Gadget Weighted Tagging: A Flexible Framework to Protect Against Code Reuse Attacks

Liwei Chen, Mengyu Ma, Wenhao Zhang, Gang Shi, Dan Meng

Gadget Weighted Tagging: A Flexible Framework to Protect Against Code Reuse Attacks

Liwei Chen, Mengyu Ma, Wenhao Zhang, Gang Shi and Dan Meng

Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
University of Chinese Academy of Sciences, Beijing, China
{chenliwei, mamengyu, zhangwenhao, shigang, mengdan}@iie.ac.cn

Abstract. The code reuse attack (CRA) has become one of the most common attack methods. In this paper, we propose gadget weighted tagging (GWT), a flexible framework to protect against CRAs. In GWT, we firstly find all possible gadgets, which can be used in CRAs. Then, we attach weighted tags to these gadgets based on the lengths and types of the gadgets, and the weighted values are configurable. At last, GWT monitors the weighted tag information at runtime to detect and prevent CRAs. Furthermore, combining with the rule-based CFI, GWT+CFI can precisely confirm the gadget start and greatly reduce the number of possible gadgets, compared to the baseline GWT. We implement a hardware/software co-design framework to support GWT and GWT+CFI. The results show that the performance overheads of GWT and GWT+CFI are 2.31% and 3.55% respectively, and GWT can defeat variants of CRAs, especially those generated by automated tools.

1 Introduction

Code reuse attack (CRA) has become the primary attack vector nowadays. Attackers find special code snippets called gadgets, ending with an indirect branch instruction, and then manipulate program control flow to chain gadgets together to construct a malicious program. The CRA is proved to be Turing-complete and can be generated by automated tools [1]. Therefore, the CRA is an attack approach which is easy-to-use, hard-to-detect and can be applied to any purpose.

Many defense mechanisms have been proposed to protect against CRAs, and control flow integrity (CFI) [3][15] is thought to be one of the most promising ways. However, CFI has high complexity and low efficiency [4]. In order to design a practicable CFI, some researches [10][11][6][5] propose the coarse-grained CFI to realize a looser notion of control flow integrity without the static control-flow graph (CFG). Abandoning CFG to reduce complexity, the coarse-grained CFI is a tradeoff between security and efficiency. Hence, the security of the coarse-grained CFI is lower than the CFI based on CFG. Some recent papers [12][13][8][14] have proved that the coarse-grained CFI could be bypassed by CRAs with special gadgets.

We observe that all possible gadgets, which can be used in CRAs, only take a part of the whole code space. As shown in Figure 1, gadgets in normal programs are discontinuous usually, whereas the CRA is composed of a **contiguous** sequence of gadgets.

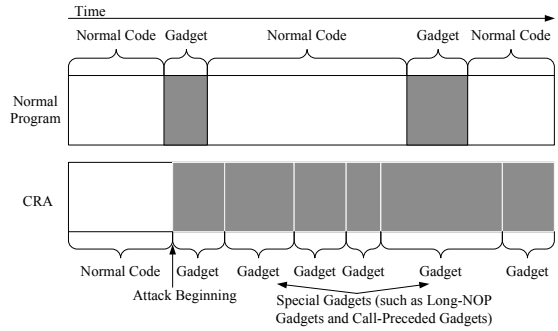

Fig. 1. The Gadgets in Normal Program and CRA

Additionally, with some special gadgets inserted, such as long-NOP gadgets and call-preceded gadgets [14][7], CRAs may pretend to be normal programs. Therefore, we can distinguish CRAs from normal programs by monitoring both normal gadgets and special gadgets at runtime.

In this paper, we propose gadget weighted tagging (GWT), a flexible framework to detect and prevent CRAs. In GWT, we classify all possible gadgets into three major types on the basis of their effects on CRAs: functional gadgets, NOP gadgets and normal codes. Then, we configure different weighted scores to these gadget types, and attach weighted tags to the gadgets in binary files. Finally, GWT computes the probability of CRA occurrence by dynamically monitoring the weighted tags at runtime.

Moreover, we propose GWT+CFI, since we discover that the control-flow rules in coarse-grained CFI are good supplements to the GWT. GWT+CFI can accurately identify the gadget start, and significantly reduce the number of potential gadgets. Hence, it is more precise to find possible gadgets in GWT+CFI, compared to the baseline GWT.

Several important parameters used in GWT, such as weighted scores for different gadget types, are configurable. Thus, users can modify the parameters as needed. Furthermore, if users discover some special gadgets, they can add the new gadget types into the GWT, and allocate different weighted scores for the gadget types. As a result, GWT can detect variants of CRAs with different gadgets, and the security of GWT can be further improved with adding more gadget types.

When a new CRA attack method or a new gadget type is proposed, the ways to construct the attack or to find the gadget should also be introduced. Hence, in GWT, we can reuse the ways to find the potential gadgets in programs, according to the descriptions in previous papers. Additionally, we can find and mark possible gadgets in GWT by reusing automated tools for CRA generation [1].

We apply GWT and GWT+CFI in a hardware/software co-design framework. The results show that the performance overheads of GWT and GWT+CFI are 2.31% and 3.55% respectively. Moreover, GWT can detect all CRAs generated by Q [1], and variants of CRAs with special gadgets.

The key contributions of this paper are summarized as follows:

1) We propose gadget weighted tagging (GWT), a novel framework to prevent CRAs, which is of high-flexibility, high-security, low-overhead and easy to apply.

2) We propose GWT+CFI, GWT combining with the rule-based CFI, which can find possible gadgets more precisely, compared to the baseline GWT.

3) We implement a hardware/software co-design to support GWT and GWT+CFI, and evaluate its security effectiveness and performance overhead.

2 Background and Related Work

2.1 Code Reuse Attacks (CRAs)

The basic idea of CRAs is to reuse instructions from the existing code space to implement malicious operations. In order to achieve attack purposes, attackers should firstly find a set of instruction sequences (called gadgets) from the entire code space, and then link the selected gadgets into a gadget chain to construct a malicious program. In short, the CRA is composed of a contiguous sequence of gadgets.

A gadget usually has several normal instructions for computation and an indirect branch instruction to change control flow to link gadgets. For simplicity, we focus on three most common indirect branch instructions in this paper: call, return and indirect-jump. Furthermore, the gadgets used in CRAs have two important features as follows:

1) Sparse Distribution. Every gadget should end with an indirect branch instruction, however, the indirect branch instructions only take a small part of the whole program. Moreover, attackers need gadgets containing special operations to construct a malicious program, or to bypass defense mechanisms, such as CFI. Thus, many existing gadgets may not meet for the special needs of attackers. As a result, the gadgets, which are useful for CRA construction, are quite rare in normal programs.

2) Small Size. The gadgets with more instructions can perform more operations, however, they also inevitably lead to more side effects and some of them may conflict with each other. Hence, attackers usually prefer to discover the short and simple gadgets only with the intended operations [1], instead of using long and complex gadgets. In fact, the gadgets in real CRAs usually have only 2 to 6 instructions [6].

According to the difference of the indirect branch instructions, CRAs can be classified into return-based ROP (return-oriented programming) and jump-based JOP (jump-oriented programming). In ROP, the stack pointer *esp* is used as the program counter in a return-oriented program. Instead, JOP uses a dispatch table to manage gadget addresses. The program counter of JOP is the register pointing into the dispatcher table [2], and an special **dispatcher gadget** is used to drive control flow. Thus, the dispatcher gadget is a key sign of JOP, which usually contains an self-modification operation and an indirect jump instruction. Generally, any gadget ending with an indirect jump instruction, that carries out the following algorithm, can be considered as a candidate of dispatcher gadget [2].

pc $\leftarrow$ *f(pc);*
*goto *pc;*

2.2 Control Flow Integrity (CFI)

There are many CFI implementations that have been proposed [3][4][15][10][6][5]. We divide these CFI methods into two main kinds: CFG-based CFI and rule-based CFI.

CFG-based CFI, also is known as fine-grained CFI, enforces the control flow to adhere to control-flow graph (CFG) generated by static program analysis. CFG-based CFI is high security, and can detect any illegal control flow transfers. However, it is almost impossible to generate an ideal CFG containing all possible control-flow paths, and the performance overhead of CFG-based CFI is high [4]. Furthermore, some recent works [16][17] have proposed CRA attack methods to bypass the CFG-based CFI with unideal CFG. In addition, even the ideal CFG-based CFI could be also bypassed by control flow bending [18] combining with the non-control-data attack.

Rule-based CFI, that is also called coarse-grained CFI, uses several control-flow rules to defend against CRAs, instead of CFG. Therefore, the rule-based CFI is low-overhead and easy to implement. The control flow rules can be classified into two types as following.

1) Control-flow transfer rules. These rules stipulate the destinations of different indirect branch instructions [11][9].

1.1) RETURN: a return instruction should point to an instruction right after a call instruction.

1.2) CALL: a call instruction should start execution at the entry point of a function.

1.3) Indirect-JUMP: an indirect-jump instruction should point to either a position inside the same function, or an entry point of another function.

2) Code length rules. These rules stipulate the code length of gadgets, and the chain length of CRA attacks [5][6].

2.1) Gadget length: a short code snippet (e.g. 2-6 instructions) ending with an indirect branch instruction is a gadget, whereas a longer code snippet is not a gadget.

2.2) CRA length: a long gadget chain (e.g. 6 gadgets) is a CRA, whereas a shorter gadget chain is not a CRA.

2.3 Legal Gadgets

The rule-based CFI has several definite rules to recognize gadgets. However, attackers can find some special gadgets that obey these rules, called legal gadgets in this paper. As a result, legal gadgets are mistaken for normal codes by the rule-based CFI, so that CRAs consisting of legal gadgets can bypass the rule-based CFI. There are several kinds of legal gadgets to bypass different control-flow rules.

1) Call-Preceded (CP) Gadget. A call-preceded instruction is any instruction in the address space of the application that immediately follows a call instruction [14]. A call-preceded gadget is a gadget that its first instruction is a call-preceded instruction. Thus, CP gadgets are legal destinations of any *return* instruction.

2) Entry-Point (EP) Gadget. An entry-point gadget is a gadget starting at the entry point of a function. Hence, EP gadgets are legal destinations of any *call* or *indirect-jump* instruction.

3) Long-NOP gadget. A long-NOP gadget [14][7] is a gadget that contains enough instructions to obey the gadget length rule of rule-based CFI, and does not induce any

side effects, i.e., the content of all registers and memory area used by the CRA is preserved. Therefore, attackers can use short and useful gadgets to perform malicious operations, and use long-NOP gadgets to bypass the rule-based CFI.

3 Threat Model

In this paper, we focus on defending against code reuse attacks. We assume that attackers have full control over data and stack/heap memory regions. In addition, we also assume that the attackers can modify all key registers of processors.

We assume that the DEP technology has already been realized, thus it forces the attackers to use CRA attacks. Moreover, we do not consider the gadgets consisting of a sequence of unintended instructions, such as in x86 platform with variable instruction sizes, since they can be detected by CFI [9].

Furthermore, we will introduce our approach, GWT, in two different systems.

1) Baseline System. The baseline system does not have any special security mechanism, and is only protected by the DEP technology.

2) Protected System. The protected system supports the DEP technology and the rule-based CFI with control-flow transfer rules as described in Section 2.2.

4 Gadget Weighted Tagging

4.1 Finding Gadgets

1) Finding Gadget End.

The only strict limit of gadgets is ending with an indirect branch instruction (return, call and indirect-jump). Thus, all code snippets ending with an indirect branch instruction can be considered as gadgets in theory. Therefore, we should discover all indirect branch instructions in program codes, and take these instructions as the gadget end.

2) Defining Gadget Types.

Q [1], the state-of-the-art automated tool for CRA generation, uses functional kinds to identify gadgets. Q defines nine kinds of gadgets, including NoOp, Jump, MoveReg, LoadConst, Arithmetic, LoadMem, StoreMem, ArithmeticLoad, and ArithmeticStore [1]. In Q, every gadget should have and only have one kind to define its function, and Q proposes a discovery algorithm to recognize the functional kind of each gadget.

In GWT, we define four types of gadgets as shown in Figure 2. The functional gadget includes the eight kinds of gadgets defined in Q, except the NoOp gadget. The dispatcher gadget and the syscall gadget are two special types of functional gadgets. Because the two types of gadgets play critical roles in CRAs, we take them as separate types out from the normal functional gadgets.

2.1) Functional Gadget. The functional gadget is a gadget that can perform a fixed and useful operation without any side effect, such as arithmetic calculation, branching, or loading data from memory. We use a method similar as Q [1] to identify functional gadgets. Details of the algorithm to identify the functional gadgets can be found in [1]. However, the original algorithm of Q can only be used to discover return-based gadgets and construct ROP chains. Hence, we have added new features, such as discovering

indirect-jump instructions and syscall instructions, to find jump-based gadgets to construct JOP chains, and to discover syscall gadgets and legal gadgets in GWT.

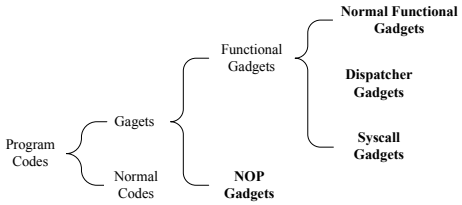

Fig. 2. The Gadget Types in GWT

2.2) Dispatcher Gadget. The dispatcher gadget is an special functional gadget, which ends with an indirect jump instruction and contains an self-modification operation to the jump destination. Furthermore, the dispatcher gadget plays a key role in the JOP [2]. It essentially maintains an virtual program counter and executes the JOP program by advancing it through one gadget after another.

2.3) Syscall Gadget. The syscall gadget is an special functional gadget ending with an system call instruction. To make CRA attacks simpler, attackers can carefully construct attacks that consist of a small number of gadgets, and then inject code. For example, attackers can use the syscall gadget to invoke an system call to alter the execute bit on an attacker-controlled buffer (i.e. destroy the DEP protection), and then redirect control flow to it [12].

2.4) NOP Gadget. The NOP gadget is a gadget which has neither useful operations nor side effects to CRAs. It includes the NoOp gadget in Q and long-NOP gadgets described in Section 2. To ensure that the NOP gadget does not break the semantics of the CRA chain, the NOP gadget should make use of only a small set of registers [14]. Generally, the NOP gadget can be used to confuse defense mechanisms, such as the rule-based CFI [6][5].

We define a parameter in GWT, MaxRegMod, which means the maximum number of registers modified by the NOP gadget. If a gadget is not a functional gadget and the registers modified by this gadget are not larger than MaxRegMod, then the gadget is a NOP gadget. Else, the gadget is not a NOP gadget (i.e. normal code).

3) Finding Gadget Start (Maximum Gadget Length).

It is simple to find the gadget end (i.e. indirect branch instructions), whereas to identify the gadget start is difficult. In previous papers, the code length is always used to recognize gadgets, since longer gadgets lead to more side effects and may conflict with other gadgets. Usually, the maximum length of a meaningful gadget is 6 instructions [6]. However, attackers can still find some special gadgets (e.g. long-NOP gadgets), which are long enough and have few side effects to CRAs, to break the limits.

In GWT, the classified gadget type still depends on code length, however, we add a new gadget type (i.e. NOP gadget) to mitigate the above problem. A long enough gadget, which has few side effects, could be recognized as a NOP gadget in GWT,

Table 1. Weighted Scores of Different Gadget Types

Gadget Type Value (3 bits)	Gadget Type	Weighted Score
000	Normal Code	zero clearing
001	Functional Gadget	1
010	Dispatcher Gadget	2
011	Syscall Gadget	4
100	NOP gadget	0
others	undefined	-

instead of normal code in previous papers. A gadget will be identified as normal code, only if the gadget contains too many instructions leading to significant side effects.

3.1) The Maximum Length of the Functional Gadget. From the gadget end of a functional gadget, walks backwards one by one (instruction), until the gadget is not a functional gadget. Then, the instruction number is the maximum length of the functional gadget, and the first instruction is the start of the functional gadget. Because the dispatcher gadget and the syscall gadget both belong to the functional gadget, the maximum lengths of the two types of gadgets are the same as the functional gadget.

3.2) The Maximum Length of the NOP Gadget. It is similar as the functional gadget. Increases the code length of a NOP gadget, until the gadget is recognized as normal code. Then, the code length is the maximum length of this NOP gadget.

4.2 Weighted Tagging

In GWT, we attach the weighted tag to each gadget end (i.e. indirect branch instruction). The structure of weighted tag is shown in Figure 3, and it has 32 bits and contains 3 parameters. The gadget type has 3 bits, and the meaning of this parameter is shown in Table 1. The maximum length of functional gadget has 14 bits, and the maximum length of NOP gadget has 15 bits.

We define five different gadget types, including normal code, functional gadget, dispatcher gadget, syscall gadget and NOP gadget. Normal code means the gadget can not be used in any CRA, and other four gadget types are introduced in Section 4.1. Different gadgets have different effects on CRAs, hence, we propose weighted scores to mark different gadget types. The foundation of weighted scores is the dangerousness and importance of each gadget type. With a higher score, the gadget type is more dangerous and important, and is also more possible to be used in CRAs.

The specific weighted score of each gadget type is shown in Table 1, and it is configurable. (i) The NOP gadget does not contain any useful operation for CRAs, and thus it has only 0 point. (ii) The functional gadget contains an useful and constant operation for CRAs, so it gets 1 point. (iii) The dispatcher gadget is a key sign of the JOP, and it is appeared every other gadget in the JOP, hence it has 2 points. (iv) System call is the start point of many important kernel functions of OS, and using the system call can significantly reduce the difficulty of CRA construction. Consequently, one common purpose of CRAs is to perform an special system call, and the syscall gadget is more

important than other gadget types, which has 4 points. (v) Since the CRA is composed of a contiguous sequence of gadgets, if the current gadget is normal code, there is no CRA at present. As a result, the weighted score of the normal code is zero clearing.

We add the gadget tag information into the program executable file by annotation. The prior of every indirect branch instruction is a weighted tag annotation. Moreover, tag annotation starts with a prefetch instruction to retain binary compatibility, which is the similar as other methods [3][9]. The prefetch instruction is followed by the weighted tag information. Because we assume that the system supports the DEP technology, the weighted tag annotation can not be modified by attackers.

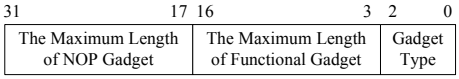

Fig. 3. Weighted Tag Information for Gadget End

4.3 Monitoring Gadget Tags

At runtime, we assume that codes between two indirect branch instructions construct a potential gadget, and we identify the type of the potential gadget by its code length and the weighted tag information attached to the second indirect branch instruction. Furthermore, we define four simple intermediate variables to calculate the probability of CRA occurrence: current gadget type (CGT), current gadget length (CGL), real gadget type (RGT) and CRA occurrence index (COI).

CGL is used to record the code length of the current potential gadget, and CGT is used to record the gadget type of the current gadget tag. When an instruction is executed, the CGL adds one to itself. When an indirect branch instruction is executed, it is the boundary between two potential gadgets. Therefore, CGT records the gadget type of the corresponding tag, while CGL records the current gadget length for RGT computation, and then starts a new count of code length for the next gadget.

RGT, which is computed based on the CGT and CGL, records the real type of the current gadget at runtime. The detailed algorithm process of RGT is described in Figure 4. The MaxFunc is the maximum length of the functional gadget, and the MaxNOP is the maximum length of the NOP gadget.

We use COI to denote the possibility of the CRA occurrence. If the COI is larger, then it is more likely CRA will occur. Furthermore, we define MaxCOI, the maximum value of COI. If the COI is larger than MaxCOI, then we assume that a CRA occurs. The COI is calculated based on RGT and weighted scores of gadget types, and the detailed algorithm process of COI is described in Figure 4.

4.4 Hardware Implementation

Figure 5 shows the hardware implementation for GWT. It contains three important hardware modules: control-flow monitor (CFM), weighted tag configuration (WTC) and

```
if (CGT == Normal Code) {
    RGT = Normal Code;
}
else if (CGT == NOP Gadget) {
    if (CGL <= MaxNOP) {
        RGT = NOP Gadget;
    }
    else {
        RGT = Normal Code;
    }
}
else { // CGT == Functional/Dispatcher/Syscall Gadget
    if (CGL <= MaxFunc) {
        RGT = CGT;
    }
    else if (CGL <= MaxNOP) {
        RGT = NOP Gadget;
    }
    else {
        RGT = Normal Code;
    }
}
```

```
COI = 0; // initialization
while (each indirect branch instruction) {
    if (COI <= MaxCOI) {
        if (RGT == Normal Code) {
            COI = 0;
        }
        else {
            COI = COI + Weighted Score of RGT;
        }
    }
    else { // COI > MaxCOI
        exit(); // a CRA is detected
    }
}
```

Fig. 4. The algorithm process of RGT and COI

CRA detecting module (CDM). In addition, we assume that the rule-based CFI has already been implemented.

The CFM is the main module for weighted information monitoring. It records the number of instructions executed (CGL) and gadget type (CGT) of current gadget at runtime. When an indirect branch instruction is executed, the CFM computes real gadget type (RGT) based on CGL and CGT, and then sends the RGT to the CDM. The CDM computes COI, and compare it to the MaxCOI. If the COI is larger than MaxCOI, then a CRA occurs, and thus CDM should send an exception to report an attack. The detailed computation algorithms of these parameters are described in Figure 4.

The WTC stores the parameters used by CFM and CDM, such as MaxCOI and weighted scores. Additionally, these parameters are configurable. Users can modify these parameters as needed. For example, if users want to improve system security, they can reduce the MaxCOI, or increase the weighted scores of gadgets. Furthermore, we assume that attackers can not directly change these parameters stored in WTC.

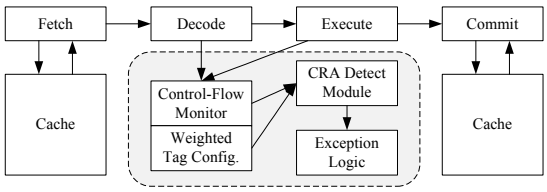

Fig. 5. Hardware Implementation of GWT

5 GWT Combining with CFI

5.1 Motivation of GWT+CFI

One main shortage of GWT is that GWT can not precisely confirm the gadget start. Only using the gadget length to identify the gadget start is not clear enough. Therefore,

we propose GWT+CFI, combining GWT and the rule-based CFI together. The rule-based CFI fits our approach very well, and introduces two benefits into the GWT.

1) Gadget Start. In order to bypass the rule-based CFI, attackers can only use some special gadgets, such as call-preceded (CP) gadgets and entry-point (EP) gadgets. Fortunately, the two types of gadgets both have fixed start points: call-preceded instruction and the function entry. Consequently, we can precisely identify the gadget start in GWT+CFI.

2) Gadget Number. Except legal gadgets, other illegal gadgets can be detected by the rule-based CFI. As a result, the number of gadgets that can be used in CRAs is reduced significantly in GWT+CFI. Therefore, we can locate useful gadgets in code space more accurately.

5.2 Finding Legal Gadgets

The major difference between GWT and GWT+CFI is the gadget discovery. In GWT+CFI, we only need to discover legal gadgets. On the other stages, such as gadget weighted tagging and monitoring, the GWT+CFI is almost the same as the GWT.

In GWT+CFI, we should find all indirect branch instructions firstly, and take them as the gadget end. Then, we should also find the gadget start. The start point of a CP gadget is a CP instruction, and the start point of an EP gadget is the function entry. Consequently, we can identify all legal gadgets based on the gadget start and end. At last, we should select useful gadgets from these legal gadgets. It is almost the same as the Section 4.1, to distinguish functional gadgets, dispatcher gadgets, syscall gadget, NOP gadgets and normal codes, as shown in Figure 6.

Note that one indirect jump instruction can jump to any position inside the same function, that does not violate the control-flow transfer rules of CFI. Hence, attackers may find legal gadgets without CP instruction or function entry by indirect jump instructions. Therefore, if we find that a function contains indirect jump instructions, the gadget discovery inside this function should follow the baseline GWT, instead of GWT+CFI.

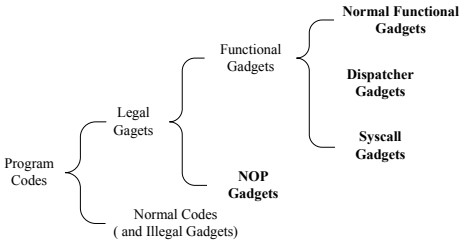

Fig. 6. The Gadget Types in GWT+CFI

6 Security Analysis

Because the hardware architecture of GWT is very simple, it is only used to record gadget information and compute COI. As a result, for simplicity, we implement a software method based on pintools [20] to simulate hardware functionality of GWT for security analysis. Pintools [20] are dynamic program analysis tools, which can manage the execution of every instruction. In addition, we select widely-used programs in Ubuntu */bin* and */usr/bin/*. In order to keep the balance between security and stability, we set the MaxRegMod is 6, MaxCOI is 6, and the weighted scores are the same as the Table 1.

Since the main difference between GWT and GWT+CFI is the gadget discovery, the CRA detection of GWT+CFI is almost the same as the GWT. Hence, we mainly focus on the security analysis of GWT in this section.

6.1 Gadget Discovery

In GWT, the average number of all possible gadgets is about 1072 in each program with average 40KB code size, and thus the gadgets take only a small part of the whole program codes. The average numbers of functional gadgets, dispatcher gadgets, syscall gadgets and NOP gadgets are 452, 7, 1.5 and 612 respectively. Furthermore, the average length of all possible gadgets is about 9.9 in each program. The average length of functional gadgets, including the dispatcher gadgets and syscall gadgets, is 5.3, and the average length of NOP gadgets is 13.3, as shown in Figure 7.

In GWT+CFI, the average number of legal gadgets is about 471, which decreases significantly compared to the baseline GWT, especially the legal functional gadgets. The average numbers of functional gadgets, dispatcher gadgets, syscall gadgets and NOP gadgets are 45, 0.9, 0.15 and 425 respectively. Conversely, the average length of legal gadgets is about 13.5, which is longer than the GWT. The average lengths of functional gadgets and NOP gadgets are 7.7 and 14.1, as shown in Figure 7.

MaxRegMod is the maximum number of registers modified, and defines the boundary between NOP gadgets and normal codes. Obviously, increasing the MaxRegMod will increase the maximum lengths of potential NOP gadgets, and include more normal codes into NOP gadgets. Thus, it may also increase the false positive rates of GWT. Figure 8 shows the changes of average lengths of NOP gadgets in GWT and GWT+CFI, with the MaxRegMod increasing. Although a processor has 32 registers, many registers have special purposes, and only 8 registers can be used for general-purpose computation usually. As a result, the average length of NOP gadgets has little change, after MaxRegMod is larger than 8. Additionally, because NOP gadgets have few effects on CRAs, and the weighted score of NOP gadget is zero, appropriately increasing MaxRegMod will not affect the CRA detection rate of GWT usually.

6.2 Practical Attacks

We test GWT against CRAs generated by Q [1], which can automatically generate ROP payloads for given programs. Since we have already marked all possible gadgets in these programs, GWT can easily detect the attacks consisting of gadgets, and count the weighted scores of gadgets in the CRA chains. Moreover, since Q can only generate

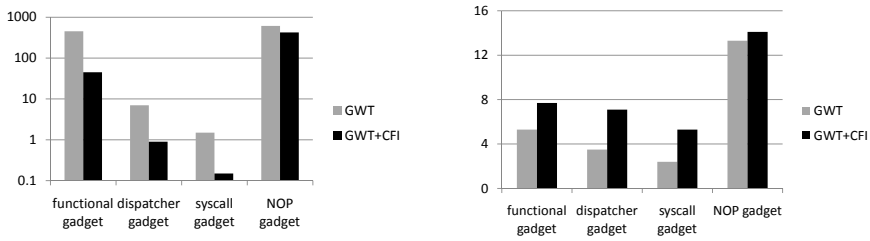

Fig. 7. The Average Numbers (left) and the Average Lengths (right) of Different Gadgets in GWT and GWT+CFI

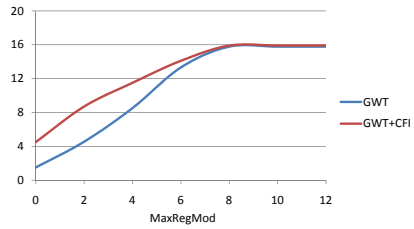

Fig. 8. The average lengths of NOP gadgets in GWT and GWT+CFI

return-based gadgets, the gadgets are normal functional gadgets in GWT, thus their weighted scores are 1 point. As a result, if the gadget number is larger than the MaxCOI, a CRA is detected. At last, the GWT can successfully detect 100% all the payloads generated from more than 100 applications under the directory */bin* and */usr/bin/*.

Then, we try to build some CRAs manually to bypass GWT. A practical way is to structure a short and simple gadget chain to close DEP, and then to inject malicious codes [12]. At last, we build a few CRAs to close DEP in a well-designed targeted program, which contain only a syscall gadget, a functional gadget and several NOP gadgets. The COI of these CRAs is 5, which is smaller than MaxCOI (6). However, it is very difficult to construct practical attacks to bypass GWT, since we can not find such satisfactory gadgets in real-world programs.

The weighted score of gadget types and MaxCOI are two critical parameters for CRA detection. Larger weighted scores and smaller MaxCOI can offer higher security, but may also increase the false positive ratio. Thus, we should define proper values to balance the security and stability. Figure 9 demonstrates the CRA detection rate and false positive rate of GWT with different MaxCOI. With the MaxCOI increasing, GWT needs more gadgets to identify the CRAs. CRAs generated by Q usually consist of 20-40 gadgets, and thus to set MaxCOI less than 20 can detect most CRAs generated by Q. On the other hand, with the MaxCOI decreasing, GWT may mistake some normal programs for CRAs. For example, a normal program invokes a system call, and this system call may be recognize as a syscall gadget. If we set MaxCOI less than 4, GWT will mistake the system call invoked by this normal program for a CRA.

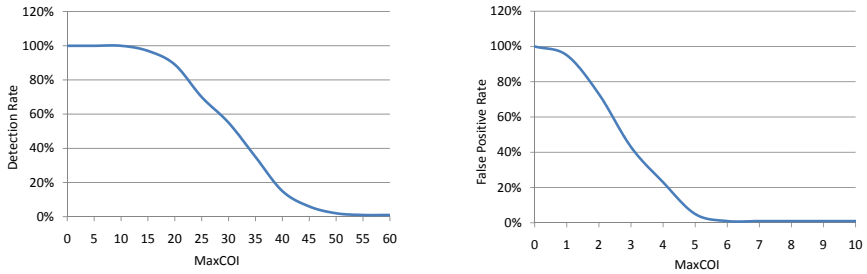

Fig. 9. The CRA Detection Rate (left) and The False Positive Rate (right) of GWT

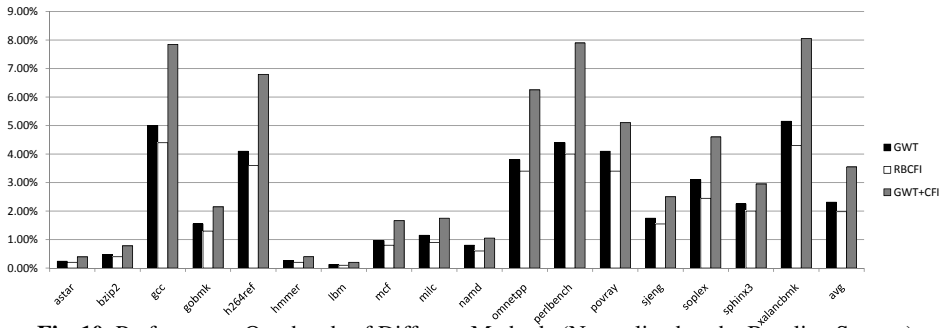

Fig. 10. Performance Overheads of Different Methods (Normalized to the Baseline System)

7 Performance Analysis

In this section, we implement hardware architecture of GWT and GWT+CFI in RTL (Register Transfer Level) using Verilog on a Loongson processor. Moreover, we select workloads from the SPEC CPU2006 benchmark [19], and these benchmarks are compiled using the GNU version of GCC at O3 optimization level.

We compare the performance overheads of GWT, RBCFI (rule-based CFI) and GWT+CFI on the target applications and the system. Note that we only perform RBCFI with three control-flow rules as described in Section 3 (i.e. protected system). We perform assembly-level instrumentation of the binaries for both GWT and RBCFI to insert the additional information needed to perform the checks for both methods.

As shown in Figure 10, the performance overhead of GWT is just about 2.31% on the average and it is less than 6% for all benchmarks. Furthermore, the performance overhead of GWT+CFI is about 3.55% on the average and it is less than 9% for all benchmarks, which includes the overhead of GWT as well as the overhead of rule-based CFI.

8 Conclusions and Future Work

In order to defend against CRAs, we propose GWT, a flexible, low-overhead and high-security framework. GWT discovers and marks all potential gadgets with static program

analysis, and then dynamically monitors the weighted gadget tags at runtime to calculate the probability of CRA occurrence. Furthermore, we propose GWT+CFI, GWT combining with the rule-based CFI, can more precisely find possible gadgets in code space, compared to the baseline GWT. We implement GWT and GWT+CFI in a hardware/software co-design system. The results show that the performance overheads of GWT and GWT+CFI are 2.31% and 3.55% respectively, and GWT can defeat variants of CRAs with different gadgets, especially those generated by automated tools. Moreover, GWT can be further optimized by adding more special gadget types and better gadget type classification.

The identification of the functional gadget, the NOP gadget and the normal code in GWT is not perfect. Attackers may manually find some useful gadgets which are identified as NOP gadgets in GWT, or find some NOP gadgets which are recognized as normal codes in GWT. Therefore, we plan to propose more precise methods to distinguish the gadget types. Additionally, it is our future work to insert discovery algorithms of special gadgets to improve the security of GWT, such as function-based gadgets.

9 Acknowledgement

This work is partially supported by the National Natural Science Foundation of China (No. 61602469).

References

1. Schwartz, Edward J. and Avgerinos, Thanassis and Brumley, David: Q: Exploit Hardening Made Easy. Proceedings of the 20th USENIX Conference on Security (SEC). 25–40 (2011)
2. Bletsch, Tyler and Jiang, Xuxian and Freeh, Vince W. and Liang, Zhenkai: Jump-oriented Programming: A New Class of Code-reuse Attack. Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS). 30–40 (2011)
3. Abadi, Martín and Budiu, Mihai and Erlingsson, Úlfar and Ligatti, Jay: Control-flow Integrity Principles, Implementations, and Applications. ACM Trans. Inf. Syst. Secur. 13, 4:1–4:40 (2009)
4. Burow, Nathan and Carr, Scott A. and Brunthaler, Stefan and Payer, Mathias and Nash, Joseph and Larsen, Per and Franz, Michael: Control-Flow Integrity: Precision, Security, and Performance. CoRR. abs/1602.04056 (2016)
5. Pappas, Vasilis and Polychronakis, Michalis and Keromytis, Angelos D.: Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. Proceedings of the 22Nd USENIX Conference on Security (SEC). 447–462 (2013)
6. Cheng, Yueqiang and Zhou, Zongwei and Miao, Yu and Ding, Xuhua and DENG, Huijie, Robert: ROPecker: A Generic and Practical Approach For Defending Against ROP Attack. Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS). (2014)
7. Kayaalp, Mehmet and Schmitt, Timothy and Nomani, Junaid and Ponomarev, Dmitry and Abu-Ghazaleh, Nael: SCRAP: Architecture for Signature-based Protection from Code Reuse Attacks. Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). 258–269 (2013)

8. Göktaş, Enes and Athanasopoulos, Elias and Polychronakis, Michalis and Bos, Herbert and Portokalidis, Georgios: Size Does Matter: Why Using Gadget-chain Length to Prevent Code-reuse Attacks is Hard. Proceedings of the 23rd USENIX Conference on Security Symposium (SEC). 417–432 (2014)
9. Kayaalp, M. and Ozsoy, M. and Abu-Ghazaleh, N. and Ponomarev, D.: Branch regulation: Low-overhead protection from code reuse attacks. Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA). 94-105 (2012)
10. Chao Zhang and Tao Wei and Zhaofeng Chen and Lei Duan and Szekeres, L. and McCamant, S. and Song, D. and Wei Zou: Practical Control Flow Integrity and Randomization for Binary Executables. Proceedings of the 34th IEEE Symposium on Security and Privacy (SP). 559-573 (2013)
11. Zhang, Mingwei and Sekar, R.: Control Flow Integrity for COTS Binaries. Proceedings of the 22Nd USENIX Conference on Security (SEC). 337–352 (2013)
12. Göktas, Enes and Athanasopoulos, Elias and Bos, Herbert and Portokalidis, Georgios: Out of Control: Overcoming Control-Flow Integrity. Proceedings of the 35th IEEE Symposium on Security and Privacy (SP). 575–589 (2014)
13. Carlini, Nicholas and Wagner, David: ROP is Still Dangerous: Breaking Modern Defenses. Proceedings of the 23rd USENIX Conference on Security Symposium (SEC). 385–399 (2014)
14. Davi, Lucas and Sadeghi, Ahmad-Reza and Lehmann, Daniel and Monrose, Fabian: Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection. Proceedings of the 23rd USENIX Conference on Security Symposium (SEC). 401–416 (2014)
15. Davi, L. and Koeberl, P. and Sadeghi, A.-R.: Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. Proceedings of the 51st Design Automation Conference (DAC). 1-6 (2014)
16. Evans, Isaac and Long, Fan and Otgonbaatar, Ulziibayar and Shrobe, Howard and Rinard, Martin and Okhravi, Hamed and Sidiroglou-Douskos, Stelios: Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS). 901–913 (2015)
17. Conti, Mauro and Crane, Stephen and Davi, Lucas and Franz, Michael and Larsen, Per and Negro, Marco and Liebchen, Christopher and Qunaibit, Mohaned and Sadeghi, Ahmad-Reza: Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS). 952–963 (2015)
18. Carlini, Nicholas and Barresi, Antonio and Payer, Mathias and Wagner, David and R. Gross, Thomas: Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. Proceedings of the 24th USENIX Conference on Security (SEC). (2015)
19. Henning, John L.: SPEC CPU2006 Benchmark Descriptions. SIGARCH Comput. Archit. News. 34, 1–17 (2006)
20. Luk, Chi-Keung and Cohn, Robert and Muth, Robert and Patil, Harish and Klauser, Artur and Lowney, Geoff and Wallace, Steven and Reddi, Vijay Janapa and Hazelwood, Kim: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 190–200 (2005)