Defending Non-control-data Attacks using Influence Domain Monitoring

Guimin Zhang, Qingbao Li, Zhifeng Chen and Ping Zhang
State Key Laboratory of Mathematical Engineering and Advanced Computing
Zhengzhou, Henan 450001 - China
[e-mail: zh.guimin@163.com, qingbao_li@126.com, xiaohouzi06@163.com, zhpingky@163.com]
*Corresponding author: Guimin Zhang

Received September 17, 2017; revised January 22, 2018; accepted March 1, 2018; published August 31, 2018

Abstract

As an increasing number of defense methods against control-data attacks are deployed in practice, control-data attacks have become challenging, and non-control-data attacks are on the rise. However, defense methods against non-control-data attacks are still deficient even though these attacks can produce damage as significant as that of control-data attacks. We present a method to defend against non-control-data attacks using influence domain monitoring (IDM). A definition of the data influence domain is first proposed to describe the characteristics of a variable during its life cycle. IDM extracts security-critical non-control data from the target program and then instruments the target for monitoring these variables' influence domains to ensure that corrupted variables will not be used as the attackers intend. Therefore, attackers may be able to modify the value of one security-critical variable by exploiting certain memory corruption vulnerabilities, but they will be prevented from using the variable for nefarious purposes. We evaluate a prototype implementation of IDM and use the experimental results to show that this method can defend against most known non-control-data attacks while imposing a moderate amount of performance overhead.

Keywords: Non-control-data attacks, influence domain, security-critical non-control-data set (SCNS), memory corruption vulnerability, instrumentation

This work is supported by the National Social Science Foundation of China (No. 15AJG012), the National Science and Technology Major Project of China (No. 2013JH00103), and the Foundation of Science and Technology on Information Assurance Laboratory (No. KJ-15-107).

http://doi.org/10.3837/tiis.2018.08.019 **ISSN : 1976-7277**

1. Introduction

Beginning with the appearance of the Morris Worm [1], the exploitation of memory corruption vulnerabilities to realize attacks has accumulated nearly 30 years of history. Due to their ubiquity, particularly for unsafe languages such as C and C++, and their high availability, memory corruption vulnerabilities remain one of the most dangerous vulnerabilities, and this condition is not expected to change in the foreseeable future.

Attacks based on memory corruption vulnerabilities can be divided into control data and non-control-data attacks, according to the type of the target data. Control-data attacks overwrite control data (such as the return address [2]) using memory corruption vulnerabilities (such as buffer overflow and format strings) to direct the target program to an unintended control flow (such as new inserted code or existing code in the memory). Non-control-data attacks were first described by Chen et al. [3] in 2005. Such attacks overwrite certain security-critical non-control data of the target (such as the user identity data, configuration data, or decision-making data) to achieve the attackers' goals without subverting the intended control flow.

In their long struggle against memory attacks, researchers have proposed multiple effective defense mechanisms. The vast majority of existing methods are focused on control-data attacks, which remain the predominant type. From the StackGuard [4] to ASLR [5], DEP [6] and CFI [7], these methods have been deployed in practical systems and greatly limit the implementation of control-data attacks. The chance for attackers to realize control-data attacks is continually becoming smaller. Inevitably, increasingly more attackers will turn to non-control-data attacks. In addition, non-control-data attacks can easily bypass all of the above defense methods and hold the same threat as control-data attacks [8]. However, there are only a limited number of defense methods aimed at defending against non-control-data attacks at present. It is an important strategic opportunity for us to remedy this imbalance before non-control-data attacks take the place of control-data attacks in the mainstream. Therefore, we focus only on non-control-data attacks in this study.

We present a method to defend against non-control-data attacks by influence domain monitoring (IDM). First, we extract all security-critical non-control data by analyzing the source code of the target program to construct the security-critical non-control-data set (SCNS). Then, for each variable in SCNS, we locate all possible operations that can modify its value. We call these operations the domain borders of this variable, and we can ensure that the value of a variable will not change between two adjacent domain borders of this variable. The basic theory of IDM is as follows: after finding all domain borders of all variables in the SCNS, we equip the monitor code to contain privileged instruction that can lead to a trap in a hypervisor after all domain borders and before all operations that use these variables in the SCNS as operands. During runtime, the hypervisor recodes the valid value of each variable in the SCNS after the domain border and retrieves the current value of the variable before the variable is used. This approach produces an alert and aborts the program if the current value is not equal to the valid one (indicating the variable has been tampered with by certain invalid operations, such as buffer overflow). IDM ensures that all security-critical non-control data are defined by valid operations and used with valid values that are defined by valid operations; thus, non-control-data attacks will fail because any unintended modification can be detected and prevented. The major contributions of our work are summarized as follows:

- We are the first to propose a definition for the data influence domain and apply it to secure non-control data.
- We propose a method to extract a complete SCNS. The SCNS provides an important reference value for non-control-data protection.
- We construct a non-control-data-monitoring architecture combined with instrumentation and a hypervisor, which ensures a satisfactory monitoring effect and self-security.
- We implement a prototype of IDM for the Intel x86 Linux kernel, which uses Intel's hardware-assisted virtualization technology (Intel VT) [9].
- We evaluate the efficacy of our implementation against various types of non-control-data attacks and the performance overhead using SPEC CPU2006 benchmark and nginx. Our evaluation shows that IDM can prevent most non-control-data attacks with an average overhead of 36%.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 defines the threat model and assumptions. Section 4 describes the design of the IDM. Section 5 discusses key technologies and challenges to realizing the IDM. Then, Section 6 evaluates the efficacy and overhead of our implementation. Section 7 discusses the limitations of our current design and future work. Finally, Section 8 concludes the paper.

2. Related Work

Since Chen et al. [3] first proposed the concept of non-control-data attacks, a series of such attacks have been developed by researchers [10-14]. Recently, non-control-data attacks have been raised to a new level by data oriented programming (DOP) [14], which provides a systematic technique to construct expressive non-control-data attacks. Nevertheless, the development of defense mechanisms is far behind the development of non-control-data attacks. Existing defense methods can be divided into four types according to their different strategies.

Memory safety. Many proposed technologies exist whose goal is to add memory safety to C/C++. CCured [15] and Cyclone [16] enhance the safety of the C language using "fat-pointers" to perform boundary checks for unsafe memory dereference, but these techniques are not practical and are incompatible with legacy code. SoftBound [17] and CETS [18] use compiler-based instrumentation to provide spatial or temporal memory safety, but the high performance overhead (116% average overhead on the SPEC CPU 2000 benchmark) hinders their usability. DataShield [19] attempts to reduce the overhead by simply protecting sensitive data with boundary checking. However, this approach suffers from the characteristic limitations of boundary checkers. For example, it cannot prevent attacks that exploit format string vulnerabilities to write one specific memory address directly without covering adjacent memory contents. The IDM can prevent this type of attack as it detects attacks according to the real value of the variable instead of boundary checking.

Data-flow integrity (DFI). Castro et al. [8] proposed DFI to protect programs against non-control-data attacks. DFI first generates the data-flow graph (DFG) of the target program by static reaching definitions analysis and then ensures that the data flow during runtime is allowed by the DFG. DFI can defend against a large spectrum of attack vectors, including control-data and non-control-data attacks. However, DFI can miss certain attacks because of imprecisions of reaching definition identifiers, such as an attack that overflows a buffer in a structure to overwrite a security-critical field in the same structure. In addition, it introduces an

average 104% time overhead and approximately 50% space overhead. IDM provides accurate protection without keeping a complex runtime definitions table and reaching definition set for all data, and it does so with a relatively low time and space overhead.

Data isolation. Isolation mechanisms are generic approaches to protecting critical data. These techniques can be divided into software-based [20, 21] and hardware-based methods [22, 23, 24]. Without a hardware assistant, software-based methods typically introduce high performance overhead [21]. Therefore, some methods sacrifice security for better performance, making it possible for information leakage or brute-force attacks to succeed [25, 26]. IDM does not have this problem because it protects the non-control data without isolation assumptions.

Other methods. Certain other methods also have the capacity to prevent non-control-data attacks. One representative method is dynamic taint analysis (DTA) [27, 28, 29]. Such mechanisms work on binaries and can detect both control-data and non-control-data attacks. Nevertheless, these mechanisms can produce false positives and introduce high performance overhead without a hardware assistant. DSR [30] randomizes the representation of data stored in memory using an XOR cipher on the contents of memory with a random key, making it challenging for attackers to modify the contents correctly. However, DSR is not binary compatible as it must recompile the target and relative libraries. In addition, this technique is subject to certain information leak attacks [31], as is the case for other randomization methods. SIDAN [32] and Gibraltar [12] extract data invariants from source code and verify these invariants during runtime; thus, these approaches cannot detect data attacks without violating any invariant constraints. Torres et al. [33] attempted to detect non-control-data attacks using hardware events. However, the analysis results showed that hardware events are susceptible to interference and are less reliable, which leads to false positives and false negatives.

3. Threat model and assumptions

Our threat model assumes that attackers can exploit memory corruption vulnerabilities of the target program to perform arbitrary reads and writes. We assume that the target program is originally benign and that the system has deployed enough defense mechanisms against control-data attacks (such as DEP [6], fine-grained CFI [7, 34], and ASLR [5]) so that attackers cannot mount any control-flow hijacking attacks and must turn to non-control-data attacks. We also assume that the operation system, the hypervisor and hardware are trusted.

4. Design

The IDM is designed to protect non-control data against attacks. When attackers have the chance to modify only non-control data to realize attacks, they typically exploit the modified non-control data in two ways: using it directly to achieve the goal (e.g., the currentuid in **Fig. 1** (a)) or using it to influence other non-control data that cannot be directly modified by known memory corruption vulnerabilities via subsequent arithmetic (e.g., the variable b in **Fig. 1** (b)).

The key idea behind IDM is that even if attackers tamper with the non-control data successfully by memory corruption vulnerabilities, they cannot use the modified non-control data as they intend under the protection of IDM. IDM monitors the influence domain of non-control data and ensures that the invalid value of non-control data cannot be used by other code directly or transmitted to other non-control data indirectly.

```
…
currentuid=1;
…
//contain one memory corruption vulnerability
//the attacker modifies currentuid to 0
…
if(currentuid)(…)else
setuid(currentuid);
…
```

(a)

```
…
a=1;
…
//contain one memory corruption vulnerability
//the attacker modifies a to 0
…
b=a+1;
if(b==2){…}else{…}
…
```

(b)

Fig. 1. Two ways to utilize modified non-control data

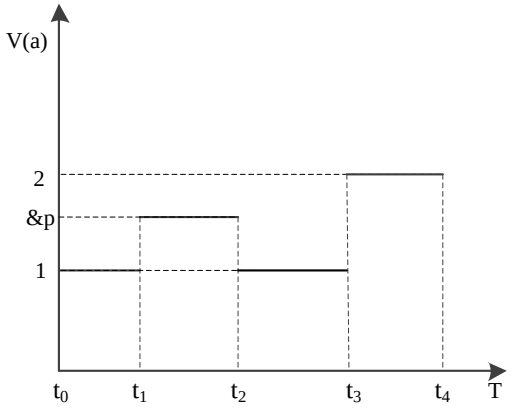
4.1 Influence domain

4.1.1 Definition

Each variable has its own life cycle, and each can be assigned different values in different phases of the life cycle. Taking the code in **Fig. 2** (a) as an example, variable a is declared and initialized to 1 at t_0, and its life circle is from t_0 to t_4. The state of a in different phases is shown in **Fig. 2** (b). The value of a can be updated only by the assignment statements and remains unchanged between two adjacent assignment statements. For example, a is equal to 1 from t_0 to t_1. Based on the above analysis, we give the following three definitions in both temporal and spatial scales to describe this characteristic better.

```
t0: int a=1;
    …
    //may exist one memory corruption vulnerability
t1: a=&p;
    …
    //may exist one memory corruption vulnerability
    …
t2: a=1;
    …
    //may exist one memory corruption vulnerability
    …
t3: a=a+1;
    if(a==2){…}else{…}
t4: end
```

(a)



(b)

Fig. 2. Code example

Definition 1: A variable possesses its own memory space once it is declared. We call this space the space influence domain (SIDom) of the variable. For variable α, its SIDom is represented by $SIDom(\alpha)$ and the content of its SIDom is represented by $Val(SIDom(\alpha))$.

The size of $SIDom(\alpha)$ is determined by the type of the variable α and will not change unless the data type changes during runtime. $Val(SIDom(\alpha))$ is simply the value of the variable α.

Definition 2: A variable is alive from when it is declared to the end of the procedure that it belongs to or to when it is freed. We call this time period the temporal influence domain (TIDom) of this variable. For variable α, its TIDom is represented by $TIDom(\alpha)$.

The $TIDom(\alpha)$ is consistent with the life cycle of α. For the variable a in **Fig. 2** for example, $TIDom(a)=[t_0,t_4]$.

Definition 3: In $TIDom(\alpha)$, all valid assignment operations of the program divide $TIDom(\alpha)$ into many sub-temporal domains, and $Val(SIDom(\alpha))$ remains unchanged in each of them. We call the sub-temporal domains the fixed-value temporal influence domain (FTIDom) of α, which is represented by $FTIDom(\alpha)$. The fixed value of $FTIDom(\alpha)$ is represented by $FVal(FTIDom(\alpha))$.

One $FTIDom(\alpha)$ is a sub-influence domain of $TIDom(\alpha)$. For a in **Fig. 2** for example, $[t_0,t_1]$, $[t_1,t_2]$, $[t_2,t_3]$ and $[t_3,t_4]$ all are $FTIDom(a)$, and the corresponding fixed values of them are 1, &p, 1 and 2.

Taking $InVal(FTIDom_i(\alpha))(i\in N^+)$ as the value of α when it enters $FTIDom_i(\alpha)$, then $FVal(FTIDom_i(\alpha))$ is equal to $InVal(FTIDom_i(\alpha))$. The value of α is maintained at $FVal(FTIDom_i(\alpha))$ until it enters another $FTIDom_j(\alpha)(j\in N^+, j\neq i)$ under normal conditions. We take $UseVal(\alpha)$ as the value of α when α is used by any instruction in $FTIDom_i(\alpha)$. According to above three definitions, we can deduce the following conclusions:

In an FTIDom of variable α:

$$FVal(FTIDom_i(\alpha))=InVal(FTIDom_i(\alpha)) \tag{1}$$

$$UseVal(\alpha)=FVal(FTIDom_i(\alpha))(i\in N^+) \tag{2}$$

$$Val(SIDom(\alpha))=FVal(FTIDom_i(\alpha))(i\in N^+) \tag{3}$$

In addition, TIDom and FTIDom of variable α have the following relationship:

$$TIDom(\alpha)=FTIDom_1(\alpha)\cup FTIDom_2(\alpha)\cup FTIDom_3(\alpha)\cup\cdots\cup FTIDom_n(\alpha)(n\in N^+) \tag{4}$$

4.1.2 Inspiration

The states of variables in programs are constantly changing. More complex programs are associated with more non-control data in the program and a larger set consisting of the various states of the non-control data. When considering the entire program as an object to analyze the changing process of the non-control data, the approach tends to fail as a result of complexity, particularly in the case of large-scale applications, which is one of the most important challenges in non-control-data protection. We are inspired by the influence domain and attempt to develop a method to break the complex problem into smaller and more manageable issues.

We decompose all non-control-data protection into the corresponding influence domains monitoring. Then, one variable's IDM can be decomposed to monitor the SIDom and TIDom. The TIDom monitoring can be further decomposed into the FTIDom monitoring. The transition process above is shown in **Fig. 3**.

Enlightened by above analysis, we design IDM and provide a new strategy to solve the problem of non-control-data protection. IDM must monitor SIDom and FTIDoms for all security-critical non-control data of the target program to enforce the non-control-data security of the entire program.

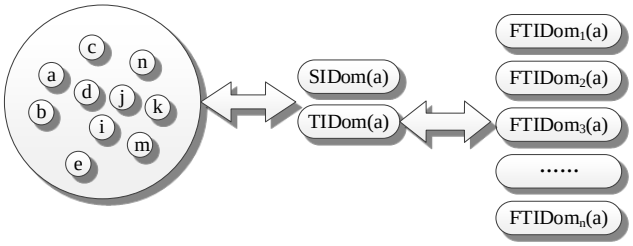


Fig. 3. Decomposition and integration procedure of non-control-data protection

4.2 Defense mechanisms

Attackers must exploit memory corruption vulnerabilities to modify one non-control datum to a particular value if they are to achieve a non-control-data attack. Furthermore, according to the thread model and above analysis, we know that the value must be different from the use-value of the FTIDom. In this section, we discuss how to defend both methods shown in **Fig. 1** to use modified non-control data by monitoring the influence domains.

A. Checking before use (CBU)

Regardless of the use of the modified non-control data, the data must be read first. Therefore, if we can determine that the non-control data to be used by the next instruction have been tampered with, then we can prevent the program from executing that instruction, and the non-control-data attack will be defeated. Based on definition 3 and conclusion (2), we know that the read operation must occur in one FTIDom and that in all cases the value of the variable is equal to the fixed value of the FTIDom. Thus, we can perform the following verification to defend against non-control-data attacks.

When the target program runs a read instruction on a variable α in $FTIDom_i(\alpha)$, we first check whether $UseVal(\alpha)$ is equal to $FVal(FTIDom_i(\alpha))(i \in N^+)$. If the two values are equal, the status of the variable α is regarded as normal. Otherwise, the value of α has been modified before this time point by an invalid assignment operation; thus, the target program should be aborted, and an alert should be given to the user.

Taking **Fig. 2**, for example, if a has been modified to zero between t_2 and t_3 via a memory corruption vulnerability (such as buffer overflow), the Boolean variable in the next statement will become false, but the normal result should be true. Then, the attacker can change the execution of the program. However, if we add a verification process before t_3 to check whether $UseVal(a)$ is equal to $FVal(FTIDom_{[t_2,t_3]}(a))$ (i.e., 1) and we terminate the program if they are not equal, this attack will be defeated.

B. Updating after definition (UAD)

To perform CBU, we must know the exact fixed value of every FTIDom of each variable. As the value of a variable is dynamically changing during the running of the program, how to know the valid fixed value is an important challenge. According to definition 1, the value of a variable is stored in the fixed SIDom, and this condition will not change during the entire life

cycle of the variable. Consequently, we monitor all valid write operations to the SIDom and record the contents written by these operations as the valid fixed value of every FTIDom.

According to conclusion (1) and definition 3, the fixed value of each FTIDom is simply the value written by the assignment statement at the start of the influence domain, that is, $InVal\left(FTIDom_i\left(\alpha\right)\right)\left(i \in N^+\right)$. Therefore, the content in SIDom just after the assignment statement is the fixed value of the corresponding FTIDom. IDM monitors all valid assignment operations on non-control data and reads the content in SIDom after these operations, then treats this content as the fixed value of the FTIDom, which starts with this assignment operation.

According to definition 3, a new FTIDom begins every time an assignment statement finishes executing. IDM updates the fixed value of the new FTIDom with the content in the SIDom of the variable at this time.

Combining CBU and UAD, IDM can ensure that all $FVal\left(FTIDom_i\left(a\right)\right)\left(i \in N^+\right)$ come from valid assignment statements of the target program and that $UseVal\left(a\right)$ is always equal to the $FVal\left(FTIDom_i\left(a\right)\right)\left(i \in N^+\right)$ during the runtime, which can stop all invalid values from being used by the program and defend against all non-control-data attacks.

4.3 SCNS definition

As all non-control data have their own influence domain, the performance overhead will be high if we monitor all non-control data of the target program, hampering the practicality of IDM. Therefore, we must determine which non-control data are closely related to security and monitor only their influence domains.

Chen et al. [3] identified four types of security-critical data that may be subject to non-control-data attacks: configuration data, user input data, user identity data and decision-making data. Most security-critical non-control data can be found and identified according this research. However, we find that certain other critical data will be neglected if we use only these standards to locate security-critical data.

Data are fluid, dynamic and not independent; thus, one datum is often affected by several other data. We assume that one variable b is a decision-making datum that is the sum of another variable c and constant 1 but that c does not belong to any type of security-critical non-control data. Without protection, c may be modified by attackers via memory corruption vulnerabilities. In this case, we assume that the original value of c is 0 and that one attacker overwrites it to 1 later. Then, the value of b will be assigned to 2 instead of 1 after executing the operation b = c+1, which leads to the program selecting an unintended control flow. As 1 and 2 are both valid values for b and the attacker does not tamper with b directly, this non-control-data attack cannot be detected or prevented by protecting b only. This result also reminds us that not only the explicit security-critical non-control data (e.g., variable b) but also other implicit data that could influence those explicit data directly or indirectly (e.g., variable a) should be protected. Another type of non-control data that should be protected is the output data of the target program. Currently, many attackers can redirect the output data to a special memory location and disclose important private information (e.g., password and user ID) to facilitate more complex attacks. Information leakage is one of the most notable challenges for software protection. Thus, the output data also should be monitored to enhance security.

In this study, we further extend the scope of security-critical non-control data and attempt to define a complete set that contains all possible security-critical non-control data.

Definition 4: We call a set the SCNS if it contains the following non-control data:

- Configuration data. Including configuration files and file path directives, these data specify access control policies and the locations of trusted executables.
- User identity data. The data that can be used to describe a user's identification, such as the user ID and group ID.
- Decision-making data. The data (non-control data) that are used to determine which branch to run according to its value, such as the Boolean variables.
- User input data. The data that come from user input operations.
- Output data. The data that are output by the program during the runtime.
- The data that can influence the above five types of data directly or indirectly.

The first four types of security-critical non-control data have been defined precisely by Chen et al. We extend the scope with the last two types. Our new definition treats the last two types of data as being as important as the others. We can acquire a much more comprehensive SCNS with the new definition and avoid the limitations of previous research.

5. Implementation

IDM can be separated into two phases: preprocessing and runtime monitoring. In the preprocessing phase, we extract SCNS from the LLVM intermediate representation (IR) of source files and produce protected executable files by instrumentation and compilation. During runtime, we monitor the state of all security-critical non-control data via a lightweight hypervisor and handle exception events. The overall architecture of IDM is shown in **Fig. 4**.

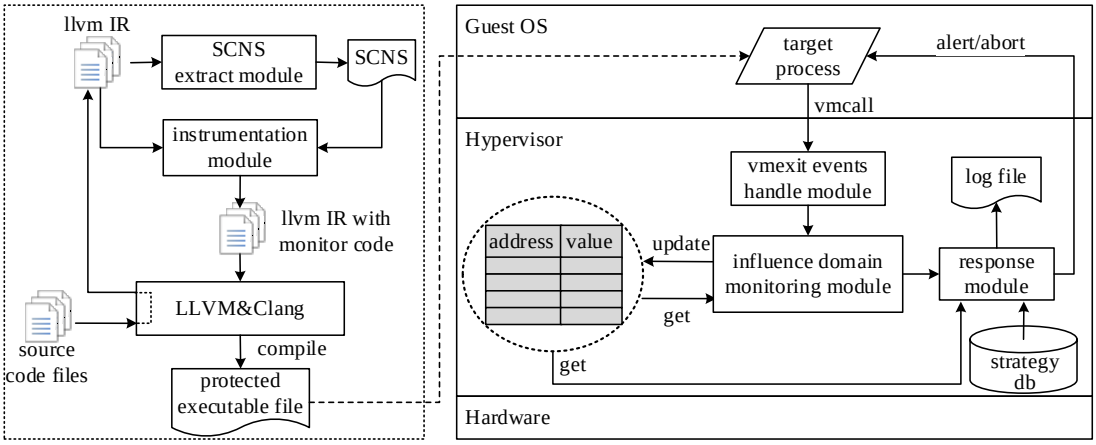


Fig. 4. Overall architecture of IDM

In this section, we will describe related key methods for IDM implementation.

5.1 SCNS construction

We extract SCNS using a combination of three analyses: sensitive instructions analysis, sensitive functions analysis and data dependency analysis. We compile the source code into LLVM IR and perform the above analysis on the result. There are two reasons to choose LLVM IR as the analysis target: first, LLVM IR is well formed, and all operations on variables

(e.g., declaration, definition, store and load) are unambiguous and easily analyzed; second, our method can be deployed for other languages with LLVM front-ends.

5.1.1 Extraction

To identify configuration data and user input data, we adopt sensitive functions analysis and taint analysis. Both configuration data and user input data must be accepted by the program before joining in the data flow of the program. Therefore, we locate related functions that accept configuration data and user input data at first (e.g., file_open, readline, or getchar) and treat variables influenced by these functions directly as external taint sources; then, we adopt taint analysis to trace the propagation of these variables and treat a variable as a taint if its value is affected by existing taints. Finally, we treat all taint variables as security-critical non-control data. Taking ghttpd-1.4 as an example, the configuration file ghttpd.conf is read by the function fopen in util.bc (the LLVM IR file corresponding to util.c of ghttpd-1.4), and we then trace all related data influenced by the data read out from ghttpd.conf and label them as security-critical non-control data.

To extract the user identity data, we first perform sensitive functions analysis to locate functions that can modify the UID or GID of the current process, such as setuid and seteuid. Then, parameter variables of these functions are used to describe the identity belonging to the user identity data; we obtain these variables through parameter analysis. Taking sudo-1.8.3p1 as an example, in sudo.bc (the LLVM IR file corresponding to sudo.c of sudo-1.8.3p1), the sensitive function setresuid is called to set the new ruid, euid and suid according to its three parameters i32 %131, i32 %134 and i32 %137. Hence, %131, %134 and %137 are all identity data.

Output data are straightforward to obtain. We first find all functions that can be used to perform output operations and recognize all variables that will be output during the runtime in parameters according to their definitions. Taking util.bc in ghttpd-1.4 as an example, printf is called to output i8 * %97, so %97 is part of the security-critical non-control data.

```
1   %a = alloca i32, align 4
2   %b = alloca i32, align 4
3    %ptr = alloca i32, align 4
4    Label:
5          %cond = icmp eq i32 %a, %b
6          br i1 %cond, label %IfEqual, label %IfUnequal
7      IfEqual:
8          ret i32 1
9      IfUnequal:
10           ret i32 0
11     store i32 3, i32* %ptr
12     %val = load i32, i32* %ptr
13     switch i32 %val, label %otherwise [
14         i32 0, label %onzero
15         i32 1, label %onone
16         i32 2, label %ontwo ]
```

Fig. 5. Code example to show the decision-making data analysis method

Decision-making data are the most widely distributed security-critical non-control data. For collecting decision-making data as comprehensively and completely as possible, we traverse the entire program to locate all conditional branch instructions (i.e., the “br”

instruction and “switch” instruction in LLVM IR). Then, we analyze arguments of each instruction to obtain the decision-making data. As shown in **Fig. 5**, there is a conditional branch “br” in line 6 and a “switch” instruction in line 13, and the arguments “i1 %cond” and “i32 %val” determine where to execute next; therefore, both “%cond” and “%val” are decision-making data.

After extracting all five types of security-critical non-control data, we adopt data dependency analysis to obtain the sixth type. This type of data can influence other types directly or indirectly by binary operations (e.g., add, sub, or mul), bitwise binary operations (e.g., shl, and, or, or xor), memory access operations (e.g., load or store) or other instructions that can change the values of variables. We call these operations sensitive operations. As before, we extract the input data and configuration data using taint analysis. All related data can be found by this process; thus, the sixth type of data that we should locate in addition refers primary to the data that can affect the user identity data, output data and decision-making data. We use backward dependency analysis to achieve this goal. At the beginning, we construct a data set that consists of all user identity data, output data and decision-making data. Then, for every element of the set, we perform backward analysis starting from the location of the element where it is found. When we find a sensitive operation that uses the element as the left operand, we add the right operands (non-constant variables) to the data set. The backward analysis will be terminated when it traverses the declaration of this variable. We iterate this process until every element of the data set has been analyzed. Taking the code in **Fig. 5** as an example, we have determined that “%val” in line 13 belongs to decision-making data; thus, we start backward analysis starting from line 13. In line 12, we find “%val” is the left operand of a load operation; thus, we insert the right operand “%ptr” into the set. At this point, “%ptr” also belongs to the security-critical non-control data. Through backward dependency analysis, we can identify more security-critical non-control data and construct a more complete SCNS.

5.1.2 Optimization

In LLVM IR, all local variables are allocated memory by the “alloca” instruction (in stack) or the “malloc” function (in heap), and global variables are declared with “global” identifiers. However, when a program is translated into LLVM IR, many temporary variables will be added into the code used to deliver values (e.g., a value or address of a real variable, or the result of a binary or bitwise binary operation) between different instructions acting as registers, and they do not exist in the original source code, such as “%cond” in line 5 and “%val” in line 12 in **Fig. 5**, which are neither local nor global variables of the program. Multiple temporary variables make SCNS complex and redundant. Therefore, we design a method to optimize the SCNS constructed in Section 5.1.1.

First, we extract local and global variables in SCNS and construct a subset for each of them. Then, we handle the other temporary variables in SCNS as follows:

1) **Those used to deliver values or addresses of real variables.** We regard these temporary variables as different expression forms of real variables. If a temporary variable is the value or address of a local or global variable, then this temporary variable belongs to the same subset as the local or global variable.
2) **Those used to save operation results.** We analyze the operands of the operations first. We handle temporary variables in operands via method 1); for local or global variables in operands, no action is necessary because each of them must belong to one subset, which was constructed at the beginning. Finally, we delete the temporary variables used to save operation results from the SCNS.

By this method, we analyze and classify all variables in SCNS into different classes that correspond to local or global variables. Thus, we focus on protecting local or global variables instead of temporary variables added by the LLVM compiler. For example, we verify "%a" and "%b" instead of "%cond" in **Fig. 5**. This approach greatly reduces the complexity and improves the availability of SCNS.

5.2 Instrumentation

5.2.1 Basic method

For monitoring the influence domain of each element of SCNS during runtime as described in Section 4.2, we implement monitor code in the LLVM IR of the target program. According to the defense mechanism, the monitor code checks whether the current value of a variable is equal to the fixed value of the FTIDom before the variable is used. Thus, we must know the fixed value of every variable's FTIDom and the value of the variable immediately before it is used.

The fixed value of the FTIDom is more challenging to obtain. Different FTIDoms may have different fixed values, which can be changed by assignment statements during runtime. We must trace the assignment statement to record the fixed value of every FTIDom if we wish to know it precisely. LLVM is a load/store architecture, and only the "store" operation can modify memory [35]. That is, for one local or global variable, its value is fixed between two adjacent "store" operations. Therefore, we need only locate all "store" operations that use the variable in SCNS as an operand to divide the variable's TIDom into certain FTIDoms through traversing the LLVM IR.

When we obtain all "store" operations, the borders of all variables' FTIDoms can be labeled. Then, we insert monitor code after every "store" instruction to read the value of the variable at this point, and this value is exactly the fixed value of the FTIDom, which starts with this "store" instruction and ends with the next one.

The value of the variable before it is used can be obtained by inserting code that reads the variable before all read operations. Therefore, we traverse the LLVM IR to find all read operations on every variable of the SCNS, and insert monitor code before these operations to obtain the current value of this variable.

The verification process can be implemented in the program by inserting additional code. However, in this situation, both the fixed value of the FTIDom and the current value are saved in the memory of the program; thus, attackers can easily bypass this verification process by modifying this content once they have the ability to perform arbitrary reads and writes through memory vulnerabilities. In addition, leaving the security mechanism in untrusted programs is unwise. Therefore, we prefer to implement the verification process in a lightweight hypervisor, and the inserted monitor code simply submits the variable status to the hypervisor.

We add monitor code by inserting new high-level code into the LLVM IR of the program. The monitor code has the following form:

```
call void asm sideeffect "vmcall",
"{ax},{cx},{dx},~{dirflag},~{fpsr},~{flags}" (i32 %t, i32 * %n, i32 %l)
```

The monitor code saves three values “%t”, “%n”, “%l” in registers EAX, ECX and EDX, respectively, then executes “vmcall” instruction to trap in the hypervisor. “%n” and “%l” are the address and the length of the variable, respectively, and “%t” indicates whether the monitor code is an update or verification code, which tells the hypervisor how to handle this vmcall event (the details of the process will be described in Section 5.3).

5.2.2 Handling function calls

Function calls in programs should be considered separately when instrumented. These called functions may have their own security-critical non-control data, and these variables must be instrumented and monitored; however, even if the called function does not contain any security-critical non-control data itself, it may have data pointer arguments that point to security-critical non-control data, which means that variables pointed to by these pointers can be modified by this function. Therefore, we must instrument and monitor the called function. We handle these functions in the following ways based on their types:

1) For functions declared in the program itself, as their definitions are also in the source files of the program, they have been instrumented in the way described in Section 5.2.1. Therefore, we treat them simply as use operations on security-critical non-control data if they use it as an argument and instrument a verification code before the callsite.
2) For functions in dynamic-link libraries, the best monitoring approach is to analyze their source code and then implement monitor code. However, this method requires the modification and recompilation of dynamic-link libraries, which will lead to the method being incompatible with legacy code. Therefore, we use another method to solve this problem.
    a) For functions that use sensitive pointers as arguments, we take two steps. First, for one function, we treat it as a use operation on the security-critical non-control data pointed to by the sensitive pointer and instrument a verification code before the callsite. Second, we analyze its source code (assuming that the dynamic-link library is open source) and check whether it has valid write operations on the security-critical non-control data. If it does, then the function can modify this variable legitimately; thus, we instrument an update code after the callsite and treat the current value as the new valid value; otherwise, the function cannot modify this variable under normal circumstances, and we simply insert another verification code after the callsite to ensure that this variable will not be modified during the runtime of the function.
    b) For the other functions, as their execution will not modify any security-critical non-control data belonging to SCNS, we simply treat the called functions as general code and avoid any additional processing.

This method makes IDM compatible with legacy code but may result in false negatives when attackers exploit memory corruption vulnerabilities in called functions of dynamic-link libraries to modify the security-critical non-control data pointed to by the sensitive pointer arguments. False negatives can also arise if attackers modify local non-control data belonging to these functions themselves but not to SCNS, as we do not delve deeply into the definitions of these functions. However, this method limits possible attacks so that these attacks can only exploit memory corruption vulnerabilities in dynamic-link libraries and cannot modify non-control data in SCNS that are not accessed during their normal execution.

5.2.3 Optimization

We found through experiments that redundant monitor code will be instrumented by the method described in Section 5.2.1. The reason is that this method does not consider application contexts at run time and lacks overall consideration: the method focuses only on security-critical non-control data and regards all inserted code as independent. However, this inserted code is not isolated and independent; it is one part of the control-flow or data flow of the program. As this redundant inserted code does not contribute to security and only increases overhead, we propose an optimization method to remove redundant monitor code based on the control-flow and data-flow analysis. The method can be described from two perspectives: in one basic block and between different basic blocks.

1) Optimizations in one basic block: If no function calls or write operations exist on any local or global variables between two adjacent read operations on the same security-critical non-control data in one basic block, attackers cannot modify this security-critical non-control data between two adjacent read operations. Therefore, the verification code before the second read operation can be removed. In addition, if one assignment operation immediately follows a read operation, the value obtained by the read operation must be the value assigned by the assignment operation. Thus, the verification code inserted before the read operation can also be removed.
2) Optimizations between basic blocks: We analyze the control flow and perform optimization on every execution path. First, we remove the repetitive monitor code directly. For example, if basic block A is the former of basic block B in one path of the control flow and there is verification code at the end of A and at the start of B for the same security-critical non-control data, the verification code at the start of B can be removed. We can also treat any two adjacent basic blocks as a whole and optimize them by the method described above to further remove redundant monitor code.

These optimizations can effectively reduce the number of non-essential verification checks without affecting the safety of the method. Take mcfutil.bc as an example, which is the LLVM IR file corresponding to mcfutil.c of 429.mcf in SPEC CPU2006. The number of verification checks in this file can be reduced by approximately 20% (from 176 to 140) through optimization.

5.3 Runtime monitor

After the target program is instrumented and recompiled, the protected executable file can be produced. Then, during the application runtime, the program will be trapped in the hypervisor when it runs the monitor code. The monitor module in the hypervisor handles these events according to the monitor code type.

When an update code arises, the monitor module reads the value of the variable in the SIDom that started from the address “%n” to “%n+%l”, and this value is simply the fixed value of the current FTIDom. When a verification check occurs, the verification process will be performed by the monitor module. First, the process reads the current value of the variable from the SIDom and then compares the current value with the fixed value of current FTIDom. If these two values are equal, no additional action needs to be taken; if not, an alert is thrown to the user, and the program is aborted.

Using the runtime monitor, any non-control-data attacks that modify the security-critical non-control data are detected and prevented before the modified variable is used in any subsequent execution.

6. Evaluation

In this section, we evaluate the overhead of our implementation and its effectiveness at defeating non-control-data attacks.

6.1 Overhead

We ran nine benchmarks of SPEC CPU2006 to evaluate the overhead of IDM, six integer benchmarks (401.bzip2, 429.mcf, 456.hmmer, 458.sjeng, 462.libquantum and 473.astar) and three floating point benchmarks (470.lbm, 444.namd and 450.soplex). In addition, to better evaluate the impact for web server programs, we measured the overhead of nginx-1.4.2.

We ran all experiments on an Ubuntu 12.04 x86-32 system with a 3.4 GHz Intel Core™ i7-3770 processor and 16 GB of memory. The kernel version was 3.2.0-29-generic-pae. We used LLVM-3.5.0 and Clang-5.0.0 as analysis and compilation tools. In addition, we designed and realized a lightweight hypervisor, assisted by Intel VT, that monitored only instructions that caused virtual machine (VM) exits unconditionally [9] to reduce the overhead introduced by the hypervisor as much as possible.

We ran each experiment three times and present the average results.

6.1.1 Space overhead

For evaluating the space overhead introduced by IDM, we translated all 9 benchmarks and nginx-1.4.2 to LLVM IR files and performed SCNS extraction. Then, we instrumented all IR files according to SCNS. Finally, we compiled these IR files to binary executables. We counted the number of the monitor code insertions and measured the size of these binaries. The results are shown in **Fig. 6** and **Fig. 7**.

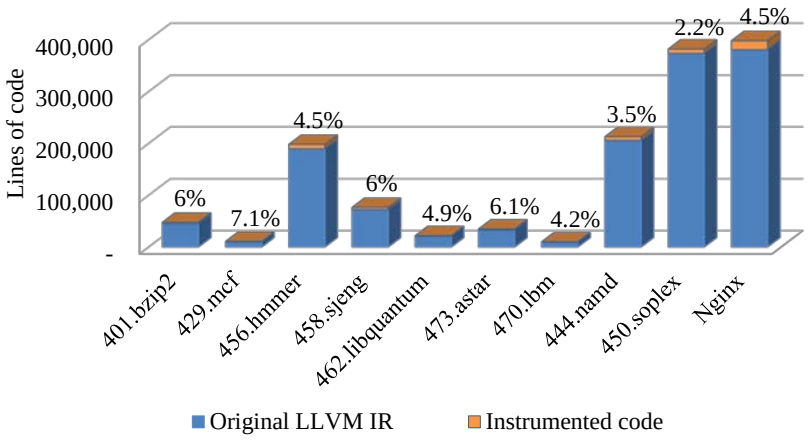


Fig. 6. Statistics on instrumented monitor code

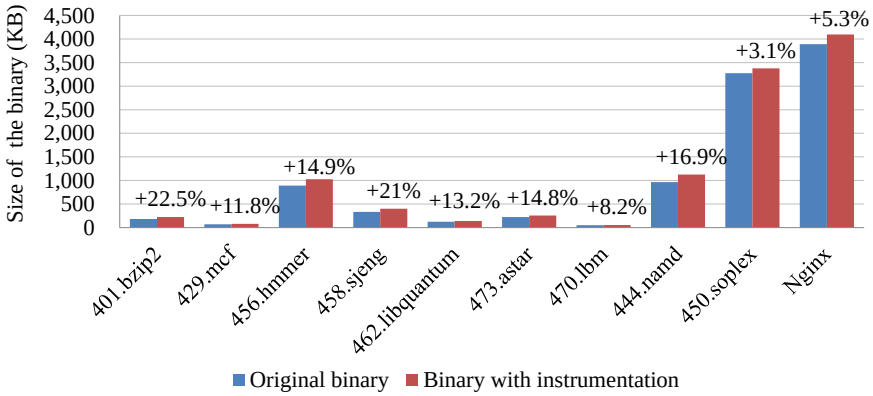


Fig. 7. The binary size before and after instrumentation

Fig. 6 shows that the average number of monitor code insertions in all test cases accounted for 4.9% of the original code and that no direct relationship exists between the proportion of inserted monitor code and the original code size. This result occurs primarily because IDM instruments all monitor code according to SCNS; thus, the amount of inserted monitor code is affected by the size of SCNS and the complexity of operations on security-critical non-control data in the program. The greater the number of security-critical non-control data contained in the target program and the more complex the operations on that data, the greater the proportion of monitoring code that is inserted.

Fig. 7 shows the final sizes of the executable files after instrumentation of the monitor code. The largest space overhead is 22.5%, and the average is 13.2%. IDM introduces less overhead than DFI [8], which has an average of 50% space overhead.

6.1.2 Time overhead

We measured the time overhead of all 9 benchmarks. As IDM uses a hypervisor for effective monitoring and the entire guest operating system runs on the hypervisor, IDM is expected to affect the performance of entire system. For a better description of the performance overhead introduced by IDM, we measured the overhead of these benchmarks in three different conditions: the Linux system without the hypervisor, with the hypervisor but with no instrumentation and with IDM. The results are shown in **Fig. 8**.

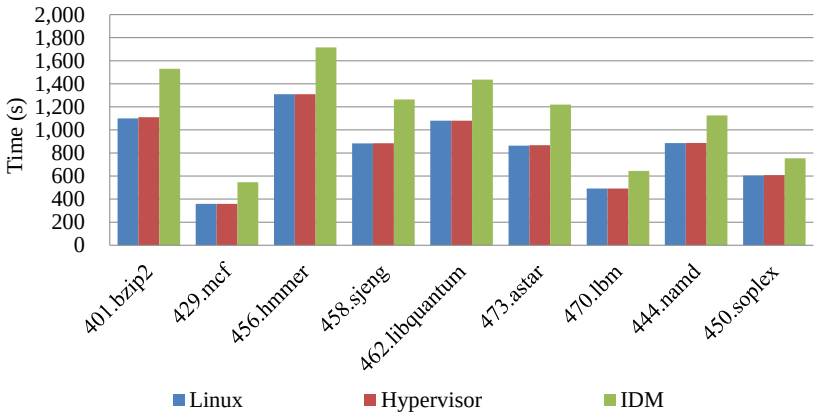


Fig. 8. Time overhead of 9 benchmarks

Fig. 8 shows that the lightweight hypervisor introduces a small overhead owing to Intel VT. With Intel VT, the hypervisor needs only to monitor root operations and does not handle the other events, which greatly reduces the overhead. After IDM is deployed, the target program traps into the hypervisor every time it runs to the monitor code; then, the hypervisor must also perform updates or verification according to the monitor code type. Therefore, IDM introduces a bigger time overhead, averaging 36% relative to the pure Linux system.

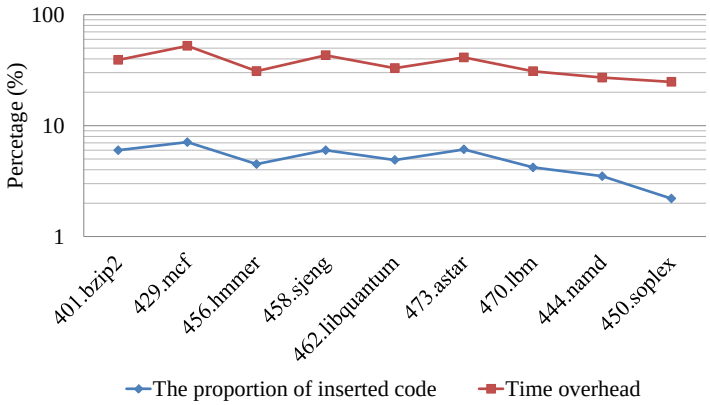


Fig. 9. The relationship between the time overhead and the proportion of inserted code

In addition, through a comprehensive analysis of the proportion of inserted monitor code and the time overhead, we found that the time overhead is proportional to the proportion of inserted monitor code, as shown in **Fig. 9**. The larger the proportion of inserted monitor code, the greater the time overhead of the program. Therefore, if we wish to reduce the overhead, we should focus on decreasing the amount of the inserted code.

As SPEC CPU 2006 consists of CPU intensive benchmarks, we also measured the time overhead of the I/O intensive program nginx. We used webbench-1.5 [36] to measure the processing capacity of nginx under the three different conditions used above. **Fig. 10** shows the measured results. The ordinate axis is the maximum number of concurrent requests without any failures over 30 seconds. The results show that the hypervisor has no significant effect on nginx and the processing capacity of nginx decreases by 11.4% because of IDM. The overhead of nginx is smaller than that of the 9 benchmarks of SPEC CPU2006, indicating that the instrumented code has a greater effect for CPU intensive programs than I/O intensive programs.

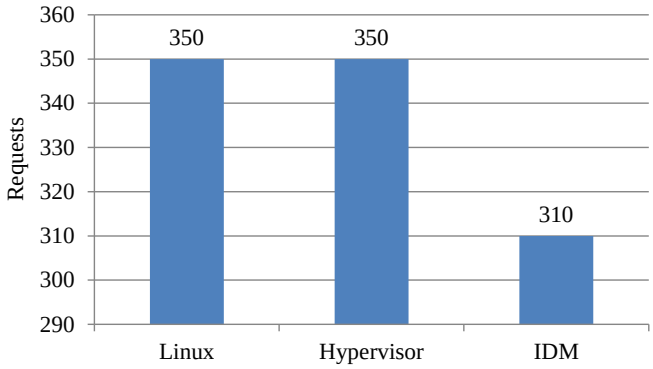


Fig. 10. Processing capacity of nginx under three different conditions

6.2 Effectiveness

We used 8 real attacks, which are listed in **Table 1**, to evaluate the effectiveness of IDM at defending against non-control-data attacks.

The first 6 attacks were described in [13] and [14], which can be downloaded directly from http://huhong-nus.github.io/advanced-DOP/. The ghttpd is a lightweight web server with a stack buffer overflow vulnerability reported in version 1.4.0-1.4.3. The attacker exploits this vulnerability to overwrite the argument of execv() and run an arbitrary executable file (e.g., a non-root shell). The proftpd server uses OpenSSL for authentication. One stack buffer overflow vulnerability can be used to overwrite the address used by a public output function to leak private data. The nginx server is a high-performance HTTP server. Attacks exploit one buffer overflow vulnerability to overwrite the web root directory string from the configuration data to leak the key. The orzhttpd server contains one format string vulnerability that can be exploited to overwrite the root directory string stored on the heap to leak random function addresses. The sudo program allows attackers to run commands as another user on a Unix-like operating system. This attack exploits a format string vulnerability to overwrite the user ID to obtain root privileges. The wuftpd is a free FTP server software for Unix-like operating systems. The attacker can also exploit one format string vulnerability to achieve privilege escalation. We implemented the last 2 attacks using techniques described in [3]. NullHttpd is a multithread web server on Linux with a heap overflow vulnerability that can be exploited by attackers to overwrite arbitrary words in memory [37]. This attack forces NullHttpd to run arbitrary commands by corrupting the CGI-BIN configuration string of this server. SSH is a secure shell server implementation from OpenSSH.org and has been reported to have an integer overflow vulnerability [38]. This attack exploits the vulnerability of overwriting a stack variable called *authenticated*, defined as a Boolean flag, to 1 to indicate that the user has been authenticated, allowing the attacker to log in to the system without being authenticated.

Table 1. Real non-control-data attacks defended against by IDM

No.	Software	Memory corruption vulnerability	Non-control-data attack	Detected?	False positives?
1	ghttpd	stack buffer overflow	run arbitrary commands	√	×
2	proftpd	stack buffer overflow	leak private data	√	×
3	nginx	stack buffer overflow	leak private data	√	×
4	orzhttpd	format string	leak random information	√	×
5	sudo	format string	privilege escalation	√	×
6	wuftpd	format string	privilege escalation	√	×
7	NullHttpd	heap overflow	run arbitrary commands	√	×
8	SSH	integer overflow	bypass authentication	√	×
8	SSH	integer overflow	bypass authentication	√	×

The test results are shown in **Table 1**. IDM can defeat all these non-control-data attacks, as these attacks all operate by modifying certain security-critical non-control data that are protected by IDM. The reason is that attackers essentially must corrupt certain security-critical non-control data in memory to implement real non-control-data attacks. However, when protected by IDM, any corrupted value in memory corresponding to the security-critical non-control data will fail to pass through verification, thus blocking the attack. Taking attack 8 as an example, the Boolean variable *authenticated* is part of the decision-making data, and therefore IDM will instrument the update code after every valid assignment operation to

maintain the valid value as $FVal\left(FTIDom\left(authenticated\right)\right)$. When attackers overwrite *authenticated* to 1 by an invalid assignment operation, $FVal\left(FTIDom\left(authenticated\right)\right)$ remains equal to 0. Hence, the verification code instrumented before *authenticated* is used to decide which branch to execute will detect this invalid modification and abort the target program.

Besides, as **Table 1** shows, IDM did not produce any false positives in these tests, which proved the high usability of this method.

7. Limitations and future work

Although the overhead of IDM is smaller than that of certain other methods such as DFI and SoftBound, IDM must still be further optimized to improve its usability. Monitor code inserted in loops can introduce large time overheads as they will execute many times. Therefore, optimizing the inserted code according to the operating characteristic of one loop is one important component of planned future work. Furthermore, current IDM provides undifferentiated protection for all variables in SCNS. However, not all variables in SCNS are equally important for the target program's security. The corruption of one variable may cause the program to crash (e.g., redirect one data pointer to an invalid memory location), leak sensitive information, or escalate privilege, among other possible results. Therefore, we plan to analyze these security-critical non-control data more meticulously to divide these data into different groups according to the extent of the damage they are prone to do, the likelihood of the attack occurring and the abnormal conditions that may result. We will then protect different variable groups according to different security requirements to reduce the overhead while meeting the security requirements.

IDM may also produce false negatives. We do not analyze and instrument the source code of the dynamic-link library to ensure that our method will be compatible with the legacy code. Thus, as described in Section 5.2.2, although IDM limits the possibility of non-control-data attacks to a great extent, certain attacks that exploit memory corruption vulnerabilities of the dynamic-link library may be missed by our method. When the source code of all dynamic-link libraries on which the target program depends is available, we can instrument all libraries to deploy IDM to reduce the possibility of the above attacks and treat them as private libraries of the target. Although this method can effectively avoid false negatives, it also makes IDM less practical. We plan to adopt one intraprocedural analysis method to better solve this problem in the future. For one function of a dynamic-link library, we will extract security-critical non-control data in its own body and also treat all its data pointer parameters as security-critical non-control data, considering that these pointers may point to other security-critical non-control data outside the function during the runtime of the target. This method can provide higher security than is currently available, and the instrumented dynamic-link library can be shared by all protected applications.

In addition, the SCNS extracted by the current method is relatively complete but not absolute. Certain new non-control-data attacks that do not depend on any variable in SCNS could be proposed. Therefore, IDM and SCNS will be further improved in future work.

8. Conclusion

We presented the IDM method to defend against non-control-data attacks. The data influence domain was first defined by us to describe characteristics of a variable during its life cycle, and IDM was designed to enforce these characteristics by extracting SCNS and instrumenting the target program. Since most non-control-data attacks must modify certain security-critical variables to special values by exploiting memory corruption vulnerabilities, IDM can protect the software from these attacks. IDM monitors the influence domains of every security-critical variable and prevents any modified value from being used by later instructions to realize attack goals. As IDM performs all operations on LLVM IR without modifying the source files and does not produce false positives, this approach offers high practicability. In addition, we evaluated the efficacy of our implementation, and the results showed that IDM can prevent most non-control-data attacks with acceptable overhead.

References

[1] Ted Eisenberg, IDavid Gries, Juris Hartmanis, Don Holcomb, M. S. Lynn and Thomas Santoro, "The Cornell commission: on Morris and the worm," *Communications of the ACM*, vol. 32, no. 6, pp. 706-709, June, 1989. Article (CrossRef Link)

[2] Aleph One, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 7, no. 49, November 8, 1996. Article (CrossRef Link)

[3] Shuo Chen, Jun Xu, E. C. Sezer, Prachi Gauriar and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *Proc. of 14th USENIX Security Symposium*, pp. 177-191, July 31-August 5, 2005. Article (CrossRef Link)

[4] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. of 7th USENIX Security Symposium*, January 26-29, 1998. Article (CrossRef Link)

[5] PaX-Team. PaX ASLR. 2003. Article (CrossRef Link)

[6] Starr Andersen and Vincent Abella, "Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory protection technologies, Data Execution Prevention," *Microsoft TechNet Library*, September, 2004. Article (CrossRef Link)

[7] Martin Abadi, Mihai Budiu, Ulfar Erlingsson and Jay Ligatti, "Control-flow integrity," in *Proc. of 12th ACM conference on Computer and Communications Security*, vol. 13, pp. 340-353, November 7-11, 2005. Article (CrossRef Link)

[8] Miguel Castro, Manuel Costa and Tim Harris, "Securing software by enforcing data-flow integrity," in *Proc. of 7th Symposium on Operating Systems Design and Implementation*, pp. 147-160, November 6-8, 2006. Article (CrossRef Link)

[9] Intel 64 and IA-32 architectures software developer's manual, December, 2017. Article (CrossRef Link)

[10] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proc. of 24th USENIX Security Symposium*, pp. 161-176, August 12-14, 2015. Article (CrossRef Link)

[11] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi and Stelios Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proc. of 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 901-913, October 12-16, 2015. Article (CrossRef Link)

[12] Arati Baliga, Vinod Ganapathy and Liviu Iftode, "Detecting kernel-level rootkits using data structure invariants," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 5, pp. 670-684, September/October, 2011. Article (CrossRef Link)

[13] Hong Hu, Z. L. Chua, Sendroiu Adrian, Prateek Saxena and Zhenkai Liang, "Automatic Generation of Data-Oriented Exploits," in *Proc. of 24th USENIX Security Symposium*, pp. 177-192, August 12-14, 2015. Article (CrossRef Link)
[14] Hong Hu, Shweta Shinde, Sendroiu Adrian, Z. L. Chua, Prateek Saxena and Zhenkai Liang, "Data oriented programming: On the expressiveness of non-control data attacks," in *Proc. of 2016 IEEE Symposium on Security and Privacy*, pp. 969-986, May 22-26, 2016. Article (CrossRef Link)
[15] G. C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak and Westley Weimer, "CCured: type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no.3, pp. 477-526, March, 2005. Article (CrossRef Link)
[16] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney and Yanling Wang, "Cyclone: A Safe Dialect of C," in *Proc. of 2002 USENIX Annual Technical Conference*, pp. 275-288, June 10-15, 2002. Article (CrossRef Link)
[17] Santosh Nagarakatte, Jianzhou Zhao, M. Martin and Steve Zdancewic, "SoftBound: highly compatible and complete spatial memory safety for C," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245-258, June, 2009. Article (CrossRef Link)
[18] Santosh Nagarakatte, Jianzhou Zhao, M. Martin and Steve Zdancewic, "CETS: compiler enforced temporal safety for C," *ACM Sigplan Notices*, vol. 45, no. 8, pp. 31-40, August, 2010. Article (CrossRef Link)
[19] S. A. Carr and Mathias Payer, "DataShield: Configurable Data Confidentiality and Integrity," in *Proc. of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 193-204, April 2-6, 2017. Article (CrossRef Link)
[20] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihar Budiu and G. C. Necula, "XFI: Software guards for system address spaces," in *Proc. of 7th Symposium on Operating Systems Design and Implementation*, pp. 75-88, November 6-8, 2006. Article (CrossRef Link)
[21] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee and Brad Chen, "Adapting Software Fault Isolation to Contemporary CPU Architectures," in *Proc. of 19th USENIX Security Symposium*, pp. 1-12, August 11-13, 2010. Article (CrossRef Link)
[22] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar and Dawn Song, "Code-Pointer Integrity," in *Proc. of 11th Symposium on Operating Systems Design and Implementation*, pp. 147-163, October 6-8, 2014. Article (CrossRef Link)
[23] Yajin Zhou, Xiaoguang Wang, Yue Chen and Zhi Wang, "Armlock: Hardware-based fault isolation for arm," in *Proc. of 21st ACM SIGSAC Conference on Computer and Communications Security*, pp. 558-569, November 3-7, 2014. Article (CrossRef Link)
[24] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee and Yunheung Paek, "HDFI: hardware-assisted data-flow isolation," in *Proc. of 2016 IEEE Symposium on Security and Privacy*, pp. 1-17, May 22-26, 2016. Article (CrossRef Link)
[25] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard and Hamed Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *Proc. of 2015 IEEE Symposium on Security and Privacy*, pp. 781-796, May 17-21, 2015. Article (CrossRef Link)
[26] Mauro Conti, Stephen Cranez, Lucas Daviy, Michael Franzz, Per Larsenz, Christopher Liebcheny, Marco Negroy, Mohaned Qunaibitz and Ahmad-Reza Sadeghiy, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proc. of 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 952-963, October 12-16, 2015. Article (CrossRef Link)
[27] G. E. Suh, J. W. Lee, David Zhang and Srinivas Devadas, "Secure program execution via dynamic information flow tracking," *Acm Sigplan Notices*, vol. 39, no. 11, pp. 85-96, November, 2004. Article (CrossRef Link)
[28] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk and R. K. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Proc. of 2005 International Conference on Dependable Systems and Networks*, pp. 378-387, June 28-July 1, 2005. Article (CrossRef Link)

[29] Jingfei Kong, C. C. Zou and Huiyang Zhou, "Improving software security via runtime instruction-level taint checking," in *Proc. of 1st ACM workshop on Architectural and System Support for Improving Software Dependability*, pp. 18-24, October 21, 2006. Article (CrossRef Link)
[30] Sandeep Bhatkar and R. Sekar, "Data Space Randomization," in *Proc. of 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 1-22, July 10-11, 2008. Article (CrossRef Link)
[31] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund and Thomas Walter, "Breaking the memory secrecy assumption," in *Proc. of 2nd European Workshop on System Security*, pp. 1-8, March 31, 2009. Article (CrossRef Link)
[32] Jonathan-Christofer Demay, Eric Totel and Frédéric Tronel, "SIDAN: A tool dedicated to software instrumentation for detecting attacks on non-control-data," in *Proc. of 4th International Conference on Risks and Security of Internet and Systems*, pp. 51-58, October 19-22, 2009. Article (CrossRef Link)
[33] Gildo Torres and Chen Liu, "Can Data-Only Exploits be Detected at Runtime Using Hardware Events?: A Case Study of the Heartbleed Vulnerability," in *Proc. of the Hardware and Architectural Support for Security and Privacy*, June 18, 2016. Article (CrossRef Link)
[34] Ben Niu and Gang Tan, "Modular control-flow integrity," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 577-587, June, 2014. Article (CrossRef Link)
[35] Chris Lattner and Vikram Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proc. of 2nd IEEE/ACM international symposium on Code generation and optimization*, pp. 75-88, March 20-24, 2004. Article (CrossRef Link)
[36] webbench-1.5. Article (CrossRef Link)
[37] Null HTTPd Remote Heap Overflow Vulnerability. Article (CrossRef Link)
[38] SSH CRC-32 Compensation Attack Detector Vulnerability. Article (CrossRef Link)

Guimin Zhang received his B.S. degree from Shandong University in 2011 and received his M.S. degree from Zhengzhou Science and Technology Institute in 2014. Currently, he is a Ph.D candidate at State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include information security and trusted computing.

Qingbao Li is currently a professor at State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests includes information security, software protection theory and trusted computing.

Zhifeng Chen received his B.S. degree, M.S. degree and Ph.D from Zhengzhou Science and Technology Institute in 2009, 2012 and 2016 respectively. He is currently a lecturer at State Key Laboratory of Mathematical Engineering and Advanced Computing. His research interests include information security, software protection theory and trusted computing.

Ping Zhang is currently an associate professor at State Key Laboratory of Mathematical Engineering and Advanced Computing. Her research interests include information security and parallel compilation.