This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2020.DOI

Value-based Constraint Control Flow Integrity

DONGJAE JUNG[1], MINSU KIM[2], JINSOO JANG[3], AND BRENT BYUNGHOON KANG.[4], (Member, IEEE)

[1]Graduate School of Information Security, School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea (e-mail: jjp1018@kaist.ac.kr)
[2]S2W LAB Inc., Seongnam, Republic of Korea (e-mail: minsu@s2wlab.com)
[3]Department of Computer Science & Engineering, Chungnam National University (CNU), Daejeon, Republic of Korea (jisjang@cnu.ac.kr)
[4]Graduate School of Information Security, School of Computing, Korea Advanced Institute of Science and Technology, Daejeon, Republic of Korea (e-mail: brentkang@kaist.ac.kr)

Co-Corresponding author: Jinsu Jang (e-mail: jisjang@cnu.ac.kr).
Corresponding author: Brent Byunghoon Kang (e-mail: brentkang@kaist.ac.kr).

ABSTRACT Control flow integrity (CFI) is a generic technique that prevents a control flow hijacking attacks by verifying the legitimacy of indirect branches against a predefined set of targets. State-of-the-art CFI solutions focus on reducing the number of targets using the context of a program such as the path to the indirect branch and the origin of the code pointer. However, these solutions work with an impractical assumption that the attacker only compromises control data; non-control data such as condition data that can also be abused by attackers are not considered. To overcome these limitations, in this paper, we propose value-based constraint CFI (vCFI) to improve the effectiveness of CFI by retrieving and protecting all data that can potentially be manipulated for control flow hijacking. We first perform static analysis such as dependency, condition, and data analyses to derive all control flow-related data. Then, vCFI protects these data during runtime by instrumenting a program to be hardened. We implemented vCFI as a compiler extension and evaluated its performance using SPEC CPU2006. The performance degradation caused by adopting vCFI was reasonable, and the average overhead was 13.6%.

INDEX TERMS Control flow hijacking, control flow integrity, non-control data, program analysis

I. INTRODUCTION

CONTROL flow is a general target for attackers attempting to compromise applications. For instance, an attacker can exploit a stack buffer overflow bug, overwrite the return address in the stack frame, and thus hijack the control flow of the application after the manipulated return address is used by indirect branch instructions (e.g., ret on x86). Hijacking control flow implies that the attacker has full control over the application as the attacker can now execute arbitrary logic. Therefore, protecting the control flow is considered key in software security.

Control flow integrity (CFI) [1] is the first approach that proposed a defensive measure to protect the control flow. The key idea is checking the legitimacy of a certain branch when an attack occurs. A group of valid branch targets is derived by a static analysis that constructs the control flow graph (CFG). Since the proposal of CFI, a long stream of work has been conducted to enhance the efficiency and accuracy of CFI [1]–[14].

Unfortunately, despite considerable effort in CFI, its efficacy is still questionable. Many studies have shown loop holes in CFI through which an attacker can exploit and bypass the CFI [15]–[21]. In particular, because pioneering CFI works protect indirect branches based on over-approximated valid branch targets derived from a static pointer analysis, the attackers can achieve the intended task without violating CFI mitigations.

Hence, state-of-the-art CFI research focuses on minimizing the number of valid targets on each indirect branch by employing runtime information to refine statically generated CFG. For example, PathArmor [11] and PittyPat [8] reduce candidate targets of branches based on the program context and the path to the indirect branch. They utilize the last branch record (LBR) and the process trace (PT) to retrieve the trace, respectively. These approaches allow the classification of valid target addresses of a certain indirect branch depending on the program path; thus, they can reduce the number of targets compared to a context-insensitive approach. However,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access

even with these approaches, the number of valid targets on a certain branch remains high, which gives the attacker an opportunity to bypass the CFI. To improve accuracy in the context of target reduction, OS-CFI [10] and μCFI [9] leverage different contextual information, the location of code pointer updates, and the constraining data that determines the conditional branch direction, separately.

Although previous works have contributed to improving the effectiveness of CFI, the assumption underlying these works is that the protection of data directly related to control flows is sufficient to mitigate a control flow hijacking attack. For example, μCFI shows that attackers can bypass CFI solutions that impose tight constraint on only control data by exploiting non-control data such as the index of array containing function pointers. Thus, μCFI identifies the non-control data that directly determines control data; it traces the value of the non-control data at runtime to narrow down a set of allowed branch targets to a unique value. However, μCFI assumes an impractical threat model wherein attackers cannot corrupt non-control data, and this indirectly affects control flows.

Another limitation of previous works is that they depend on architecture-specific hardware features such as LBR and PT, which are manufactured by Intel. Although these hardware features enable CFI solutions to obtain indispensable runtime information for the strict refinement of CFG without incurring significant performance overhead, they can result in the following drawbacks. First, the dedicated design for the specific hardware hinders the deployment of solutions to systems that do not support the hardware features. Second, the proposed measure inherits the limitation of such hardware features. For instance, the number of branches that LBR can track is limited to 16, and thus, path information derived from LBR is limited as well. In addition, it has been known that the program trace tracked by PT can be lost based on the amount of monitored information [10]. The fundamental flaws of hardware features can cause security holes to be exploited.

To improve the accuracy of the branch target reduction, we propose vCFI. The key idea of vCFI is protecting every operation associated with generating indirect and conditional branches related to the indirect branches. Toward this end, we analyze the data of the program and extract targets of protection that determine the direction of branches. We retrieve instructions that are responsible for generating or updating the conditional value and indirect branch address. Then, the instructions are patched such that all intermediate operands for the computation of the conditional and indirect branch values are stored in the shadow stack of the location hidden from the attacker. This ensures that all values that affect control flow are isolated (protected); thus, the benign control flow is preserved during program runtime. Compared to previous work, vCFI assumes a more strict attack model. That is, the previous research exempts the existence of non-control data attack that can diverge the control flow, such as manipulating the flag in the conditional statement. In contrast, vCFI assumes a non-control data attack and proposes a facility to protect important data that can be exploited to divert control flow. Further, we use vCFI as a compiler extension so that dependency to a specific hardware feature can be minimized.

The contributions of this paper are as follows:

1. Preventing control data attacks and non-control data attacks related to control flow: We proposed vCFI as a more comprehensive and realistic method that eliminates the limitation of conventional CFI approaches by ensuring the integrity of control-related data and condition-related data.

2. Providing a generally applicable method: We designed and implemented a prototype of vCFI for instrumenting control/condition-related data. vCFI extracts data to be protected through static analysis and enforces them in runtime. We implemented it such that it is not dependent on hardware and is thereby more scalable.

3. Empirical evaluation on control flow-related data: We evaluated the system using the common SPEC CPU2006 benchmark. Furthermore, through statistics and analysis on the control flow-related data, we demonstrated the importance of control data and non-control data to affect control flow. In addition, we proposed an applicable and more efficient method based on performance analysis. The results showed that this system performed more accurately and better compared to other conventional CFI techniques.

The remainder of this paper is organized as follows. Sections II and III describe the motivation behind the study and provide the preliminary with code example. In section IV, we describe the design of vCFI. In section V, we evaluate our approach. In section VI, we describe related work. In section VII, we discuss the limitations of our work. In section VIII, we conclude the paper.

II. MOTIVATION

State-of-the-art CFIs are still vulnerable because of the following reasons.

A. LIMITATIONS OF POINTER ANALYSIS

The effectiveness of CFI depends on the accuracy of CFG. Unfortunately, because the pointer value—the target of indirect branches (e.g., indirect jump, call, and return)—is dynamically determined during runtime, it is considerably difficult to conduct sound and complete pointer analysis [22]. The difficulty of conducting an accurate pointer analysis results in an overapproximated CFG that forces CFI techniques to create an opportunity for attackers to bypass the security perimeter. Previous works [5], [6], [13]have used heuristic approaches to constrain indirect branch targets and improve the accuracy of CFG.

B. IMPRACTICAL ASSUMPTIONS WITH NON-CONTROL DATA

Most CFI approaches are designed with a relaxed attack model that exempts the possibility of compromising non-control data that affects control flow (e.g., a conditional flag). Therefore, existing CFI techniques have only focused on

```
1  typedef int (*FUNC)();
2  FUNC func_ptr[4] = {&add, &sub, &empty, &exec};
3  int calc_ptr(int authencate, int idx, char *c) {
4    int auth = authencate;
5    int sum = 0 ;
6    FUNC func = &empty;
7    while(idx) {
8      if ( auth == UNPRIV ) {
9        if (idx < 3)
10         func = func_ptr[idx];
11     }
12     else {
13       func = func_ptr[idx];
14     }
15     strcpy(buf,c);     // buffer overflow
16     sum += (*func)(); // indirect call
17     idx = idx-1;
18   }
19   return sum;
20 }
21
22 int main(int argc, const char *argv[]) {
23   int sum ;
24   char buf[100] = {0,};
25   int arg = atoi(argv[1]);
26   int idx = atoi(argv[2]);
27   FUNC func = NULL;
28
29   sum = calc_ptr(arg, idx, buf) ;
30
31 }
```

FIGURE 1: An example of a code snippet vulnerable to a control-flow attack

protecting control data such as the function pointer. However, this is an impractical assumption because the attacker is free to manipulate either control data or non-control data based on cost efficiency in terms of a successful attack. Therefore, a reasonable attack model should not limit the capability of the attacker to altering only the control data. In other words, the possibility of compromising non-control data as well as control data should be considered from the perspective of the defender.

III. PRELIMINARY

Before in-depth discussion on the motivating example, we categorize data into four types: control data, control dependency data, condition data, and condition dependency data; we also define each data type to avoid ambiguity. Control data is data directly referred to by indirect branch instructions (i.e., indirect jmp/call and return) as an operand. This data includes function pointers in vtable and GOT, code pointers in jump table, return address in stack, etc. Control dependency data is data that affects control data. Furthermore, data used to determine control dependency data is also control dependency data.

Code example: The indirect branch includes an indirect call, an indirect jump, and return instructions, and its target-address is determined during the process runtime. Owing to this nature, CFG is generated with approximation on the candidates of branch targets. Therefore, the CFI techniques inevitably check the validity of an indirect branch based on the set of branch target candidates derived by static analysis.

Unfortunately, validation with the set is not enough to defeat a control flow hijacking attack. As shown in a previous study [15], the attacker can achieve turing-complete computation by changing the original indirect branch target to one of the candidates in the set. There are several ways to manipulate the indirect branch. First, the attacker can directly modify the control data such as a function pointer value. Second, the control dependency data that attributes the generation of control data can be abused. Third, the condition data such as a flag in a conditional statement can be manipulated because it can resolve control data values. Finally, condition dependency data that affects condition data can be manipulated as well.

We present an extreme example in Figure 1 to illustrate various attack points that can be abused to bypass CFI. In the calc_ptr function, the exec can be executed when the value of auth is "PRIV" and idx is 3. If we assume that the attacker's goal is launching the exec function, the func value-the control data-can be directly modified to the address of exec. In contrast, non-control data that affects control data can be corrupted as well. For example, the attacker can change the value of auth and idx to "PRIV" and "3," respectively; hence, the control flow veers to line 13 and func is set to the address of exec owing to the modified idx.

Existing approaches assumed that non-control data—idx and auth in this example—are not manipulated. For instance, μCFI can particularly check the validity of func (in line 16) by tracking the value of idx. Therefore, provided that idx is modified by the attacker, μCFI including other existing approaches cannot detect control flow bending. We argue that such an assumption is not practical, and we attempt to harden the CFI with an even stronger attack model that assumes the attacker can manipulate non-control data to deviate control flow. In Section IV, we illustrate the design of vCFI to show how it tackles such stronger attacks and reinforces CFI.

IV. DESIGN

A. OVERVIEW

In this section, we describe the design of value-based constraint CFI (vCFI). The designed vCFI aims to enhance CFI by effectively reducing the target set of the indirect branches. Although previous work proposed novel approaches [9], [10] to achieve the same goal, we consider a more strict and practical attack model that allows the attacker to manipulate both non-control and control data. To tackle this malicious attack, vCFI attempts to protect every data that can be abused to diverge control flow. Therefore, extracting a proper set for control flow-related data forms the core of vCFI. Toward this end, we perform static analysis to create a general CFG based on basic block granularity. Besides, we conduct dependency analysis for retrieving the relationship between instructions and constraint analysis for augmenting the accuracy of pointer analysis. Finally, control data, control

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access

Algorithm 1 Dependency analysis

Input: instructions - all instructions of program
Output: dependency map
$DM \leftarrow 0$ //dependency map

repeat
  for $inst \leftarrow instructions$ **do**
    if isAllocInst(inst) **then**
      $DM \leftarrow DM \cup (sink, dst)$
    else if isStoreInst(inst) **then**
      $DM \leftarrow DM \cup (valueOpnd, ptrOpnd)$
    else if isCallInst(inst) **then**
      $func \leftarrow calledFunc(callinst)$
      for $arg \leftarrow args(func), opnd \leftarrow opnds$ **do**
        $DM \leftarrow DM \cup (opnd, arg)$
      end for
      for $ret \leftarrow returns(func)$ **do**
        $DM \leftarrow DM \cup (ret, dst)$
      end for
    else
      for $opnd \leftarrow opnds$ **do**
        $DM \leftarrow DM \cup (opnd, dst)$
      end for
    end if
  end for
until no instruction is found

dependency data, condition data, and condition dependency data are retrieved by data analysis. We instrument the program such that these data are protected in a shadow memory; thus, the integrity of control flow is preserved at runtime.

B. STATIC ANALYSIS

Our static analysis consists of branch, dependency, condition, and data analysis. Through the analysis, we derive necessary information such as control and dependency data, which play critical roles in building vCFI. In the following subsections, we describe the mechanism and goal of each analysis.

1) BRANCH ANALYSIS

The CFG describes the path between the basic block that is a linear sequence of code. Because each basic block ends with branch instructions (e.g., jump), we first conduct branch analysis to draw a simple CFG. Then, we further conduct type and pointer analysis to obtain more constraint for CFG. For example, we can specify the candidate callee functions by comparing the type and argument of callees and those set in the caller.

Static value flow (SVF) analysis was used for pointer analysis. SVF analysis enables flow-, context-, heap-, and field-sensitive analysis. Specifically, pointer analysis and value-flow analysis in SVF are conducted for only top-level pointers and address-taken variables. We used information analyzed in SVF to place more constraints on ambiguous pointers.

2) DEPENDENCY ANALYSIS

vCFI preserves control flow by protecting every data associated with the calculation of conditional and control data. To realize this, we thoroughly protect all data found in the static analysis. However, this naive approach not only significantly increases the program size, but also incurs a considerable performance overhead. Therefore, we perform dependency analysis to explicitly extract operations that indifferently affect an indirect branch and the conditional branches on the pass to that indirect branch. The Algorithm 1 shows pseudocodes for dependency analysis.

In LLVM, intermediate representation (IR) is the language internally used by a compiler to represent the source code. By using IR, the compiler can analyze and optimize the program. Similarly, for a general programming language, each line of IR defines variables or operations. In particular, IR supports various IR instructions, the operations for which range from arithmetic, terminator (e.g., ret instruction), memory management, and conversion operations. Each instruction can have corresponding operands. The operand can be another instruction as well as a constant value. Therefore, we can track the propagation of values by creating dependency mappings between the identical operand of different instructions and by traversing them. For example, if we have a statement with a binary operation "a = b + c," the dependency relation—(source, destination)—of this statement can be defined as (b,a) and (c,a). Then, by backtracking the mapping from a certain point of instruction (destination), we can find every instruction (source) that affects the operation of the destination instruction.

Dependency information is significantly leveraged when we perform data analysis to derive important data (e.g., condition data and its dependency data) that need to be protected.

3) DEPENDENCY CHECK

Algorithm 2 shows pseudocodes for the dependency check procedure. We create a dependency map between the source and the destination operands for every instruction across all functions. We specifically illustrate the dependency check for the call instruction (callinst in LLVM IR). For callinst, we need to check the dependency between the caller and the callee.

In the callee, we first check the dependency data for the return instruction by traversing the CFG backward from all return instructions to the entry of the called function. Then, we pair the callee's formal return and caller's actual return (e.g., (callee_rtn, caller_rtn)). The dependency of the passed arguments is checked as well. In this case, we perform forward traversing. If one of the callee's argument types is pointer, we check the dependency of that argument. This is because the value of a pointer-typed argument can be changed in the callee. Similar to the return, a pair of the callee's formal argument and the caller's actual argument is created (e.g., (callee_arg, caller_arg)).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access

Algorithm 2 Dependency check

Input: target(src,dst), DM(dependency map)
Output: dependency data set
$DS \leftarrow target(src, dst)$ //dependency data set

repeat
  for $target(src, dst) \leftarrow DS$ **do**
    for $dep(src, dst) \leftarrow DM$ **do**
      if $src(target) = dst(dep)$ **then**
        $DS \leftarrow DS \cup dep(src, dst)$
      end if
    end for
  end for
until no new dependency data set is found

4) BACKWARD TRACING

Once we obtain the dependency map between the operands, we perform backward traversing on it to collect all source operands for the individual operand. As a result, sets of dependency data for every operand can be finally retrieved from this phase.

Src operands are tracked using backward traversing based on the dependency map for the instruction operands. This process finds all src operands with operands on the dependency map as dst. The process of finding src operands having the found operands as dst is repeated until there none are left. Finally, a set of all dependency data related with the operands of the target instruction to be analyzed can be obtained.

5) CONDITION ANALYSIS

vCFI needs to find the sets of all condition data that are on the path between the current basic block with an indirect branch and another basic block with a preceding indirect branch. To this end, we conduct condition analysis to extract the condition data that lead control flow to get to the current basic block.

Algorithm 3 to find the conditional data is straightforward. We continue to recursively traverse the previous basic blocks of current basic block until the analysis reaches the entry of the program. Thus, every basic block can have a set of previous blocks. Note that the set has previous information that can be retrieved from the path between the current block and the entry of the program. Therefore, there could be redundancy in the previous block information for different basic blocks on the same path. We further discuss the removal of this redundancy in Section IV-D.

6) DATA ANALYSIS

We find all data that can affect the indirect branch by conducting data analysis. These data include control data, control dependency data, condition data, and condition dependency data. Recall that a callee is determined by the control data. Moreover, the control flow to the instruction that determines the control data is constructed with conditional branches. Based on the results from the previous analysis, the CFG,

Algorithm 3 Condition analysis

Input: targetbb(target basic block)
Output: condition data set
$CS \leftarrow 0$ //condition data set
$curbb \leftarrow targetbb$

repeat
  $BBs \leftarrow$ previous(curbb)
  for $bb \leftarrow BBs$ **do**
    $CS \leftarrow CS \cup condition(bb)$
    $curbb < -bb$
  end for
until no new bb is found

previous(bb)
$func \leftarrow getfunc(bb)$
if bb is func entryblock **then**
  $BBs \leftarrow callsite(func)$
else
  $BBs \leftarrow predecessor(bb)$
end if

dependency map, and the set of condition data, we retrieve all control flow-related data. The data analysis is fulfilled as follows: 1) find the control data of current indirect branch; 2) find all dependency data and track instructions backward; 3) find all dependency instructions and retrieve the control dependency data; 4) find all basic blocks that contain control data and control dependency data; 5) find all previous blocks and retrieve condition data; 6) find all condition dependency data; 7) find all basic blocks that involve the condition-related data (condition data and condition dependency data); and 8) Repeat steps 5 to 7 until no basic block is found.

7) EXAMPLE

We illustrate the procedure for retrieving the control flow-related data with LLVM IR in Figure 2. We first generate the CFG by branch analysis and the dependency map by dependency analysis. As an example of dependency map, line 3 (sink, %4), line 8 (%0, %4), line 7 (sink, %8), and line 12 (@empty, %8) can be created. After creating the CFG and dependency map, we analyze the conditional branches. Then, we start the data analysis with initially deriving the control data, which is a pointer (%32) in line 53. We leverage the dependency map to find the source of %32, which in our example, is line 52 (%8, %32). In turn, we obtain the source of %8 such as line 46 (%28, %8), line 35 (%22, %8), line 12 (@empty, %8), and line 7: (sink, %8). We repeat this until all control dependency data are found.

Afterwards, we find all basic blocks that encompass the control and control dependency data from the previous step. For instance, label 29 is the basic black that contains control data %32 (line 52). Line 46, line 35, line 12, and line 7 are defined in basic blocks, the labels of which are 24, 18, and the entry, respectively.

Then, the path to the found blocks is derived. In other

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access


```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @calc_ptr(i32, i32, i8*) #0 {
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i8*, align 8
  %7 = alloca i32, align 4
  %8 = alloca i32 (...)*, align 8
  store i32 %0, i32* %4, align 4
  store i32 %1, i32* %5, align 4
  store i8* %2, i8** %6, align 8
  store i32 0, i32* %7, align 4
  store i32 (...)* bitcast (i32 ()* @empty to i32 (...)*), i32 (...)** %8, align 8
  br label %9

; <label>:9:                                ; preds = %29, %3
  %10 = load i32, i32* %5, align 4
  %11 = icmp ne i32 %10, 0
  br i1 %11, label %12, label %38

; <label>:12:                               ; preds = %9
  %13 = load i32, i32* %4, align 4
  %14 = icmp eq i32 %13, 0
  br i1 %14, label %15, label %24

; <label>:15:                               ; preds = %12
  %16 = load i32, i32* %5, align 4
  %17 = icmp slt i32 %16, 3
  br i1 %17, label %18, label %23

; <label>:18:                               ; preds = %15
  %19 = load i32, i32* %5, align 4
  %20 = sext i32 %19 to i64
  %21 = getelementptr inbounds [4 x i32 (...)*], [4 x i32 (...)*]* @func_ptr, i64 0, i64 %20
  %22 = load i32 (...)*, i32 (...)** %21, align 8
  store i32 (...)* %22, i32 (...)** %8, align 8
  br label %23

; <label>:23:                               ; preds = %18, %15
  br label %29

; <label>:24:                               ; preds = %12
  %25 = load i32, i32* %5, align 4
  %26 = sext i32 %25 to i64
  %27 = getelementptr inbounds [4 x i32 (...)*], [4 x i32 (...)*]* @func_ptr, i64 0, i64 %26
  %28 = load i32 (...)*, i32 (...)** %27, align 8
  store i32 (...)* %28, i32 (...)** %8, align 8
  br label %29

; <label>:29:                               ; preds = %24, %23
  %30 = load i8*, i8** %6, align 8
  %31 = call i8* @strcpy(i8* getelementptr inbounds ([100 x i8], [100 x i8]* @buf, i32 0, i32 0), i8* %30) #5
  %32 = load i32 (...)*, i32 (...)** %8, align 8
  %33 = call i32 (...) %32()
  %34 = load i32, i32* %7, align 4
  %35 = add nsw i32 %34, %33
  store i32 %35, i32* %7, align 4
  %36 = load i32, i32* %5, align 4
  %37 = add nsw i32 %36, -1
  store i32 %37, i32* %5, align 4
  br label %9

; <label>:38:                               ; preds = %9
  %39 = load i32, i32* %7, align 4
  ret i32 %39
}
```

FIGURE 2: LLVM IR code example

words, the blocks with labels 23 and 24 for the block with label 29, the block with label 12 for the block with label 24, the block with label 15 for the block with label 18, and the previous block for the callsite of @calc_ptr. Therefore, we obtain the set of basic blocks with the entry, label 9, label 12, label 15, label 18, label 23, and the label 24. Next, we extract the condition operand of terminator instruction in each basic block. For example, %11 in label 9, %14 in label 12, and %17 in label 15 are the extracted condition data. The basic blocks other than those for the extraction are terminated with an unconditional branch so no condition data are available. The dependency analysis is performed against these condition data to derive the condition dependency data. This procedure is repeated until no additional basic block is found.

C. RUNTIME ENFORCEMENT

The data found in the static analysis stage are all the data that can be misused to manipulate the control flow when indirect control transfer occurs. During execution, these data should be protected from modification attacks based on memory corruption. Therefore, we ensure control flow integrity by checking whether the corresponding data have been modified during execution. To ensure the integrity of corresponding data, the instructions using the corresponding data as operands are instrumented, whereby the values indicating the corresponding data of destination operands are stored in a shadow memory, and the values indicating the corresponding data of source operands are read from the shadow memory. In this case, the data integrity can be ensured because data values prior to modification are maintained in the shadow memory even if data are modified by memory corruption. The values saved in the shadow memory are referenced when the data values are explicitly used by instructions.

```

setValue
Idx = Hash(&data);
Secure[Idx].add(&data, data);

getValue
Idx = Hash(&data);
Secure_data = Secure[Idx].find(&data,data);
```

FIGURE 3: Implementation code for runtime enforcement

The destination operands of each instruction should write data values on the shadow memory, and the source operands should add instructions to read the data values from the shadow memory. Therefore, codes were added for this. The selected method keeps a value of data in a secure region when it is written on the memory and reads a value of data from the secure region when read from memory. Figure 3 shows pseudocodes of setValue for executing a write operation and getValue for executing a read operation. setValue generates Idx with the address value of data and afterwards, saves the data in a safe region. On the other hand, getValue generates

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access

	Control	Control dependency	Condition	Condition Dependency	Control dependency (excluded)	Condition dependency (excluded)	Total	Protected data* (%)
bzip2	20	1109	736	3453	1036	3447	4226	16.96031
Hmmer	9	932	1693	7611	1004	9206	9440	20.26229
perlbench	137	2405	20880	50285	10296	74613	71305	18.82451
gcc	205	1134	48807	38831	20907	157601	86657	15.98301
gobmk	44	201	9907	23080	1233	49984	32987	11.72408
soplex	513	2324	2240	8606	6245	17818	11300	13.18384
povray	173	1486	7934	27173	8429	44290	35194	14.02554
Milc	4	5	333	1230	7	1558	1559	5.603882
omnetpp	709	4897	3349	11045	12239	22529	16142	12.25469
Total	1814	14493	95879	171314	61396	381046	268810	
Average	201.5555556	1610.3333	10653.222	19034.88889	6821.777778	42338.44444	29867.78	14.313573

TABLE 1: Full condition analysis
In the sixth and seventh columns, "excluded" refers to data that are excluded because they are not used in the memory operations.
* protected data = (total / overall data in the program)

Idx with the address value of data as well and afterwards, finds the data from the safe region and returns it. Then, computation is performed using the safe data obtained. setValue and getValue are added to the respective operations of one instruction. setValue is added at a part where the data to be protected are used as destination operands of the instruction, and getValue is added at a part where the data to be protected are used as source operands of the instruction.

D. OPTIMIZATION

Considering that an attacker abuses memory corruption, attack may occur when data are in the memory. In other words, the attacker has no way of directly modifying a corresponding operand when the operand of instruction uses a register, and not a memory reference. Therefore, the data that need to be protected are limited to the operands of memory read or write instructions. Therefore, only the data related with memory operations were verified rather than checking every operation related to the data. Furthermore, the constant values existing in the code region were excluded from the data to be protected because the code region could not be modified directly because of data execution prevention (DEP). The performance overhead is minimized by reducing the number of targets to be monitored through this method.

Analysis is performed in the basic block unit in each process of the static analysis to improve its efficiency. In particular, when a certain block already has an analyzed result, the existing analysis result is used for the corresponding basic block to avoid redundant analysis.

V. EVALUATION

We conducted the test using Spec CPU2006 Benchmark. The test environment consisted of Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-72-generic x86_64), Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, and 128 GB memory. The analysis was performed at the IR level of LLVM (version 7.0.0).

Statistical analysis results of the test are provided for all data affecting the control flow of indirect calls. Based on the analysis of these data, we investigate whether this method is effective in terms of security and performance.

A. FULL CONDITION ANALYSIS

With respect to Table 1, analysis was performed for the respective amount of control-related data (i.e., control data and control dependency data) and condition-related data (i.e., condition data and condition dependency data) for all indirect calls in the benchmark program. These data are all data that can directly or indirectly change the target address of the indirect branch. The total column indicates the amount of all the control flow-related data affecting each indirect branch (duplicates are excluded). As shown in the Table 1, the part occupied by the control data are only 0.67% of total control flow-related data, which is very insignificant. Furthermore, the control-related data including control data and control dependency data account for only 6% of total control flow-related data. Thus, the attack surface for control dependency data that can indirectly change the pointer value is much larger than that for directly changing the control data to modify a pointer value. Because all methods proposed in the existing papers checked only control data, the possibility of bypassing these methods is high. The excluded data refers to the quantity of data unrelated to the memory operation among the extracted data. A large amount of data to be protected has been practically reduced in the program. This is because vCFI extracts only the data that need to be protected through static analysis. The proportion of data to be protected among all instruction-related data after excluding the data unrelated to memory operation is 14.3%. When analyzed statistically, it can be said that 14.3% of control and condition-related data are processed in runtime.

B. ONE-TIME CONDITION ANALYSIS

Protecting all condition-related data requires considerable analysis time and system resource although it can completely protect all control flows on paths reached to indirect calls. Furthermore, when considered in the perspective of the attacker, as the distance of attack target (i.e., indirect branch) from a location where memory corruption caused by

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access

	Control	Control dependency	Condition	Condition Dependency	Control dependency (excluded)	Condition dependency (excluded)	Total	Protected data* (%)
bzip2	20	1109	226	2121	1036	2059	2387	9.579805
Hmmer	9	932	242	3946	1004	4382	4255	9.133057
perlbench	137	2405	6526	22735	10296	37078	29541	7.798822
gcc	205	1134	11267	11759	20157	76143	22950	4.232896
gobmk	44	201	400	2699	1233	5383	3156	1.121691
soplex	513	2324	1080	5101	6245	12489	6699	7.8158
povray	173	1486	3513	14647	8429	27896	18356	7.315246
Milc	4	5	4	15	7	22	24	0.086269
omnetpp	709	4897	2579	9397	12199	19627	13736	10.4281
Total	1814	14493	25837	72420	60606	185079	101104	
Average	201.5555556	1610.3333	2870.7778	8046.666667	6734	20564.33333	11233.78	6.39018737

TABLE 2: One-time condition analysis
In the sixth and seventh columns, "excluded" refers to data that are excluded because they are not used in the memory operations.
* protected data = (total / overall data in the program)

vulnerability occurs, it becomes increasingly difficult for the attacker to modify the data as desired because of the side effects caused by instructions in the middle; moreover, the difficulty (cost) of attack increases because more detailed attack configurations are required. Based on this fact, there seems to be a room for applying a more practical method than the full condition analysis method. Therefore, we also tested a one-time condition analysis method that improved performance by selectively analyzing the condition data and condition dependency data that could be misused in actual attacks.

When condition-related data are extracted through condition analysis in the full condition analysis method, the condition data and condition dependency data are extracted by using control-related data as input values. Then, the condition data and condition dependency data are selected again through the condition analysis. This condition analysis process is performed recursively, thereby extracting all condition-related data.

In contrast, the one-time condition analysis method performs a condition analysis only once. In other words, this method performs the analysis for the condition-related data of basic blocks directly branching to the basic blocks containing the control-related data. Therefore, the extracted control data and control dependency data are identical to the results of full condition analysis. However, for the condition-related data, the analysis results are produced for only the condition data of the basic block that can directly change the control data and the control data dependency. Although this method does not provide perfect integrity for indirect calls, it can protect data that are relatively easy to modify when an attack occurs; thus, improvements such as more efficient monitoring and faster performance can be expected.

As shown in Table 2, control data and control dependency data derived from the one-time condition analysis were identical to the results of full condition analysis; however, condition-related data derived as protection targets based on a one-time condition analysis were reduced to 26.9% and 42.3%, respectively, compared to the results of Table 1. Furthermore, the total amount of data to be monitored was reduced to 37.6%.

However, although monitored amount of data has decreased, it does not mean that the strength of security provided by vCFI decreased considerably. In comparison with conventional CFI studies, vCFI ensures same or higher security strength compared to conventional techniques of protecting only control data because vCFI protects all control-related data including control data even if a one-time condition analysis is applied. When a one-time condition analysis is applied, some part of the condition-related data, which are protected in the full condition analysis, are excluded from the protection targets; however, no serious security problem occurs because it is relatively difficult to use the excluded data in attacks.

We consider the cost and complexity of protection according to the frequency of performing the condition analysis for comparison with a full condition analysis-based method. In this process, control dependency data, which are data that can directly/indirectly change the control data, are all extracted, and these data become the analysis targets of the condition analysis.

Let us assume a trace reaching a target basic block containing control data from the entry basic block of CFG. Here, the control data exist in the target basic block and the control dependency data exist at a location close to the target basic block. Suppose the condition analysis is performed based on these control-related data. Then, adjacent basic blocks will be extracted starting from the previous basic block of the basic block containing the control-related data. Let x be the distance between the target basic block and the furthest basic block from the target basic block among the basic blocks containing the control-related data. Then, when the condition analysis is performed only once, the previous basic blocks of the control-related basic blocks will be extracted, and the distance will be at least x+1. If the condition analysis is performed once more, the distance will become x+2. If this is repeated continuously, it ultimately becomes the full condition analysis, and the distance becomes the distance from the target basic block to the entry basic block.

As the condition analysis is repeated, condition-related

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access

	Source size (bytes)			Analysis time (s)		
	Original	One-time	Full	Original	One-time	Full
bzip2	141460	311908	443292	0.84	7.48	32
Hmmer	309332	639776	1045952	1.82	33.98	326
perlbench	2540524	5148340	8617780	1333.88	5685	22118
gcc	5309816	8000236	13558296	700.58	6929	33951
gobmk	4592036	5402728	7624832	150.93	1561	31077
soplex	732544	1233416	1567364	11.39	1478	3917
povray	1888900	3570312	4897752	210.59	1960	4191
Milc	197952	216564	321752	0.71	0.69	1
omnetpp	1378472	2579024	2777996	66.4	4474	6938
Total	17091036	27102304	40855016	2477.14	22129.15	102551
Average	1899004	3011367.1	4539446.2	275.2377778	2458.794444	11394.55556

TABLE 3: Analysis performance

data located further away from the indirect call are protected. Furthermore, if the relative distance from the indirect call increases, various operations can change (affect) the values of data. Hence, the possibility of a successful attack decreases as it is difficult to compose the attack codes. In addition, the possibility of becoming an attack target drops as well because a large cost is required for an attack.

In general, it can be said that as the distance to the indirect call decreases, the importance increases compared to that of data relatively farther away. Consequently, a one-time condition analysis is a method of protecting control/condition-related data near a relatively more important indirect call. Moreover, it also has the advantage that less performance overhead can be expected compared to a full condition analysis.

C. ANALYSIS PERFORMANCE

The source size change and analysis time were evaluated according to the adoption of vCFI. In the case of source codes, the bitcode of LLVM IR was targeted. In general, the increase in the source code size was proportional to the amount of control-related data and condition-related data that are to be protected among all the data of the program. When the code size is compared with the average value of data to be protected among the entire data shown in the last column of Table 1 and 2, it can be seen that they are proportional to each other.

The results indicate that static analysis time increases exponentially as the size of analysis target program increases. This is because the analysis is performed based on the complex CFG of the program. The CFG of the program has a tree structure, and as the depth increases, the number of child basic blocks increases exponentially. In addition, the program analysis time is affected by the complexity of the program graph.

D. RUNTIME PERFORMANCE

This test measured the amount of overhead that occurs in actual runtime. We performed the benchmark test repeatedly for the original program, one-time condition analysis-applied program, and full condition analysis-applied program, respectively, and then, we obtained the average time for the execution. The factor that had the largest influence on the runtime performance was the depth of the indirect call in the control flow of the program. Because the data from the beginning of the execution trace to the indirect call must be protected, the amount of data to be protected is determined by the location of the indirect call, which considerably affects performance.

In the case of a full condition analysis, the average overhead was 13.7%. In contrast, it was 6.7% in the case of a one-time condition analysis, thereby showing a considerably better performance than the full condition analysis. In the case of the program protected based on the one-time condition analysis, a much smaller performance degradation was observed because not every data existing on the execution flow path from the entry to the indirect call was protected.

E. MEMORY PERFORMANCE

This test measured the memory usage of each process in actual runtime. The measured amount represents the actual memory size of the executed process. The increased amount compared to the original memory size is the result of storing additional runtime enforcement code and control-related data in shadow memory. In the case of one-time condition analysis, the average overhead was 1.7%. On the other hand, it was 4.3% in the case of full condition analysis.

F. SECURITY ANALYSIS

vCFI finds and protects all dependency data affecting the control flow of indirect control transfer. These data include all control-related data and condition-related data, and they are protected from modification. In a full condition analysis, no attack can occur on the control flow of an indirect branch because every data that can modify the indirect branch is protected. Therefore, the allowed target of the indirect branch must always be 1.

In a one-time condition analysis, control-related data are protected; however, not all condition-related data are protected. In this case, the control flow of the indirect branch

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.2980026, IEEE Access

	Original (s)	One-time (s)	Full (s)	One-time overhead (%)	Full overhead (%)
bzip2	8.187	8.986	10.97	9.759374618	33.9929156
Hmmer	4.978	5.919	5.938	18.90317397	19.28485335
perlbench	0.1412	0.1481	0.1679	4.886685552	18.90934844
gcc	1.317	1.331	1.377	1.06302202	4.555808656
gobmk	17.06	17.31	19.03	1.465416178	11.54747948
soplex	0.02045	0.02228	0.02287	8.948655257	11.83374083
povray	0.001968	0.002013	0.002086	2.286585366	5.995934959
Milc	23.01	23.34	23.51	1.434159061	2.172968275
omnetpp	0.5244	0.5829	0.6021	11.15560641	14.81693364
Total	55.240018	57.641293	61.619956	59.90267843	123.1099832
Average	6.137779778	6.4045881	6.8466618	6.655853158	13.67888703

TABLE 4: Runtime performance

	Original (KB)	One-time (KB)	Full (KB)	One-time overhead (%)	Full overhead (%)
bzip2	20128	20264	21024	0.675675676	4.451510334
Hmmer	8036	8336	8396	3.733200597	4.479840717
perlbench	7676	7976	8416	3.908285565	9.640437728
gcc	18984	19404	19980	2.212389381	5.246523388
gobmk	26368	26584	26728	0.819174757	1.365291262
soplex	4844	4860	5064	0.330305533	4.541701073
povray	2636	2712	2788	2.883156297	5.766312595
Milc	10068	10100	10228	0.317838697	1.589193484
omnetpp	11628	11696	11832	0.584795322	1.754385965
Total	110368	111932	114456	15.46482182	38.83519655
Average	12263.11111	12436.8889	12717.333	1.718313536	4.315021838

TABLE 5: Memory performance

can be modified by manipulating the condition-related data. In this case, the number of allowed targets is not always 1. However, only because the allowed target of the one-time condition analysis does not always become 1, it does not mean that the security strength is lower than that of the existing approaches. This is because the assumption itself is different from that of the existing one. We assume that every data that can change the control flow can be manipulated whereas other studies considered control data only attacks. Therefore, when a comparison is performed under the same assumption of control data only attacks as in existing studies, the one-time analysis method shows that the number of allowed targets is always 1 as well because all control-related data are protected from modification. On the other hand, under the assumption that a non-control data attack is possible, it cannot be said that the number of allowed targets is always 1 for a one-time condition analysis as well as for other approaches.

VI. RELATED WORK

Control-flow hijacking attack including code-reuse attack is one of the prevalent methods to lead a vulnerable software to deviate from its original execution path. Furthermore, the widespread adoption of W⊕X (Write XOR Execute) primitive prohibits attackers from code injection attacks, which makes the attack technique an indispensable part of the attack process. With the prevalence of this attack, CFI [1] was proposed as a mitigation technique. As CFI enforces original control-flow based on a control flow graph (CFG) derived by a static analysis (e.g., point-to analysis), a sound and complete static analysis is required to avoid disrupting the functionality of the software and introducing security holes. However, a sound and complete point-to analysis suffers from the undecidability of aliasing [22].

As unsound point-to analysis can disturb the original functionality of the program, early CFI works—coarse-grained CFIs [2], [3]—employed sound but incomplete point-to analysis to generate CFG. The goal of coarse-grained CFI works focuses on reducing performance overhead to practically enforce CFI policies to programs. Thus, they relax the allowed target on indirect branches. For example, BinCFI [3] allows indirect calls to transfer to any function entries of the program. Unfortunately, it has been shown that attackers still construct payloads under the relaxed CFI restriction [12].

Fine-grained CFI is considered as a relatively secure CFI policy compared to coarse-grained CFI. However, attackers can bypass fine-grained CFI by exploiting over-approximated edges of CFG [15]–[18]. To address the problem of fine-grained CFI, researchers have strived for enhancing the accuracy of CFG. MCFI [4] enables a fine-grained CFI policy to be adopted to each module, which makes dynamic-link libraries (DLLs) to be efficiently enforced by CFI. Forward-

edge CFI [5] restricts the forward control transfers (i.e., indirect jmp/call and virtual call) to more strict targets. TypeArmor [6] also enforces strong fine-grained CFI on forward-edges. As compared to forward-edge CFI, TypeArmor can be adopted to a binary when the source code of the application is not available.

Although previous works contribute to mitigate control-flow hijacking attack practically, they inherit the drawback of static analysis. In particular, only one target of each indirect control transfer is determined as the runtime context. This means that a valid target set derived by static analysis, except for one target, can still be exploited to conduct a control-flow hijacking attack. To overcome the limitation of static analysis, mutational CFI, which refines fine-grained CFG derived from static analysis by leveraging runtime information, is used. Generally, extracting runtime information is required for the original programs to be instrumented, which incurs significant performance overhead. Thus, mutational CFI uses hardware features such as branch trace store (BTS), last branch record (LBR), and processor trace (PT), which emit runtime information at the core-level and store them to dedicated registers or buffer.

GRIFFIN [23] and PT-CFI [7] track the context information from Intel PT, and then practically restrict backward-edges to the call site of each function call. Furthermore, recent works [8]–[11] have proposed a way to strictly narrow down the allowed target of forward-edges. PathArmor [11] supports context-sensitive CFI, which checks whether the consecutive path stored in LBR exists within CFG. PITTY-PAT [8], similar to PathArmor, enforces path-sensitive CFI on a forward-edges; however, the target set of each indirect branch is updated by the executed path at runtime while the CFG of PathArmor remains intact. OS-CFI [10] defines the concept of origin, which is the address and context of the last assignment instruction for c-style indirect branch targets and the address of object creation instruction for virtual calls. With the help of the origin stored as meta-data, the target set of each indirect branch can be divided into more fine-grained branches. Although these works can eliminate inaccurate forward-edges, they still approximated edges in exceptional circumstances wherein a function pointer is retrieved from a function pointer array and determined by the index of the array at runtime. μCFI [9] addresses this issue of identifying and tracking non-control data, which directly affects the determination of control-flow.

Despite tremendous improvement on the accuracy of CFI, the state-of-the-art research does not consider non-control data attack which can hijack control-flow by corrupting the data unrelated to control data. In general, since we assume the threat model that a strong attacker can corrupt any data in writable memory space, previous CFI approaches that only assumes control data attack can be ineffective under control-flow hijacking attack in the wild. Furthermore, the dependent design on architecture-specific hardware features can be an obstacle to wide deployment.

VII. DISCUSSION

In the runtime enforcement stage, we allocated a random memory region and saved the data in that region. However, brute force attack can occur for the randomly allocated memory region. For instance, if the address of the random memory region is preserved even with repeated forking of the same instrumented process, which enables the random region to be found without corresponding address change [24], the data stored in the random memory region can be modified. Furthermore, the protected data can be attacked if the random address is found because of the vulnerability of the random algorithm.

Such a drawback of random approaches can be overcome by using hardware-based security techniques similar to those of existing studies. For example, as shown in OS-CFI, a protected memory region can be created by using Intel MPX and TSX, thereby protecting the control-related and condition-related data in the corresponding region.

VIII. CONCLUSION

We proposed vCFI, which can protect all data that can change the control flow related with an indirect branch. Through the vCFI, we extract control-related data that affects the indirect branch target and condition-related data that have influence on the decision of control-related data. Furthermore, all the extracted data, which potentially affect the control flow, are protected in real time by instrumenting the program.

vCFI introduced a more powerful attack model, which assumed that non-control data could be attacked as well and resolved the problem. In that regard, the significance of vCFI is large. Furthermore, the performance measurement has demonstrated that the vCFI provides much stronger security compared to the conventional techniques while maintaining reasonable runtime overhead.

REFERENCES

[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] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in Security and Privacy (SP), 2013 IEEE Symposium on. IEEE, 2013, pp. 559–573.

[3] M. Zhang and R. Sekar, "Control flow integrity for cots binaries." in USENIX Security, 2013, pp. 337–352.

[4] B. Niu and G. Tan, "Modular control-flow integrity," in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, vol. 49, no. 6. ACM, 2014, pp. 577–587.

[5] 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 USENIX Security, vol. 26, 2014, pp. 27–40.

[6] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016, pp. 934–953.

[7] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in Proceedings of the 7th ACM Conference on Data and Application Security and Privacy. Scottsdale, Arizona, USA: ACM, march 2017.

[8] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 131–148.

[9] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2018, pp. 1470–1486.

[10] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in 28th {USENIX} Security Symposium ({USENIX} Security 19), 2019, pp. 195–211.

[11] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 927–940.

[12] 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.

[13] B. Niu and G. Tan, "Per-input control-flow integrity," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 914–926.

[14] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity." in NDSS, vol. 26, 2015, pp. 27–30.

[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] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in 2015 IEEE Symposium on Security and Privacy. IEEE, 2015, pp. 745–762.

[17] 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.

[18] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015, pp. 901–913.

[19] 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.

[20] 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.

[21] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point (er): On the effectiveness of code pointer integrity," in 2015 IEEE Symposium on Security and Privacy. IEEE, 2015, pp. 781–796.

[22] G. Ramalingam, "The undecidability of aliasing," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 16, no. 5, pp. 1467–1471, 1994.

[23] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," ACM SIGPLAN Notices, vol. 52, no. 4, pp. 585–598, 2017.

[24] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in 2014 IEEE Symposium on Security and Privacy. IEEE, 2014, pp. 227–242.

DONGJAE JUNG is currently pursuing a Ph.D. degree from the Graduate School of Information Security in the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, Rep. of Korea. He received his BS degree in Information and Computer Science from Ajou University, Suwon, Rep. of Korea and his MS degree in Information Security from the Korea Advanced Institute of Science and Technology, Daejeon, Rep. of Korea. His current research interests include program analysis, system security, and malware detection and analysis.

MINSU KIM is currently a principal researcher at S2W LAB Inc., South Korea. He received the his M.S. and Ph.D. degrees in Inforamation Security from Korea Advanced Institute of Science and Technology (KAIST), South Korea, and also received the B.S. degree in Computer Science from KAIST. His research interest includes system security, particularly, especially in the mitigations against code reuse attack (CRA).

JINSOO JANG is currently an assistant professor at the Department of Computer Science & Engineering, at Chungnam National University (CNU). Dr. Jang received his Ph.D. and MS in Information Security from Korea Advanced Institute of Science and Technology (KAIST), and BS from Ajou University. He has been working on systems security areas, particularly in hardening the trusted execution environment (TEE) and leveraging general hardware features to build various defensive measures.

BRENT BYUNGHOON KANG (M'09) is currently an associate professor at the Graduate School of Information Security at Korea Advanced Institute of Science and Technology (KAIST). Before KAIST, he has been with George Mason University as an associate professor. Dr. Kang received his PhD in Computer Science from the University of California at Berkeley, , and his MS degree from the University of Maryland at College Park, and his BS degree from Seoul National University. He has been working on in the field of systems security area including botnet defense, OS kernel integrity monitors, trusted execution environment, and hardware assisted security. He is currently a member of the IEEE, the USENIX, and the ACM.