Information Systems Frontiers (2021) 23:269–283
https://doi.org/10.1007/s10796-020-10014-7

HoneyGadget: A Deception Based Approach for Detecting Code Reuse Attacks

Xin Huang[1] · **Fei Yan**[1] · **Liqiang Zhang**[1] · **Kai Wang**[1]

Published online: 4 May 2020

Abstract

Code reuse attacks such as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP) are the prevalent attack techniques which reuse code snippets named gadget in vulnerable applications and hijack control flow to achieve malicious behaviors. Existing defense techniques for code reuse attacks attempt to prevent illegal control flow transition or make locating gadgets a hard work. However, decades of the arms race proved the ability to detect and prevent advanced attacks is still outdated. In this paper, we propose HoneyGadget, a deception based approach for detecting code reuse attacks. HoneyGadget works by inserting honey gadgets into the application as decoys and keep track of their addresses once the application is loaded. During the execution phase, HoneyGadget traces the execution records using Last Branch Record (LBR), compares the LBR records with the maintained address list, and alarms code reuse attacks if some records match. HoneyGadget not only prevents code reuse attacks, but also provides LBR records for researchers to analyze patterns of these attacks. We have developed a fully functioning prototype of HoneyGadget. Our evaluation results show that HoneyGadget can capture code reuse attacks effectively and only incurs a modest performance overhead.

Keywords Gadgets insertion · Deception · Control flow · Last Branch Record

1 Introduction

Individuals can perform many different behaviors to protect themselves from some computer security threats (Crossler et al. 2019), but it is still impractical to completely avoid the effects of security threats like control flow hijack attacks. Memory corruption vulnerability is one of the most prevalent attack vectors to hijack control flow of the program. Control flow hijack attacks can be divided into code injection attack and code reuse attack. With the widely deployment of Data Execution Prevention (DEP) (Andersen and Abella 2004) and $W \oplus X$, attackers are forced to reuse existing code snippets in binary. Triggered by a memory error, code reuse attack proved that it can perform arbitrary Turing-complete computation without injecting any malicious code (Checkoway et al. 2010). There are many disclosed CVEs rely on code reuse attack to trigger the execution of payload. On the other hand, several automated tools and methods can be used to help attackers to launch an attack (Salwan 2011; Schwartz et al. 2011). In addition, Automatic Exploit Generation (Avgerinos et al. 2014) has become a mature subject, introducing automation technology into the exploit.

With the development of code reuse attack, researchers have put forward many defenses. Representative defense techniques include randomization and control flow checking (Pappas et al. 2013, 2015). A general purpose of Address Space Layout Randomization (ASLR) is to make code snippets of target application and the exact memory address of the gadget unpredictable. Control Flow Integrity (CFI) introduced by Abadi et al. (2005) limit the transfer of control flow by constructing Control Flow Graph (CFG) statically and applying integrity checks during execution.

✉ Fei Yan
yanfei@whu.edu.cn

Xin Huang
huangxxxin@whu.edu.cn

Liqiang Zhang
zhanglq@whu.edu.cn

Kai Wang
blankaiwang@whu.edu.cn

[1] Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China

Despite decades of research effort, the ability to detect and prevent up-to-date attacks is still outdated since existing defense methods are either one-time effort or detecting malicious behavior according to pre-defined policies.

For a long time, cybersecurity has to face such a cruel reality that defenders are struggling to cope with countless vulnerabilities to ensure security, and adversaries only need to find one vulnerability to carry out the attack. The initiative of the cyber attack and defense game is in the hands of the attacker. To change the situation, it is essential to know which tactics attackers employ and what is happening in their minds when they are committing criminal activities (Silic and Lowry 2019). For example, attackers take advantage of the connections among people to deceive them to achieve the diffusion of deception in social media (Vishwanath 2015). The arms race based on code reuse will still continue. Using deception based techniques such as honeypots to capture these up-to-date code reuse attacks may help defenders take the initiative. Aimed to log malicious behavior of remote attackers, Honey-Patches (Araujo et al. 2014) patches the vulnerable web service with a specially designed function. Remote attackers send malicious HTTP requests to the server and get responses forged by the patched function, then the malicious behavior is captured by honeypot. Unfortunately, remote code execution does not ask for a malformed HTTP request. The patched function of Honey-Patches cannot help capturing remote code execution attempts.

In order to capture the pattern of these attacks or even some previously unknown code reuse attacks, we propose HoneyGadget, a deception based defense scheme just like Honey-Patches. HoneyGadget inserts honey gadgets as decoys to the target application and its related libraries, then we can detect code reuse attacks at runtime if inserted gadgets are executed. We have implemented a prototype of HoneyGadget on x86-based Linux platform. The experiment results show that HoneyGadget incurs a modest overhead of 6.8% on average. Compared to other defenses methods, the overhead brought by HoneyGadget is within an acceptable range.

In summary, our main contributions of this paper include:

1. We present HoneyGadget, a deception based approach for detecting all types of code reuse attacks, which is a brand-new method to defend against these attacks.
2. To the best of our knowledge, we design the first generic solution based on deception to defeat code reuse attacks and gain information about how attackers operate. That is, our scheme can help to analyze future evolutions of the attack.
3. We propose novel techniques combining constructing different types of gadgets, inserting gadgets automatically and runtime detection method to capture code reuse attacks without affecting the normal execution of the program.
4. We implement a prototype of HoneyGadget, and our evaluation shows that HoneyGadget achieves high efficiency with low overhead, proving our scheme practical.

The rest of this paper is organized as follows. We begin in Section 2 by introducing background knowledge on existing code reuse attacks and characteristics of the Intel's Last Branch Record (LBR). In Section 3, we detail our threat model and assumptions. The design of HoneyGadget and the concrete implementation are illustrated in Sections 4 and 5 . We evaluate our scheme in Section 6 and discuss its security features in Section 7. Related works are given in Section 8, and conclude in Section 9.

2 Background

In this section, we briefly summarize the techniques behind code reuse attack and introduce characteristics of the Intel's Last Branch Record (LBR).

2.1 Code Reuse Attacks

Code reuse attacks exploit a vulnerability to illegally transfer the control flow to an attacker-specified function or code snippet. These sequences of instructions named gadget that are already existing in the attacked program. Each sequence ends with an indirect branch instruction (such as the ret instruction or other indirect jumps with similar functions, call/jmp, etc) to transfer control from one sequence to the subsequent sequence, and finally use system calls to achieve the purpose of attack. According to the type of gadget exploited, code reuse attacks can be classified into Return-Oriented Programming (ROP) (Shacham 2007), Jump-Oriented Programming (JOP) (Bletsch et al. 2011) and Counterfeit Object-oriented Programming (COOP) (Schuster et al. 2015) and so on.

Return-Oriented Programming (ROP) is a typical code reuse attack. Gadgets chosen for gadget chain in ROP attack are usually short with no more than 6 instructions (Cheng et al. 2014) to avoid unplanned adjustment to pointers or registers. Each gadget in the attack payload is responsible for performing one or several steps of computation, such as loading argument from a specific register or performing arithmetic operations (Checkoway et al. 2010). Triggered by an inconspicuous vulnerability, stack buffer overflow for example, control flow of the target application is hijacked.

In ROP attack, the last instruction in each gadget must be a ret instruction to facilitate the continuous operation of gadgets. Jump-Oriented Programming (JOP)

uses branch instruction (jump) to replace the ret instruction, and can also construct gadgets with the same Turing-complete computing power (Bletsch et al. 2011). In order to orchestrate the orderly execution of each gadget, JOP attack requires a critical dispatcher gadget to change the control flow. JOP attacks ensure the continuous operation of gadgets by controlling EIP register. Because there is no need to change the ESP register to hijack the control flow, the addresses of the gadgets and their corresponding data in the JOP attack can be stored not only in the stack, but also in any rewritable data segment of the program.

Counterfeit Object-oriented Programming (COOP) is another novel code reuse attack technique which induces malicious program behavior by only invoking chains of existing C++ virtual functions in a program through corresponding existing call sites (Schuster et al. 2015). The attack strategy of COOP relies on Object Oriented Programming (OOP) principles and primarily aimed at C++ applications. In contrast to ROP and JOP attack, the control flow exploited by COOP attack is more similar to a benign execution flow and hard to be identificated. That means it is more difficult to defend against COOP attack based on control flow integrity.

Together with the deployment of defense schemes like ASLR and CFI in modern system, code reuse techniques update correspondingly (Carlini et al. 2015; Carlini and Wagner 2014; Göktas et al. 2014). In modern application situations, except from software on the local host, applications and services provided by remote servers become a growing trend (Riden et al. 2007; Araujo et al. 2014; Durumeric et al. 2014). Correspondingly, attacks on those remote hosts based on remote code execution and code reuse techniques appear. Based on the feature that servers do not rerandomize the address space layout after a crash under particular circumstances, Blind ROP (BROP) (Bittau et al. 2014) rewrites every single byte of stack canary after several attempts, and this corrupts stack integrity protection. The adversary then invokes write to dump more available gadgets in process memory. BROP enriches the arsenal of remote attackers and expand the attack surface of code reuse attacks.

2.2 Last Branch Record

Last Branch Record (LBR) provides a way to trace the execution control flow of a program, as it can log the executed branch information in a looped buffer at real-time, including the address of a branch instruction (from) and the target address (to). The most recent branch decisions recorded in LBR can be used to reconstruct the program's behavior. The advantage of using LBR is that the CPU can record branch information in parallel while execution, and it incurs no slowdown. For an Intel Skylake CPU, LBR can record the last 32 executed instructions. While the looped buffer of LBR is filled, the newly recorded branches overwrite the old ones (Guide 2011). The functionality of LBR is enabled/disabled by certain model-specific registers (MSRs). The access to MSRs requires kernel privilege, which makes the status of LBR transparent to programs running in user space.

3 Threat Model and Assumptions

HoneyGadget aims to detect and prevent all forms of code reuse attacks from both localhost and remote attackers. To ensure that our scheme is practical, we define our threat model based on strong yet realistic attack assumptions. With attack models in previous literature (Bittau et al. 2014; Carlini et al. 2015; Carlini and Wagner 2014; Göktas et al. 2014) and application scenarios of HoneyGadget, we generate the threat model as follows.

We assume that the target application is released and distributed without side information such as source code and debugging information. We further assume the application has at least buffer overflow vulnerability and the adversary has ready knowledge to exploit the vulnerability. The adversary is allowed to exploit the vulnerability repeatedly and can use automatic gadget generating tools to locate available gadgets and construct attack payload. We assume the adversary cannot get a higher privilege level through normal operations, for example, the adversary targeting a user space process cannot directly access the kernel privilege except invoking a system call maliciously. Specifically, we assume that the adversary cannot change the LBR records since they stored in hardware registers that cannot be written directly from user space.

For remote side, we assume servers restart their worker processes after a crash and do not change their address space layout. Currently, servers such as Nginx and Apache are compatible with this feature. We further assume that the adversary is allowed to overwrite a variable length of bytes including a return instruction pointer (Bittau et al. 2014). These assumptions mean that the adversary has a chance to mount BROP attack successfully.

We assume the operating system enables standard defense mechanisms such as $W \oplus X$ and ASLR by default. However, as HoneyGadget focuses on capturing the malicious behavior of adversaries, methods aim to stop unintended control flow transfer such as CFI are disabled.

4 Design

In this section, we describe the design of HoneyGadget. We first introduce the overview of our scheme, then we give out the detail of each component of HoneyGadget.

4.1 Overview

We design HoneyGadget as two main components: *static processing module* and *runtime checking module* (see Fig. 1). The static processing module is responsible for (1) source code iteration and locating places to insert honey gadgets as decoys; (2) generating different types of gadgets that meet the requirement of potential code reuse attacks and (3) gadgets insertion. After processed by the static processing module, the input file together with secured libraries are then taken over by runtime checking module. The runtime checking module of HoneyGadget (1) maintains address list of inserted gadgets and a pre-defined sensitive function list; (2) performs runtime monitoring of execution and (3) prevents any suspicious code reuse attacks and collect attack data. At last the output file runs as a service program can be accessed by local users and remote users. The output file has no interference on normal operations. However, those inserted honey gadgets are tempting but dangerous traps for attackers.

4.2 Static Processing Module

As we mentioned, the key idea of HoneyGadget is deception. Based on the observation of attack principle of code reuse attacks, we draw a conclusion that those attacks assemble gadgets into attack payload and hijack the control flow of victim program no matter attack tricks transform. Thus, we can insert honey gadgets that meet the requirement of code reuse attacks as decoys to lure attackers to launch attacks. The main function of this component is to insert different types of gadgets into the binary file. Obviously, the types of honey gadgets and the locations where the gadgets inserted are the main factors that affect the effectiveness of HoneyGadget.

4.2.1 Types of Honey Gadgets

Gadgets are existing instructions which can read & write memory, perform operations or shift execution flow (Shacham 2007). It has been proved that instruction sequence build by a set of gadgets is Turing-complete (Checkoway et al. 2010). HoneyGadget defines gadgets from two perspectives, which are the functionality of the gadget and its control transfer instruction.

According to the functionality of gadgets, they can be divided into four categories, which are memory read/write gadgets, arithmetic and logic operation gadgets, function call gadgets and system call gadgets. The memory read/write gadgets can read argument from memory and then saves it to a specific register. The arithmetic and logical operation gadgets are essential in most gadget chains. The arithmetic operation gadgets consist of four basic operators including add, subtraction, multiplication and division. The logical operation consists of another four basic operators including *and*, *or*, *not* and *xor*. The completness on functionality of computaion makes assurance for Turing-complete computational capability of gadgets.

Function call gadgets can perform arbitrary function call in linked library by manipulating return address and passed arguments. This is also the foundation of ret2libc attacks. Different from function call gadgets, system call gadgets do not rely on stack but registers to transfer arguments. Besides, system call gadgets save the system call number in *eax* register then it invokes *linux-gate.so.1* provided by kernel to perform the corresponding system call. After system call being invoked, the return address is also saved in *eax*. The adversary can perform arbitrary system call by manipulating the values in the registers which are responsible for saving system call number and passing arguments through read/write gadgets and arithmetic and operation gadgets before *lcall %gs:0x10* is executed.

According to the type of control flow transfer instruction, gadgets can be divided into ret-based gadgets, jmp-based gadgets and call-preceded gadgets (Schuster et al. 2015). These kinds of gadgets are able to coexist in a gadget

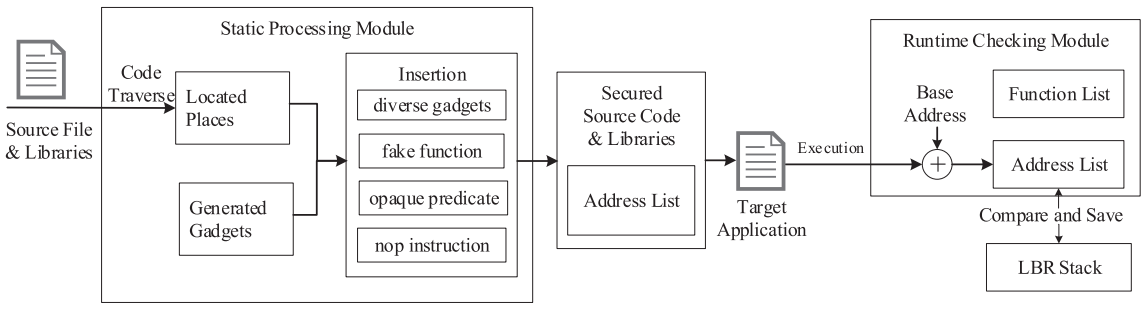

Fig. 1 Overview of HoneyGadget

chain. Ret-based gadgets and jmp-based gadgets differ in their ending instruction just as their names indicate. Call-preceded gadgets leverage instruction *call* and *ret* to manipulate control flow. Similar to an ordinary function call, call-preceded gadgets use *call* instruction to adjust the values in the registers. Meanwhile, *ret* instruction is responsible for transferring control flow to the following gadget.

4.2.2 Locations of Honey Gadgets

The locations to insert gadgets should be carefully arranged. Inserting gadgets inside normal instruction sequences may conflict with benign execution. For example, the gadget which modifies register *eax* may change the return address of benign execution flow. Consequently, the gadgets should be placed to unreachable execution paths. In general, We design three methods to insert honey gadgets to meet the requirements of not changing the program execution flow, which are summarized as function outlet, opaque predicate and fake function. The layout of code segment after inserting gadgets and *nop* instructions is shown in Fig. 2.

Function Outlet: The outlet of a function can be identified with the *ret* instruction and normal execution flow would never reach code snippets right after *ret* instruction. However, inserting honey gadgets right after *ret* will grant the function with multiple outlets. Automatic gadget generating tool such as ROPEME (Le 2010) and ROPgadget (Salwan 2011) will regard the second function outlet as a fake one and discard it. In order to separate honey gadgets from existing function outlets, the static processing module selects instruction *nop* to complete this task. Those inserted *nop* sequences form interspaces between original outlet and the inserted gadgets, and it confuses the automatic gadget generating tools.

Opaque Predicate: Opaque predicate have been used in software protection extensively. By setting predicate according to the value of invariant, context or execution result, opaque predicate is designed to clutter the control flow graph and it can redirect execution flow to a certain path. Suppose there is a basic block, block0, which is changed by designing the opaque predicate: if (always true condition) {block0;} else {junk code;} or if (always false condition) {junk code;} else {block0;}. Obviously the junk code will not be executed by the program, so we can insert our honey gadgets into it.

Fake Function: There is another method help us to introduce honey gadgets by inserting a seemingly normal fake function. After inserting the entire function into the binary file, since there is no other code to call this function, it will not be executed by the program. So it is completely feasible to insert the gadgets into the generated fake function without altering the program execution flow.

For each honey gadget inserted, the static processing module records the offset to the start of the binary file in a formulation of address list. The address list is then maintained by runtime checking module during executions.

4.2.3 Unintended Gadgets

Due to the poor alignment on x86 platform and the architecture actually allows instructions to be embedded within other instructions, there are several unintended gadgets

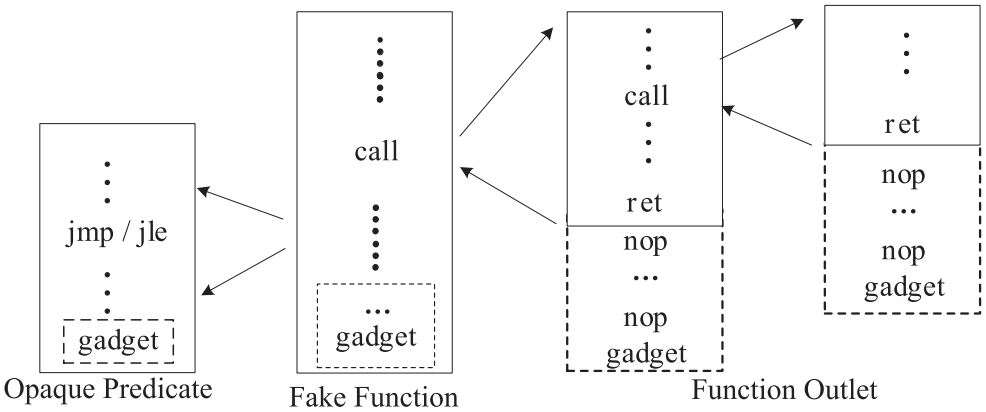

Fig. 2 Layout of code segment after inserting honey gadgets

enrich attackers' options on their way to construct gadget chains. In order to strengthen defense, HoneyGadget eliminate potential unintended gadgets by randomly inserting *nop* instructions (0x90) before each assembly instruction. Shown in Fig. 3, by inserting a *nop* sequence between instruction "*mov [ecx], edx*" and "*add ebx, ebx*", the unintended gadget disappears. We will introduce the detailed implementation of inserting *nop* instructions and gadgets in Section 5.

4.3 Runtime Checking Module

Runtime checking module is designed to check whether there are honey gadgets in execution branches of CPU. Since inserted gadgets are independent from existing code segments, and there is no legal control flow transferred to them, it is most likely triggered by malicious attackers once the inserted gadgets are executed. When loading application, ASLR randomizes the space layout of the application. Thus, in order to have an accurate record of inserted honey gadgets, the runtime checking module updates the saved address list with the mapping information of a particular application. This module adds the base address of code segments with offsets of each honey gadget when the application is loaded. This maintenance procedure is done in kernel space, which is also transparent to user level applications.

Checking every branch instruction of the application is inefficient. The timing of detection should be event driven and non-bypassable. Based on the observation that malicious executing code will eventually need to perform system calls to achieve something meaningful, the static processing module pre-defines a sensitive function call list and saves it in the kernel module together with the address list. The sensitive function list contains system calls that can elevate privilege or perform arbitrary execution such as *execve()* and *setreuid()*. While the target application is about to perform a sensitive function call, the runtime checking module pauses the execution of the target application and reads from the looped buffer of LBR. Then the runtime checking module compares the recorded instruction addresses with maintained address list. If one or more records match, HoneyGadget confirms a code reuse attack. Once this happens, runtime checking module immediately saves the LBR record and returns the system call with an error code.

5 Implementation

In this section, we describe the implementation of our prototype, and give the algorithms for gadgets insertion.

5.1 Honey Gadgets Insertion

In order to avoid potential altering of execution flow caused by our inserted gadgets, HoneyGadget inserts them to locations where benign control flow would never reach. Moreover, the diversification of unreachable execution path should be guaranteed to avoid those honey gadgets from identification. We implement three ways based on LLVM compiler to insert honey gadgets to meet the above requirements: code spaces right after function outlet, opaque predicate and fake function.

On the other hand, by means of randomly combining different operation instructions and control flow transfer instructions, HoneyGadget is able to generate all types of gadgets. This makes those honey gadgets inserted applicable for constructing a gadget chain. In general, the longer the length threshold of generated gadget, the richer the types of gadgets that can be generated, but at the same time the memory overhead will increase. In our previous work (Huang et al. 2019), we had ever set length threshold to 6 since a gadget is usually short with no more than 6 instructions in the real-world ROP attacks (Cheng et al. 2014). However, in other types of code reuse attacks, some special gadgets are more than 6 instructions. For example, in BROP attack, the BROP gadget contains 7 instructions. In order to balance detection efficacy and memory overhead, the length of generated honey gadget is no more than 8 instructions.

5.1.1 Function Outlet

As mentioned in Section 4, inserting gadgets directly after ret instruction will grant a function with multiple outlets. In this case, automatic gadget generating tools will recognize the inserted gadgets and discard them. In HoneyGadget, we insert *nop* instructions to space out the two outlets as a disguise. After analyzing several frequently used dynamic libraries including *glibc* and *ld*, we noticed that there are several *nop* instructions between basic blocks. The number of *nop* instruction is between 5 to 40. Thus, we disguise those inserted gadgets as normal code segments by inserting 5 to 40 *nop* instructions after *ret*.

The algorithm of honey gadgets insertion after function outlet is given in Algorithm 1. HoneyGadget randomly inserts *nop* instructions and gadgets after the ret instruction at the probability of *pGadget*. For each insertion location, static processing module generates a random number *pRand*. If *pRand* is less than *pGadget*, static processing module first inserts several *nop* instructions after the ret instruction, and then it randomly chooses a set of operation instructions such as *call*, *mov* or *sub* and an ending instruction to construct a gadget.

Algorithm 1 Honey gadgets insertion after function outlet.

Input:
(1) The list of functions, $FList$;
(2) The probability of insertion, $pGadget$;
(3) List of candidate operation instruction, $operationTypeTable$;
(4) List of ending instruction, $endTypeTable$.
Output:
The list with deception gadgets inserted, $DList$.

```
numOperationTypes ← operationTypeTable
numEndTypes ← endTypeTable
for F ∈ FList do
    pRand = random (0,1)
    if pRand < pGadget then
        i = the ret instruction of F
        numNOP = random (5,40)
        insertAfter(i, nop, numNOP)
        i ← i.next (numNOP)
        nOpt = random (1,7)
        for index from 1 to nOpt do
            optIndex = random (0,
            numOperationTypes)
            insertAfter(i, operationTypeTable
            [optIndex])
            i ← i.next
        end for
        endIndex = random (0, numEndTypes)
        insertAfter(i, endTypeTable [endIndex])
    end if
end for
return DList
```

5.1.2 Opaque Predicate

For opaque predicate, there exists three different types, which are invariant opaque predicate, contextual opaque predicate and dynamic opaque predicate (Ming et al. 2015). In HoneyGadget, we focus on using invariant opaque predicate due to its easy deployment, and it is the most frequently leveraged opaque predicate (Ming et al. 2015). There is two ways to insert honey gadgets through opaque predicate. The first method is to make use of some existing open source tools. We can use Obfuscator-LLVM (Junod et al. 2015) to process the source code and then use KLEE (Cadar et al. 2008) to locate the unreachable path. In HoneyGadget, we choose another method to insert gadgets. We implement an LLVM pass to create invariant opaque predicate during the code generation phase and then insert honey gadgets after that.

Algorithm 2 Honey gadgets insertion after opaque predicate.

Input:
(1) The list of basic blocks $BBList$;
(2) The probability of insertion, $pGadget$;
(3) List of candidate operation instruction, $operationTypeTable$;
(4) List of ending instruction, $endTypeTable$.
Output:
The list with deception gadgets inserted, $DList$.

```
numOperationTypes ← operationTypeTable
numEndTypes ← endTypeTable
for BB ∈ BBList do
    pRand = random (0,1)
    if pRand < pGadget then
        insert(opaque predicate)
        branch = unreachable execution branch
        nOpt = random (1,7)
        for index from 1 to nOpt do
            optIndex = random (0,
            numOperationTypes)
            insertAfter(branch, operationTypeTable
            [optIndex])
        end for
        endIndex = random (0, numEndTypes)
        insertAfter(branch, endTypeTable
        [endIndex])
    end if
end for
return DList
```

The algorithm of honey gadgets insertion after opaque predicate is given in Algorithm 2. It traverses each basic block, randomly inserts opaque predicate and then inserts honey gadgets on the unreachable path. The main difference is that inserting honey gadgets after opaque predicate does not require the insertion of extra nop instructions to space out the two outlets.

5.1.3 Fake Function

Since most of the current applications are written with thousands of lines of code, it is hard to find some extra dummy functions inserted in them. The function of inserting fake funtion is also implemented over LLVM compiler. We can randomly generate some functions that just look like normal functions through an LLVM pass. LLVM provides a lot of APIs for manipulating Intermediate Representation (IR), so it is convenient to use these interfaces to generate fake functions. We can use a set of C++ classes that provide

methods for constructing the corresponding functions, basic blocks and instructions.

When generating a fake function, we first construct a function using Function::Create(), and then randomly add basic blocks and instructions inside. The algorithm of fake function generation is given in Algorithm 3. In a fake function, we can embed multiple types of gadgets at the same time, such as code snippets ending with jmp or ret instructions. We can also insert some special gadgets to enhance the ability to detect specific code reuse attacks. Moreover, by inserting virtual functions into the binary file, we can even lure attackers to mount COOP attacks. In this scenario, we can capture all forms of code reuse attacks.

Algorithm 3 Fake function generation.

Input:
(1) The list of functions, $FList$;
(2) The probability of insertion, $pGadget$;
(3) List of candidate operation instruction, $operationTypeTable$;
(4) List of ending instruction, $endTypeTable$.

Output:
The list with deception gadgets inserted, $DList$.

```
numOperationTypes ← operationTypeTable
numEndTypes ← endTypeTable
for F ∈ FList do
    pRand = random (0,1)
    if pRand < pGadget then
        func = createFunction()
        numBB = random (1,5)
        for i from 1 to numBB do
            BBi = createBasicblock()
            nOpt = random (1,7)
            for index from 1 to nOpt do
                optIndex = random (0,
                numOperationTypes)
                insertAfter(BBi, operationTypeTable
                [optIndex])
            end for
            endIndex = random (0, numEndTypes)
            insertAfter(branch, endTypeTable
            [endIndex])
            insertAfter(func, BBi)
        end for
        insert(func)
    end if
end for
return DList
```

5.2 nop Insertion

As presented in Section 4, HoneyGadget randomizes code layout by randomly inserting *nop* before each instruction, this procedure can eliminate potential unintended gadgets to strengthen defense.

Similar with gadgets insertion procedure, during *nop* insertion procedure, static processing module traverses each instruction from the first line in source code. For each instruction traversed, the module generates a random number *pInsert*. If *pInsert* is less than *pNop* defined previously, static processing module inserts a *nop* ahead of the instruction.

5.3 Runtime Detection

We implement runtime checking module as a loadable kernel module leverages LBR to monitor execution states of instruction branches. The module inserts hooks in the system call table to intercept the risky system calls, and then invokes the checking algorithm to ensure that no honey gadgets have been executed. We leverage bit-vectors to store the information about inserted gadgets due to it is space and time efficient. Moreover, the maintained address list is in kernel space, this makes the address list transparent to adversaries and immune to information leakages in application layer.

HoneyGadget pre-defines a sensitive function list containing function calls that can elevate privilege or perform arbitrary execution. It will trigger runtime detection mechanism if one of the sensitive functions is called. It need to be noted that sensitive function list can be customized, in our prototype system, it include system calls *mprotect*(), *mmap*(), *execve*() and *setreuid*(). These system calls are very risky and are often used by attackers in the real-world code reuse attacks. While the detection mechanism is invoked, HoneyGadget pauses the execution of the target application. Then runtime checking module sends the privilege instruction *rdmsr* to kernel to read LBR buffer. After reading the 32 recorded instructions, the module checks whether there exist one or more recorded instructions match with items in the address list. Since only malicious execution flow can reach inserted gadgets, those addresses in the address list shall never appear in LBR records during normal execution. Once the matching address is detected, the checking module immediately saves the related LBR records and then returns the system call with an error code. The saved branch information can be used to analyze how attackers operate.

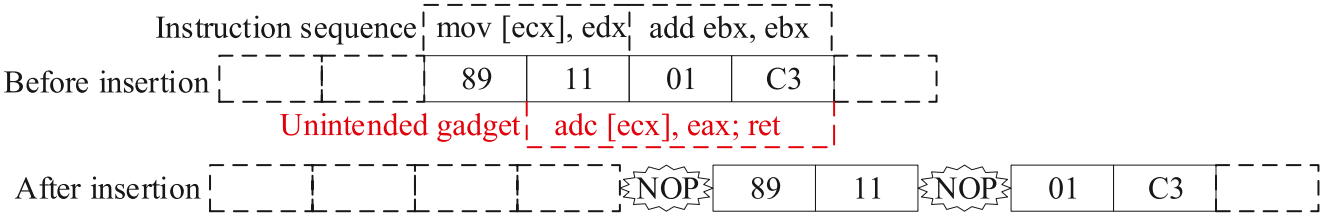

Fig. 3 The layout of instruction sequence after *nop* insertion

6 Evaluation

We implement HoneyGadget on Ubuntu 16.04 and the deployed LLVM and Clang version are both 3.5.2. The machine equips an Intel Skylake i5-6500 CPU with 8GB available memory. We evaluate the extra memory requirement of inserting *nop* instructions and gadgets, effectiveness and performance overhead of HoneyGadget.

6.1 Memory Cost Evaluation

When the application is loaded, those inserted gadgets and *nop* instructions are loaded into the memory along with the application. Consequently, the memory consumption of the target application inevitably increases. In our experiment, we evaluate the memory cost of *nop* insertions and gadgets insertion.

6.1.1 Evaluation of Gadgets Insertion

We use HoneyGadget to process common Linux applications and evaluate the memory cost of honey gadgets insertion. In this experiment, we set *pGadget* from 0.1 to 0.9 and test the memory cost of inserting honey gadgets into function outlet, opaque predicate and fake function by using the corresponding LLVM pass, respectively. As shown in Table 1, three insertion methods take 8.2%, 18.9%, 9.4% extra memory respectively when *pGagdet* is set to 0.5. We can see that memory cost of gadgets insertion is positively correlated with the insertion probability *pGadget*.

In addition, we set *pNop* and *pGadget* to 0.5, and use three LLVM passes to insert gadgets into the binary simultaneously, the result shows that the average increase in binary size is about 35%. The proportion of inserted gadgets is more than 60%.

6.1.2 Evaluation of nop Insertion

Similar to the above evaluation, We leverage same applications to evaluate the memory cost and the percentage of remained unintended gadgets after *nop* insertion. In this test, we set *pGadget* to 50% as benchmark. Figure 4 shows that it takes 1.2% extra memory space while *pNop* is set to 0.1, and 10.6% extra memory cost while *pNop* is set to 0.9. We can see that the extra memory requirement has a linear positive relationship with *nop* insertion probability. On the other hand, along with the increase of *pNop*, the possibility of corrupting an unintended gadget raises. The dashed line in Fig. 4 gives the percentage of remained unintended gadgets. We can see that the percentage of remained unintended gadgets drops to 3.4% when *pNop* is 0.9.

6.2 Effectiveness

In order to assess the effectiveness of our scheme, we verify HoneyGadget with different types of code reuse attacks under two real world vulnerabilities. During these tests, *pNop* and *pGadget* are both set to 50%. Results of these tests indicate that HoneyGadget can prevent code reuse attacks

Table 1 Memory cost of inserting honey gadgets

pGadget	Function Outlet(%)	Opaque Predicate(%)	Fake Function(%)
0.1	1.7	3.6	2.1
0.2	3.3	7.4	3.7
0.3	5.0	11.2	5.6
0.4	6.5	15.3	7.6
0.5	8.2	18.9	9.4
0.6	9.9	23.2	11.7
0.7	11.6	27.8	14.4
0.8	13.4	32.4	16.6
0.9	14.6	36.2	19.3

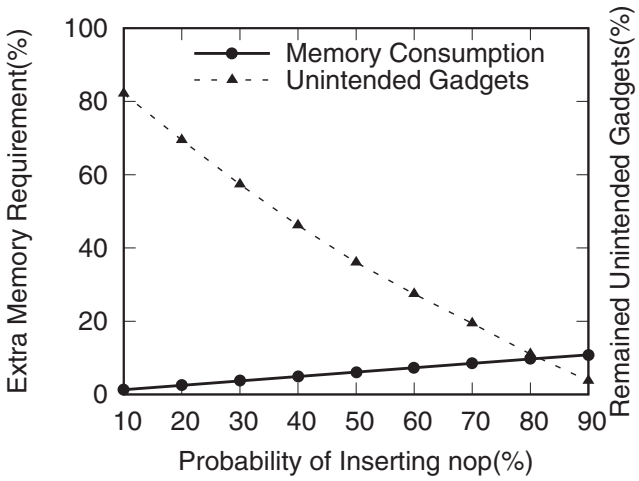

Fig. 4 Memory cost and effectiveness of *nop* insertion

effectively. As expected, our HoneyGadget can detect those attacks with no false positive.

ROP attack In the first test, we evaluate the effectiveness of HoneyGadget against ROP attacks. We first construct a small program that contains a stack buffer overflow vulnerability. By inputting long parameters, the vulnerability will be triggered and then be utilized to launch attacks. We use the automatic ROP gadget generating tool ROPGadget (Salwan 2011) to search available gadgets and randomly choose them to construct a ROP gadget chain. We repeat this test 50 times and detecte 49 out of them.

We also select No-IP Dynamic Update Client version 2.1.9 for testing. The application fails to perform a boundary check when calling the vulnerable function *strcpy*(). The exploit database Exploit-db provides an example of a ROP gadget chain. We replace the gadgets in the gadget chain with gadgets generated by automatic tool. We use same method to choose different gadgets to generate 50 gadget chains as attack payload to exploit. Among the 50 repeated tests, 48 of them used at least one of the inserted gadgets to construct the ROP gadget chain.

JOP attack Similar to the first test, we use the above two vulnerability programs to evaluate the effectiveness of HoneyGadget against JOP attacks. We use tool ROPGadget to find JOP gadgets and randomly choose them to construct JOP gadget chains. Finally, we generate 50 JOP gadget chains to attack that small program, HoneyGadget detect all of them due to payloads contain at least one inserted gadget. Then we repeat the JOP attack 50 times against No-IP Dynamic Update Client using the similar method. The result shows that HoneyGadget can detect 100% all the JOP attacks.

BROP attack In the last test, we assess the effectiveness of HoneyGadget against BROP attacks. Nginx web server is one of the most popular web servers in real world application situations. However, the weak security enforcement makes it vulnerable to a couple of attacks (Bittau et al. 2014; Evans et al. 2015). We exploit a simple stack vulnerability on Nginx 1.4.0 (64-bit) to initiate a BROP attack. BROP attack (1) uses stack reading to build memory leakage to defeat stack canaries and bypass ASLR; (2) finds a particular BROP gadget to locate function *call* and *write* in PLT, and gadgets to control arguments; (3) invokes those functions is enough to dump memory pages of the remote sever, and the dumped binary is transferred through network, that enables known exploits techniques such as the classic ROP. We apply HoneyGadget on Nginx server, and the scheme inserte honey gadgets that meets the requirement of BROP attack automatically. We repeat BROP attack attempt 50 times, all of them are detected since they leveraged at least one inserted gadget during stage 2 or 3 in attack payload.

6.3 Performance Overhead

To evaluate the overhead brought by HoneyGadget, we divide the evaluation into two phases. Corresponding to the architecture of HoneyGadget, the first phase is static processing, and another one is runtime checking.

We set *pGadget* and *pNop* to 50% and evaluate the overhead of static processing phase by adding *-time-passes* argument. During processing, the module recognizes all instructions, basic blocks and functions. Therefore, the larger the library size, the longer the static processing takes. Time for processing frequently-used libraries are shown in Fig. 5. As the results show, except from some huge libraries, it demands about 30 seconds to process a dynamic library.

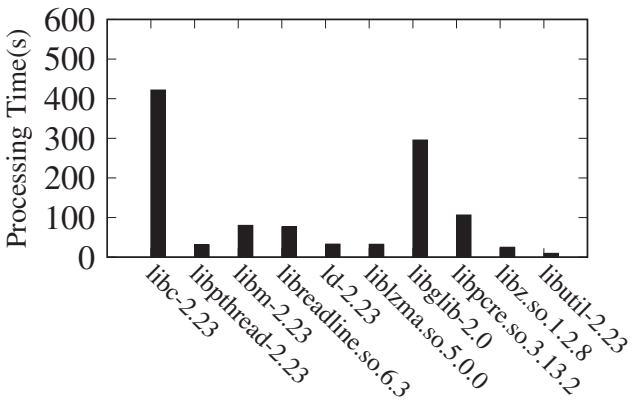

Fig. 5 Time consumption for processing different libraries

For example, it takes 32.9 seconds to process ld-2.23.so and 32.7 seconds to process liblzma.so. As for libraries with a huge quantity of basic blocks and functions, processing these libraries requires much time. Taking the library libc-2.23.so as an example, the time consumption increases to 421.6 seconds. Although it does take some time to do the static processing work, fortunately, operations in static processing phase is mostly a one-time effort, for libraries can be shared by different applications.

We evaluate the performance overhead of runtime checking module by using *phoronix test suite* (Larabel and Tippett 2011) which provides a number of benchmark tests for the Linux platform, and the results for each benchmark are presented in Table 2. Since inserted gadgets are unreachable for benign control flow, so in fact they don't introduce runtime overhead during execution. The main performance overhead incured by HoneyGadget is to compare LBR records and execute the inserted *nop* instructions. The results show that HoneyGadget introduces an average overhead of 6.8%, that is less than Readactor++ (8.4%) and other fine-grained CFI solutions. This indicates the overhead of HoneyGadget is within an acceptable range. In addition, insignificant runtime overhead prevents attackers from detecting the deployment of HoneyGadgte because the time difference between program executions is not so great. Even if nuances are noticed, attackers will most likely think that is caused by other factors, such as computer status or network latency. So it is not easy for attackers to detect the deployment of HoneyGadget through a differential method.

7 Discussion

7.1 Stack Pivot Attack

Stack pivot attack (Snow et al. 2013; Yan et al. 2016) manipulates the stack pointer to point to another memory region that stores attack payload. By applying indirect control flow manipulation, stack pivot attack can be utilized to bypass control flow integrity checks. Different from mechanisms using control flow integrity theories, the main idea of HoneyGadget is deception. By inserting gadgets as traps and *nop* instructions to eliminate potential unintended gadgets, HoneyGadget traps adversaries who tried to use inserted gadgets as attack payload. LBR can honstly record executed instruction branches and check with inserted gadgets. Thus, HoneyGadget can detect stack pivot attacks as long as inserted gadgets are used to construct the gadget chain.

7.2 Detection Rate

The theoretic detection rate of HoneyGadget depends on the number of hoeny gadgets and their types. We assume it takes *numGadgets* gadgets to construct an ROP payload,

Table 2 Runtime Overhead of HoneyGadget on the Phoronix Test Suite

Benchmark	Metric	Original	HoneyGadget	Overhead
SQLite	sec	71.98	77.95	8.3%
Tremulous	Frames/s	73.13	68.73	6.4%
HMMer	sec	46.39	48.29	4.1%
C-Ray	sec	112.40	114.09	1.5%
BYTE	Lines/s	9069220	7646897	18.6%
Parallel BZIP2	sec	45.89	47.45	3.4%
Smallpt	sec	621.46	674.28	8.5%
LZMA	sec	762.37	782.19	2.6%
Gcrypt	microsec	6142	6308	2.7%
dcraw	sec	157.44	168.62	7.1%
MP3 Encoding	sec	41.19	43.58	5.8%
PostgreSQL	Trans/s	278.97	248.64	12.2%
GnuPG	sec	21.07	22.61	7.3%
Average				6.8%

the number of original gadgets in source code is *numReal*, and the number of inserted gadgets is *numDec*. We can confirm code reuse attacks once one of the inserted gadgets is executed. The ideal detection rate of HoneyGadget comes to:

$$P = 1-(numReal/(numReal+numDec))^{numGadgets} \quad (1)$$

However, the function of gadgets varies in a gadget chain, and the frequency of their appearance differs. As a result, the appearance frequency of different types of gadgets is an important factor that influences the detection rate.

HoneyGadget requires a comprehensive set of inserted gadgets. We further assume the functionality of honey gadgets is consistent with original gadgets and they are of equal probability to be chosen to construct a ROP gadget chain. The total gadget type is n. For gadget type i, there are $numReal_i$ gadgets in the source code, and $numDec_i$ honey gadgets are inserted by HoneyGadget, the number of occurrences of gadget type i in a gadget chain is $numGadget_i$. We can conclude that:

$$numReal = \sum_{i=1}^{n} numReal_i \quad (2)$$

$$numDec = \sum_{i=1}^{n} numDec_i \quad (3)$$

$$numGadget = \sum_{i=1}^{n} numGadget_i \quad (4)$$

The theoretic detection rate of HoneyGadget while all gadgets are of equal possibility to be chosen to assemble a gadget chain can be concluded as:

$$P = 1-\prod_{i=1}^{n}(numReal_i/(numReal_i+numDec_i))^{numGadgets_i} \quad (5)$$

We can conduct the conclusion that the detection ratio of HoneyGadget has a *positive relationship* with the proportion of inserted honey gadgets. On the other hand, we can also infer that the detection ratio is closely related to the type of inserted gadgets. For example, as long as any type of inserted gadget has a ratio of more than 90% of the corresponding type of gadget, the theoretic detection rate of HoneyGadget can be greater than 90%. Therefore, by analyzing the pattern of code reuse attacks, we can insert specific gadgets to increase the detection rate. The attack data collected by LBR can help us to do this thing.

7.3 Special Gadget Chain

There are some kinds of code reuse attacks that relies on specific gadgets. For example, BROP (Bittau et al. 2014) relies on a special gadget named BROP gadget which is consisted of 7 instructions to pop contents stored in the stack. BROP gadget is critical for BROP attack since it can be used to control the first two arguments of calls. Since the number of BROP gadgets in the binary itself is relatively small, we increase the detection rate by inserting some BROP gadgets. Consequently, on launching a BROP attack, the adversary has a high probability of choosing inserted BROP gadget to construct attack payload. According to Section 6.2, detection on BROP comes to 100%. Similarly, because JOP attack requires a critical dispatcher gadget to orchestrate the orderly execution of each gadget, we can use the same method to increase the detection ratio.

7.4 Robustness

The detection effect of HoneyGadget relies heavily on isnerted gadgets. By inserting honey gadgets and eliminating unintended gadgets in source code, HoneyGadget induces the adversary to use inserted gadgets as attack payload. However, several originally exist gadgets are still available for adversary. Though the possibility of choosing those gadgets is diluted by inserting *nop* instructions, the possibility for adversary to pick out a gadget chain using original gadgets still exists. For such cases, the detection mechanism of HoneyGadget fails to react. Our approach is to analyze the pattern of code reuse attacks to maximize the success rate of detecting attacks. One possible solution to prevent such attacks is combining HoneyGadget with other techniques like G-Free (Onarlioglu et al. 2010) or binary rewriting to eliminate critical gadgets from original program code to prevent them from being misused by attackers.

We also have to consider that the adversary can try to bypass HoneyGadget by avoid reusing code snippets follows directly behind ret instruction consciously. Under this circumstance, some of inserted gadgets are filtered. Fortunately, HoneyGadget not only inserts gadgets after return of functions, but also unreachable branches of opaque predicate and fake functions. Locating opaque predicate or fake function requires for a reverse engineering and performing symbolic execution, which is not an easy task for attackers, especially in the case of large applications. As a result, the gadgets inserted into unreachable execution branches of opaque predicate and fake functions are still tempting traps for adversary. Inserting gadgets into branches of opaque predicate and fake functions just enriches code diversity. Thus, in this assumption, HoneyGadget is still able to detect potential code reuse attacks.

8 Related Work

To defend code reuse attacks, several defenses have been proposed. Address Space Layout Randomization (ASLR) is a representative mechanism by changing the space layout of the application and its related libraries. However, a single memory leakage vulnerability is enough to de-randomize the whole memory space. Enhancement to ASLR focus on applying fine granularity and re-randomization. For example, ASLP (Kil et al. 2006) randomizes the target application at the function level, Remix (Chen et al. 2016) randomizes the address space at basic block level, and ILR (Hiser et al. 2012) realizes randomization at instruction level. Bigelow et al. promoted a timely randomization scheme to re-randomizes address layout during execution (Le 2010). Although these fine-grained ASLR significantly increase the difficulty for attackers to locate gadgets, it also brings extra time and sapce consumption.

kGuard (Kemerlis et al. 2012) uses a *nop* sled to increase the difficulty for the attacker to obtain internal information of the program, but they only do this to protect and diversify the kernel. HoneyGadget uses deception for detecting code reuse attacks by randomly inserting gadgets to the target application and its related libraries. As mentioned in Section 5, the diversity of gadget types and inserted locations makes attackers hard to distinguish inserted gadgets from original ones. In order to further enhance the detection effect, HoneyGadget inserts *nop* instructions to eliminate unintended gadgets.

CFI poses an unacceptable overhead of more than 20%, that is too significant to deploy in commodity systems. In order to make CFI practical, a few coarse-grained mechanisms are proposed. CCFIR (Zhang et al. 2013a) and binCFI (Zhang and Sekar 2013b) relax the limitation of legal indirect control flow transfers and reduce overhead to an acceptable level. However, the loose checking policy can be bypassed by advanced attacks (Schuster et al. 2015). Another way to reduce overhead of checking the validity of control flow transfer is based on hardware. Liu et al. introduced a CFI enforcement using Intel Processor Trace (Liu et al. 2017). Kbouncer (Pappas 2012) and ROPecker (Cheng et al. 2014) use LBR to detect attacks. Compared to using IPT to trace the execution path, CPU is able to read LBR registers parallel at execution. This feature makes LBR a more efficient way to log instruction branches of the application. HoneyGadget also uses LBR to record execution branch of the target application. However, different from these approaches, the main idea of HoneyGadget is tempting adversaries to launch attacks by inserted gadgets, then capture and log the behavior of attacker.

Booby trap (Crane et al. 2013) is a mechanism to actively detect and respond to attacks against a target application proposed by Crane et al. The main idea of booby traps is as follows: in a diversified application, code sequences (the actual booby traps) are added that trigger an active response, such as terminating the program or generating an alert. Readactor++ (Crane et al. 2015) inserts booby traps in both PLT and vtables to mitigate blind probing of table entries. HoneyGadget inserts *nop* instructions and honey gadgets to lure adversary to launch attack. Compare to Readactor++, our HaneyGadget is more active and has a greater chance of getting attackers into the traps.

9 Conclusion and Future Work

In this paper, we present a deception based approach for detecting code reuse attacks named HoneyGadget. By inserting honey gadgets to the application, our HoneyGadget confuses adversary with traps. In case an attacker uses the inserted gadgets to mount attack, we can detect it and record his behavior. HoneyGadget maintains an address list recording addresses of inserted gadgets in kernel space and defines a set of sensitive function calls. Once executing the sensitive function call, HoneyGadget pauses execution of the target application and reads LBR buffer to check whether recorded instruction branches match with maintained addresses. If the record matches, HoneyGadget confirms a potential code reuse attack and saves the relevant information. Our evaluation shows that HoneyGadget incurs an acceptable runtime overhead of 6.8%. Compared to other defenses, the key idea of HoneyGadget is deception, which is a brand-new method to detect code reuse attacks.

The key for enhancing the efficacy of HoneyGadget is to enrich the type and location of inserted gadgets. Most gadgets in a ROP gadget chain are common and replaceable, however, some types of gadgets are rare but necessary for a certain kind of code reuse attacks. Consequently, limiting the types of honey gadgets to those critical gadgets will certainly increase the possibility for attacker to sink into our traps. On the other hand, it is necessary to figure out how the different types of honey gadgets and different placement strategies affect the effectiveness of HoneyGadget. How to reduce the system's false negative rate and how to prevent attackers from detecting the deployment of HoneyGadget also require further research. Future work could focus on systematizing the limitations and refining the type of inserted gadgets to get the better results.

Acknowledgements This work was supported in part by the National Natural Science Foundation of China under Grant No. 61272452 and the National Basic Research Program of China (973 Program) under Grant No. 2014CB340601.

References

Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J. (2005). Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (pp. 340–353): ACM.

Andersen, S., & Abella, V. (2004). *Data execution prevention*. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies.

Araujo, F., Hamlen, K. W., Biedermann, S., Katzenbeisser, S. (2014). From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security* (pp. 942–953): ACM.

Avgerinos, T., Sang, K. C., Rebert, A., Schwartz, E. J., Woo, M., Brumley, D. (2014). Automatic exploit generation. *Communications of the Acm*, *57*(2), 74–84.

Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D. (2014). Hacking blind. In *2014 IEEE Symposium on Security and privacy (SP)* (pp. 227–242): IEEE.

Bletsch, T., Jiang, X., Freeh, V. W., Liang, Z. (2011). Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (pp. 30–40): ACM.

Cadar, C., Dunbar, D., Engler, D. R., et al. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, (Vol. 8 pp. 209–224).

Carlini, N., & Wagner, D. (2014). Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium* (pp. 385–399).

Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R. (2015). Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium* (pp. 161–176).

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

Chen, Y., Wang, Z., Whalley, D., Lu, L. (2016). Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy* (pp. 50–61): ACM.

Cheng, Y., Zhou, Z., Miao, Y., Ding, X., Deng, H. (2014). Ropecker: A generic and practical approach for defending against rop attack. *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*.

Crane, S., Larsen, P., Brunthaler, S., Franz, M. (2013). Booby trapping software. In *Proceedings of the 2013 New Security Paradigms Workshop* (pp. 95–106): ACM.

Crane, S.J., Volckaert, S., Schuster, F., Liebchen, C., Larsen, P., Davi, L., Sadeghi, A.R., Holz, T., De Sutter, B., Franz, M. (2015). It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (pp. 243–255): ACM.

Crossler, R. E., Bélanger, F., Ormond, D. (2019). The quest for complete security: an empirical analysis of users' multi-layered protection from security threats. *Information Systems Frontiers*, *21*(2), 343–357.

Durumeric, Z., Bailey, M., Halderman, J.A. (2014). An internet-wide view of internet-wide scanning. In *USENIX Security Symposium* (pp 65–78).

Evans, I., Fingeret, S., Gonzalez, J., Otgonbaatar, U., Tang, T., Shrobe, H., Sidiroglou-Douskos, S., Rinard, M., Okhravi, H. (2015). Missing the point (er): on the effectiveness of code pointer integrity. In *2015 IEEE Symposium on Security and privacy (SP)* (pp. 781–796): IEEE.

Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G. (2014). Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and privacy (SP)* (pp. 575–589): IEEE.

Guide, P. (2011). Intel® 64 and ia-32 architectures software developer's manual. Volume 3B: System programming Guide, Part 2.

Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J. W. (2012). Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and privacy (SP)* (pp. 571–585): IEEE.

Huang, X., Yan, F., Zhang, L., Wang, K. (2019). Honeygadget: A deception based rop detection scheme. In *International Conference on Science of Cyber Security* (pp. 121–135): Springer.

Junod, P., Rinaldini, J., Wehrli, J., Michielin, J. (2015). Obfuscator-LLVM – software protection for the masses. In Wyseur, B. (Ed.) *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15* (pp. 3–9). Firenze: IEEE. https://doi.org/10.1109/SPRO.2015.10.

Kemerlis, V.P., Portokalidis, G., Keromytis, A.D. (2012). kguard: lightweight kernel protection against return-to-user attacks. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)* (pp. 459–474).

Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P. (2006). Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual* (pp. 339–348): IEEE.

Larabel, M., & Tippett, M. (2011). *Phoronix test suite*. Phoronix Media, [Online] Available: http://wwwphoronix-test-suitecom/ [Accessed July 2019].

Le, L. (2010). *Payload already inside: datafire-use for rop exploits*. USA: Black Hat.

Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., Guan, H. (2017). Transparent and efficient cfi enforcement with intel processor trace. In *2017 IEEE International Symposium on High performance computer architecture (HPCA)* (pp. 529–540): IEEE.

Ming, J., Xu, D., Wang, L., Wu, D. (2015). Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (pp. 757–768): ACM.

Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E. (2010). G-free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference* (pp. 49–58).

Pappas, V. (2012). *kbouncer: Efficient and transparent rop mitigation*.

Pappas, V., Polychronakis, M., Keromytis, A.D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security Symposium* (pp. 447–462).

Pappas, V. (2015). *Defending against return-oriented programming*. New York: Columbia University.

Riden, J., McGeehan, R., Engert, B., Mueter, M. (2007). Know your enemy: Web application threats, using honeypots to learn about http-based attacks.

Salwan, J. (2011). Ropgadget–gadgets finder and auto-roper.

Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A. R., Holz, T. (2015). Counterfeit object-oriented programming: on the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and privacy (SP)* (pp. 745–762): IEEE.

Schwartz, E.J., Avgerinos, T., Brumley, D. (2011). Q: Exploit hardening made easy. In *USENIX Security Symposium* (pp. 25–41).

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

Silic, M., & Lowry, P. B. (2019). Breaking bad in cyberspace: Understanding why and how black hat hackers manage their

nerves to commit their virtual crimes. *Information Systems Frontiers*, 1–13.

Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A. R. (2013). Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and privacy (SP)* (pp. 574–588): IEEE.

Vishwanath, A. (2015). Diffusion of deception in social media: Social contagion effects and its antecedents. *Information Systems Frontiers*, *17*(6), 1353–1367.

Yan, F., Huang, F., Zhao, L., Peng, H., Wang, Q. (2016). Baseline is fragile: On the effectiveness of stack pivot defense. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)* (pp. 406–413): IEEE.

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W. (2013a). Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and privacy (SP)* (pp. 559–573): IEEE.

Zhang, M., & Sekar, R. (2013b). Control flow integrity for cots binaries. In *USENIX Security Symposium* (pp. 337–352).

Xin Huang is a graduate student at Wuhan University, China. He received his B.S. degree in Computer Science from Wuhan University, China, in 2018. His research interests include software security, moving target defense, hardware-assisted security and AI security.

Fei Yan is an associate professor of the School of Cyber Science and Engineering at Wuhan University, China. He received his Ph.D. degree from Wuhan University, China, in 2007. He is a co-founder of ChinaSigTC (Chinese special interest group on trusted cloud) and served associate chair of program committee of CTCIS (Chinese Trusted Computing and Information Security Conference) from 2017 to 2019. His current research interests include system security, trusted computing, moving target defense and AI security.

Liqiang Zhang is a lecturer of the School of Cyber Science and Engineering at Wuhan University, China. He is a senior evaluator for Chinese Classified Protection of Cybersecurity, and a senior Chinese Information Technology Project Management Professional. His current research focus on system security, trusted execution environment and security measurement. He has published more than 20 papers in these areas.

Kai Wang was a graduate student at Wuhan University before July of 2019. He received his M.S. degree in Cyber Science and Engineering from Wuhan University, China, in 2019, and received his B.S. degree from Northwestern Polytechnical University, China, in 2016. His research interests include system security and software security.