` by using the `WinExec` function.

The exploits that we made attack has the same manner as above, however, gadgets can bypass the existing coarse-grained CFI. Table 4.2 describes in gadgets used by the exploit and the name of DLL where the gadget resides. The gadgets were constituted with call gadgets and call-preceded gadgets to circumvent the existing coarse-grained CFI. The call gadget was used to call the `virtualProtect` function. The EBP value should be adjusted before using the call gadget to inject the parameters normally. Several call-preceded gadgets were connected to create a NOP gadget. The three vulnerabilities were supposed to load different DLL, and the gadgets were searched in MSVCR71.dll and MSVCR100.dll.

We exploited the experimental environment where coarse-grained CFI and RecurFI are installed with the three attack codes made above. In the experimental environment, we compared the effectiveness of the two programs by installing ROPGuard, which are open source coarse-grained CFI source code and

Table 4.3: The effectiveness of RecurFI

Vulnerability	Operating System and Vulnerable application	EMET	RecurFI
CVE-2012-4969[17]	windows7 SP1 with IE8	✗	✔
CVE-2013-1347[18]	windows7 SP1 with IE8	✗	✔
CVE-2013-2551[19]	windows7 SP0 with IE8	✗	✔

RecurFI. Table 4.3 shows the effectiveness of RecurFI. The EMET misjudged all three ROP attacks as a normal control flow, failing to secure the CFI of the IE. On the contrary, RecurFI having more powerful detection policies detected all the ROP attacks, securing the CFI of the IE.

4.3 Empirical Case Study

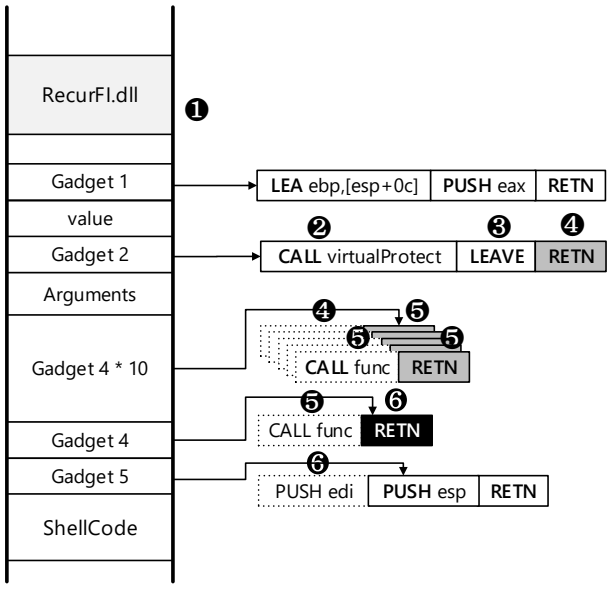


Figure 4.2: The process of attacking IE vulnerability and applying RecurFI's security policy

Figure 4.2 illustrates the process of attacking IE with CVE-2013-1347 vulnerability and RecurFI's security policy applied to the process by ROP that can circumvent existing coarse-grained CFI for the detailed explanation. RecurFI receives an IE executable file from a user and creates an IE process, and inserts RecurFI.dll. We describe the process of Figure 4.2 each step.

Step 1. The RecurFI.dll inserted into the IE process instruments the API specified by the user including the `virtualProtect` function. Gadget1 modifies framePtr so that the `virtualProtect` function

called in gadget2 can read the parameters properly. Gadget1 does not have a valid return address, but we do not perform a return address integrity check because these are instructions that are executed before the windows API is called.

Step 2. When the `CALL virtualProtect` instruction of gadget 2 is called, execution control is passed to RecurFI and **Algorithm 2** is executed. The return address of the `virtualProtect` function points to the `LEAVE` instruction. Since the instruction is located after the `CALL` instruction and is an executable address, RecurFI determines that the `virtualProtect` function has been called normally. RecurFI recursively checks subsequent instructions because it only considers the `JMP` instruction to be a normal execution flow.

Step 3. Since the `LEAVE` instruction changes the stack pointer value, it records that the value of the stack base register is changed to the stack base register + 4 value according to **Algorithm 1**.

Step 4. When the `RET` instruction is encountered, the return address is validated. The return address points to the `RET` instruction with gadget4, and since the instruction is immediately after the `CALL` instruction, RecurFI determines that the return address is valid.

Step 5. Gadget 4 is a gadget consisting of only one `RET` instruction that comes immediately after the `CALL` instruction, and is NOP gadgets with no semantic meaning. ROPGuard, a conventional coarse-grained CFI, performs a clean lookup of the `RET` instruction in 10 instructions after an API call. Therefore, if the attacker repeats gadget four more than ten times, ROPGuard judges that the control flow has a valid return address only and does not apply CFI technology from gadget 5. On the other hand, since RecurFI recursively checks the return address, it considers that each return address is valid and continues the analysis without judging it as a normal execution flow.

Step 6. After running gadget 4 10 times, gadget 4, which runs 11 times, has gadget five as the return address, and RecurFI checks the validity of the return address. Since the return address of the `RET` instruction corresponds to the "`PUSH` esp" instruction and this instruction follows the "`PUSH` edi" instruction, the return address is not valid according to RecurFI's security policy, so it is judged that the stack is corrupted. After the process is terminated, the user is notified.

Chapter 5. Discussion

We discuss the limitations and suggest possible solutions. There are few limitations that RecurFI system has due to either the underlying design of the system. We describe several issues and explain how to improve to make a more effective RecurFI system.

5.1 Limitations and Future work

Unhandle the JMP and CALL instructions: We considered that the potential gadget does not have a `JMP` or `CALL` instruction because execution of ROP is preceded by `RET` instruction. Thus, we assume that there is no branch statement except `RET` instruction in ROP attack. However, such assumption leads the RecurFI not to detect specific attacks such as JOP [14] that the instruction is executed by the `JMP` instruction instead of the `RET` instruction.

RecurFI assume that the gadget is connected via the `RET` instruction, checks the gadget recursively by following the `RET` instruction. This mechanism cannot detect JOP attacks because gadgets running with the `JMP` instruction cannot be checked recursively. RecurFI might determine the execution of ROP is normal execution flow.

In order to detect attacks such as JOP, it is necessary to be able to trace the destination of the `JMP` instruction. As in recurFI followed by the `RET` instruction, the new method disassembles the target address of the `JMP` instruction in order to detect a JOP attack. In addition, the CFI security policy can be defined by extracting features of a JOP attack that are not in the normal execution flow.

Operating system and architecture dependency: The current prototype of RecurFI can only protect 32-bit processes in the Windows operating system. But its protection mechanism, **Algorithm 2** is equally applicable to 64-bit processes and Linux as well. To support x64 architecture, the way to disassemble with x86 architecture should be adapted to the x64 architecture which has a larger size and number of general purpose registers than x86 architecture and should be tailored to increase the number of extra registers. Also, the calculation of the stack pointer must be adapted to the 64-bit operation.

Chapter 6. Conclusion

In our research, we presented RecurFI, a novel coarse-grained CFI solution based on heuristics of code reuse attack. RecurFI provides an effective runtime monitoring mechanism to detect advanced ROP attacks.

We evaluated the effectiveness of RecurFI in the Windows 7 operating system with IE environment. From the experiments, we confirmed that RecurFI is effective and showed that RecurFI could detect all the exploits the existing coarse-grained CFI solutions cannot detect in our experiment. We also measured the RecurFI can defense the advanced ROP attack less than 2% runtime overhead.

We can claim that the existing coarse-grained CFIs are good in performance. Since the attack techniques have been able to bypass all of the proposed solutions, coarse-grained CFIs needed a sweet spot between new security and performance. The proposed idea, RecurFI is believed to be a more secure detection method than the existing coarse-grained CFI. However, RecurFI is not able to completely detect the known and unknown attacks, but it can detect the wild ROP attack in practice.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 2005, pp. 340–353.

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security.* ACM, 2007, pp. 552–561.

[3] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 941–955.

[4] I. Fratrić, "Ropguard: Runtime prevention of return-oriented programming attacks," 2012.

[5] "Microsoft Enhanced Mitigation Experience Toolkit," https://www.microsoft.com/emet.

[6] M. Prasad and T.-c. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," in *USENIX Annual Technical Conference, General Track*, 2003, pp. 211–224.

[7] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 447–462.

[8] "Control flow guard," https://msdn.microsoft.com/ko-kr/library/windows/desktop/mt637065(v=vs.85).aspx, accessed: 2016-10-28.

[9] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[10] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[11] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 575–589.

[12] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 417–432.

[13] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 385–399.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[15] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[16] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 952–963.

[17] "Ms12-063 microsoft internet explorer execcommand use-after-free vulnerability," https://goo.gl/Y5r0et.

[18] "Ms13-038 microsoft internet explorer cgenericelement object use-after-free vulnerability," https://goo.gl/1SoQM1.

[19] "Ms13-037 microsoft internet explorer coalinedash stylearray integer overflow," https://goo.gl/votFNZ.

Acknowledgments in Korean

언제나 저를 바른 길로 이끌어 주시는 김광조 교수님께 큰 감사함을 느낍니다. 그리고 저에게는 과분할 정도의 도움을 주신 차상길 교수님과 김용대 교수님께 감사드립니다. 끝으로 오늘의 제가 있을 수 있도록 사랑으로 키워 주신 가족들에게 감사드립니다. 저의 이 작은 결실이 그분들께 조금이나마 보답이 되기를 바랍니다.

Curriculum Vitae in Korean

이 름: 홍 진 아

생 년 월 일: 1991년 6월 3일

전 자 주 소: jina3453@kaist.ac.kr

학 력

2007. 3. – 2010. 2. 천안 북일여자고등학교

2010. 3. – 2015. 2. 충남대학교 컴퓨터공학과 (B.S.)

2015. 3. – 2017. 2. 한국과학기술원 정보보호대학원 (M.S.)

경 력

2014. 12. – 2015. 2. 한국전자통신연구원(ETRI) 인턴

연 구 업 적

1. **홍진아**, 차상길, 김광조, *RecurFI: 윈도우 바이너리에 대한 효율적 제어흐름 무결성 검사 시스템*, 한국정보보호학회 동계학술대회(CISC-W'16), Seoul (Korea), December, 2016.
2. **홍진아**, 김광조, 김용대, *상용 exploit탐지 제품의 오탐율 실험결과*, 한국정보보호학회 하계학술대회(CISC-S'16), Busan (Korea), June, 2016.
3. **홍진아**, 김광조, *메모리 접근 분리기법의 활용에 대한 비교 분석*, 정보보호학회학술발표회 영남지부, Busan (Korea), January, 2016.
4. Kyung-min Kim, **Jina Hong**, Kwangjo Kim, and Paul D.Yoo, *Evaluation of ACA-based Intrusion Detection Systems for Unknown-attacks*, Symposium on Cryptography and Information Security (SCIS 2016), Kumamoto (Korea), January, 2016.
5. 김경민, **홍진아**, 김광조, *개미군집 알고리즘과 의사결정트리를 활용한 알려지지 않은 공격 탐지 시스템*, 한국정보보호학회 동계학술대회(CISC-W'15), Seoul (Korea), December, 2015.