Marlin: Mitigating Code Reuse Attacks Using Code Randomization

Aditi Gupta, Javid Habibi, Michael S. Kirkpatrick, and Elisa Bertino

Abstract—Code-reuse attacks, such as return-oriented programming (ROP), are a class of buffer overflow attacks that repurpose existing executable code towards malicious purposes. These attacks bypass defenses against code injection attacks by chaining together sequence of instructions, commonly known as gadgets, to execute the desired attack logic. A common feature of these attacks is the reliance on the knowledge of memory layout of the executable code. We propose a fine grained randomization based approach that breaks these assumptions by modifying the layout of the executable code and hinders code-reuse attack. Our solution, *Marlin*, randomizes the internal structure of the executable code by randomly shuffling the function blocks in the target binary. This denies the attacker the necessary a priori knowledge of instruction addresses for constructing the desired exploit payload. Our approach can be applied to any ELF binary and every execution of this binary uses a different randomization. We have integrated Marlin into the bash shell that randomizes the target executable before launching it. Our work shows that such an approach incurs low overhead and significantly increases the level of security against code-reuse based attacks.

Index Terms—Return oriented programming, code randomization, security, malware

1 Introduction

Return oriented programming (ROP) attacks are an advanced form of buffer overflow attacks [1] that reuse existing executable code towards malicious purposes. While earlier exploits involved the injection of malicious code [1], the recent trend has been to reuse executable code that already exists, primarily in the application binary and shared libraries such as *libc*. These code reuse attacks can bypass traditional defenses against code injection attacks such as $W \oplus X$ protection [2] that prevents execution of arbitrary code that is injected into the memory. In a basic code reuse attack, for instance return-into-*libc* attack [3], a buffer overflow corrupts the return address to jump to a *libc* function, such as `system`. This type of attack then evolved into a more generic ROP attack [4]. In ROP, the attacker identifies small sequences of binary instructions, called *gadgets*, that end in a `ret` instruction. By placing a sequence of carefully crafted return addresses on the stack, the attacker can use these gadgets to perform arbitrary computation. These attacks continued to evolve, with newer techniques using gadgets that end in `jmp` or `call` instructions [5].

As these attacks rely on knowing the location of code in the executable and libraries, the intuitive solution is to randomize process memory images. In basic address space layout randomization (ASLR), the start address of the code segment is randomized. That is, two different running instances would have a different base address, so the addresses that an attacker needed to jump to in one instance would not be the same as the addresses in the other instance. Although said approach initially seemed promising, 32-bit machines provide insufficient entropy as there are only 2^{16} possible starting addresses. This makes the approach vulnerable to brute-force attacks [6]. While upgrading to 64-bit helps, it is not a universal solution. Specifically, 32-bit (and smaller) architectures will continue to be used as legacy systems and in embedded systems. Furthermore, recent work has demonstrated that an attacker can use information leakage to discover the randomization parameters, thus eliminating the defensive benefits of upgrading [7].

Rather than abandoning the idea of randomization on 32-bit architectures, we propose to re-examine the granularity at which randomization is performed as a defense against ROP attacks. In considering a new defensive technique, we start with two observations. First, the main shortcoming of earlier randomization-based techniques was insufficient entropy, thus making brute-force attacks feasible. Second, executable code can naturally be broken into many *function blocks* that can potentially be shuffled. Consequently, the amount of possible randomization generated can be significantly increased by permuting these code blocks within the executable. For instance, if an application has 500 function blocks, there are $500! \approx 2^{3,767}$ possible permutations of these function blocks which significantly increases the brute force effort required from an attacker.

Our system, *Marlin*, introduces a randomization technique that shuffles the function blocks in an application binary. This technique is integrated into a customized bash shell that randomizes the target binary at load time just before execution. This randomization approach has many benefits. First, as stated above, for any considerably sized code base with a large number of function blocks, the number of possible randomized results clearly makes brute-force attacks infeasible. Second, this approach can be applied to any ELF binary without requiring the source code of an application. Third, the randomization is performed just

- *A. Gupta, J. Habibi, and E. Bertino are with the Department of Computer Science, Purdue University, West Lafayette, IN47907 E-mail: {aditi, jhabibi, bertino}@purdue.edu.*
- *M.S. Kirkpatrick is with the Department of Computer Science, James Madison University, Harrisonburg, VA22807. E-mail: kirkpams@jmu.edu.*

Manuscript received 28 Dec. 2013; revised 22 June 2014; accepted 11 July 2014. Date of publication 4 Aug. 2014; date of current version 15 May 2015. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2014.2345384

before executing the binary which means that potentially every execution of this binary results in a different address layout. Finally, our scheme offers an alternative to approaches that dynamically monitor critical data like return addresses. Although these schemes are effective, they distribute the performance cost throughout the execution life-time of the process. In our solution, the entire performance cost is paid once during process setup, and is quite reasonable; after the execution begins, the code runs as originally designed.

This work extends a preliminary approach that was presented in our previous papers [8], [9]. Specifically, we have made the following extensions. First, the earlier approach was applicable only to limited type of binaries, that is, the binaries that do not use function pointers. Our current work addresses this limitation by handling such binaries as well. Second, the earlier approach provided an offline-tool to perform the randomization. In the current work, we have fully integrated the Marlin technique into a custom bash shell so that the application randomization happens seamlessly when the application is executed using this modified bash shell. We have also implemented a whitelist approach that allows user to specify the list of binaries that should be randomized. For example, one may wish to randomize only those applications that accept input from a remote user and are thus more vulnerable to attack. Third, we have improved our earlier implementation of Marlin to make the binary randomization much faster. Finally, in the current paper we include a more extensive set of experiments to evaluate the Marlin technique.

We are not the only researchers to have investigated software diversity for ROP attack mitigation. As discussed in Section 2.2, the other approaches suffer from one or more of the following limitations. First, the software diversification is not done frequently enough. Second, some of the existing defenses require the source code or other additional information that is not usually available. Third, the randomization is not fine grained enough leaving large code chunks unrandomized. Fourth, significant runtime overhead is incurred throughout the runtime of the application by introducing additional data structures. Marlin addresses these limitations and provides a strong and efficient defense technique against ROP attacks.

With any solution, there are always costs that must also be considered. In our proposed scheme, there is a performance impact when the process begins. We have evaluated the time to randomize compiled binaries on a selection of commonly used applications and Linux `coreutils`, showing that the performance penalty for Marlin is reasonable in the average case. Thus, our work demonstrates that, although Marlin imposes certain performance costs, its success in thwarting ROP attacks makes this a feasible approach for systems that prioritize execution integrity over optimal performance. In Section 3 we describe techniques for minimizing this performance impact. For instance, performing the randomization during offline pre-processing significantly reduces the startup costs.

The remainder of this paper is structured as follows. We start by surveying code-reuse attacks and proposed defenses in Section 2. Section 3 describes our approach in more detail, including optimization techniques to reduce overhead. Section 4 discusses the implementation details of Marlin. Section 5 shows the results of various experiments that were performed to evaluate our approach. Section 6 concludes the paper and outlines the future work.

2 Background and Related Work

In this section, we start with a brief summary of ROP attacks and existing defenses. We then summarize critical factors of code-reuse attacks.

2.1 Return-Oriented Programming

Return-oriented programming is an exploit technique that has evolved from stack-based buffer overflows. In ROP exploits, an attacker crafts a sequence of *gadgets* that are present in existing code to perform arbitrary computation. A gadget is a small sequence of binary code that ends in a `ret` instruction. By carefully crafting a sequence of addresses on the software stack, an attacker can manipulate the `ret` instruction semantics to jump to arbitrary addresses that correspond to the beginning of gadgets. Doing so allows the attacker to perform arbitrary computation. These techniques work in both word-aligned architectures like RISC [10] and unaligned CISC architectures [4]. ROP techniques can be used to create rootkits [11], can inject code into Harvard architectures [12], and have been used to perform privilege escalation in Android [13]. Initiating a ROP attack is made even easier by the availability of architecture-independent algorithms to automate gadget creation [14]. Additionally, the same technique of stringing together gadgets has been used to manipulate other instructions, such as `jmp` and their variants [5], [15], [16].

2.2 Defenses

Address obfuscation [17] and address-space layout randomization (e.g., PaX [2]) are two well-known techniques for defending against code-reuse attacks. Address obfuscation and ASLR on 32-bit architectures have the same short-comings of instruction set randomization in that the small amount of randomization leaves applications vulnerable to attacks [6], [18]. Shacham et al. demonstrated that existing randomization techniques can be defeated by brute-force. Also, information leakage can allow an attacker to learn the randomized base address of *libc* [7]. Consequently, simply randomizing the base address does not effectively block the attack. Bhatkar et al. [17] suggest randomizing function blocks as one of the address obfuscation techniques; however this particular technique was neither implemented nor discussed in detail.

Another approach for code-reuse attacks is to detect and terminate the attack as it occurs. DROP [19] is a binary monitor implemented as an extension to Valgrind [20]. DROP detects `ret` instructions and initiates a dynamic evaluation routine based on a statistical analysis of normal program behavior. When a `ret` instruction would end in an address in *libc*, DROP determines if the current execution routine exceeds a candidate gadget length threshold. These thresholds are based on a static analysis of normal program behavior. The binary to be run must be compiled with DROP enabled. Another approach, DynIMA [21], combines the memory measurement capabilities of a TPM with dynamic taint analysis to monitor the integrity of the process in execution. Other approaches store sensitive data, such as return addresses, on a shadow stack and validate their integrity before use [22], [23]. Defense techniques that monitor and/or lock control flow [24], [25], [26] to stop ROP attacks have also been proposed. The disadvantage of these approaches is that there is a non-zero performance cost for every checked instruction. Also, with the exception of [23], [24], [25], [26], these schemes assume gadgets end in `ret` instructions, and do not consider the more general case where gadgets may end in jump instructions.

Compiler-based solutions [27], [28] that create code without `ret` instructions have also been proposed. However, these techniques have the obvious disadvantage that they fail to prevent attacks based on `jmp` instructions. Compiler techniques have also been proposed to generate diversity within community of deployed code [29]. That is, instead of all users executing the same compiled image (i.e., a monoculture), when a user downloads an application from an "app store" model, the compiler generates a unique executable, which would stop a single attack from succeeding on all users. While we find this approach very promising, it is not universally applicable, and would not stop an attacker with a singular target. Further, it would require access to application's source code that is not typically available.

Proactive obfuscation [30] uses an obfuscating program that applies a semantics-preserving transformation to the protected server application. That is, the executable image differs each time the obfuscator runs, but the end result of the computation is identical. The *proactive* aspect means that the server is regularly taken off-line and replaced with a new obfuscated version, thus limiting the time during which a single exploit will work. Our work can be seen as another instance of proactive obfuscation. However, our approach has more general applicability than to replicas in distributed services. Some techniques such as [31], [32] reorder functions for performance optimization at linking stage. Since the output of these approaches is just one optimized binary, they do not diversify the binary and hence do not offer any strong protection against ROP attacks.

Another work that uses similar methodology as ours is ASLP [33]. However, this work substantially differs from our work in intent, requirements and low-level techniques. ASLP requires user input, while Marlin works without user input. ASLP requires relocation information, without which the program has to be recompiled and relinked. It involves rewriting ELF header, program header and section headers and shuffling around sections in addition to functions and variables. We randomize only the function blocks within the code segment and show that such randomization introduces sufficient entropy to thwart ROP attacks. Thus, our approach incurs less overhead than ASLP. Bhatkar et al. [34] also propose a randomization approach to protect against memory error exploits. However, their technique differs from Marlin since they associate a function pointer with every function and transform every function call into an indirect function using this function pointer while we perform binary rewriting. Also, unlike Marlin, function reordering is not done at load time.

Some very recent research approaches also explore the idea of software diversity as a defense against ROP attacks. ILR [35] randomizes location of every instruction in the application code and relies on a process-level virtual machine that incurs a performance cost throughout the duration of the application. In contrast, Marlin's performance impact is primarily limited to the start-up cost. Pappas et al. [36] propose an in-place code randomization technique that probabilistically breaks 80 percent of the instruction sequences that are useful for attacks. However, Marlin provides stronger guarantees by shuffling the entire memory image, thus probabilistically breaking all instruction sequences. Also, Marlin randomizes the executable with *every run* unlike [35] and [36] that do not re-diversify the binary. XIFER [37] and STIR [38] apply software diversification to an application at runtime to protect against code-reuse attacks. While XIFER and STIR apply diversification at the granularity of basic blocks, we randomize at function block level and show that this is sufficient to make brute force attacks infeasible. Also, these techniques would incur more overhead than Marlin as they randomize at a very fine granularity. Marlin is a novel solution for thwarting ROP attacks that does not have the limitations discussed above.

2.3 Enabling Factors for Code-Reuse Attacks

Based on our survey of ROP attacks and defenses, we have identified distinct characteristics and requirements for a successful exploit. The fundamental assumption and enabling factor for such attacks is as follows:

The relative offsets of instructions within the application's code are constant. That is, if an attacker knows any symbol's address in the application code, then the location of all gadgets and symbols in application's codebase is deterministic.

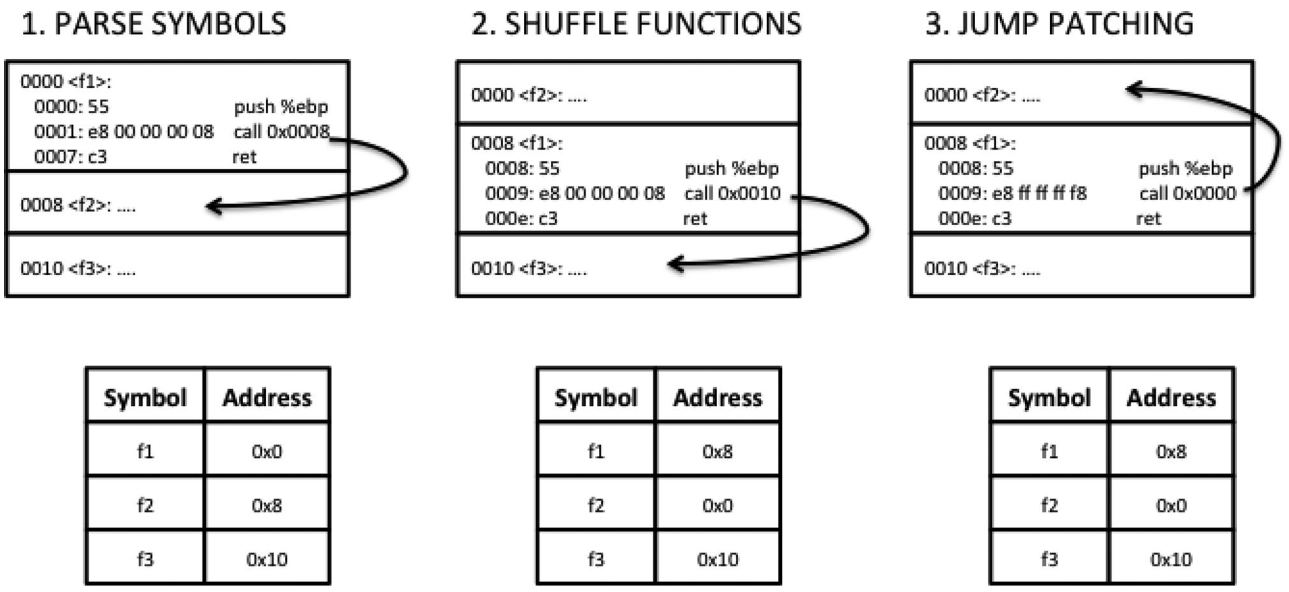
We argue that a defensive technique that undermines these invariants will present a robust protection mechanism against these threats.

3 Marlin Defense Technique

Code-reuse attacks make certain assumptions (as discussed in Section 2.3) about the address layout of application's executable code. Marlin's randomization technique aims at breaking these assumptions by shuffling the code blocks in the binary's `.text` section with *every* execution of this binary. This significantly increases the difficulty of such attacks since the attacker would need to guess the exact permutation being used in the current process execution. This shuffling is performed at the granularity of function blocks as discussed in Section 3.2. The various steps involved in Marlin processing are shown in Fig. 1. Marlin is integrated into a modified bash shell that randomizes the target application just before the control is passed over to this application for execution. Thus, every execution of the program results in a different process memory image as illustrated in Fig. 2a. Fig. 2b illustrates how shuffling the code results in a sequence of gadgets that is not intended by the attacker. We now present Marlin technique in detail.

3.1 Attack Assumptions

We start by describing the basic assumptions for a ROP attack scenario. The vulnerable application may have a buffer overflow or heap overflow vulnerability that can be leveraged by an attacker to inject an exploit payload. The



```
0000 <f1>:
  0000: 55                push %ebp
  0001: e8 00 00 00 08    call 0x0008
  0007: c3                ret
0008 <f2>: ....
0010 <f3>: ....
```

Symbol	Address
f1	0x0
f2	0x8
f3	0x10

```
0000 <f2>: ....
0008 <f1>:
  0008: 55                push %ebp
  0009: e8 00 00 00 08    call 0x0010
  000e: c3                ret
0010 <f3>: ....
```

Symbol	Address
f1	0x8
f2	0x0
f3	0x10

```
0000 <f2>: ....
0008 <f1>:
  0008: 55                push %ebp
  0009: e8 ff ff ff f8    call 0x0000
  000e: c3                ret
0010 <f3>: ....
```

Symbol	Address
f1	0x8
f2	0x0
f3	0x10

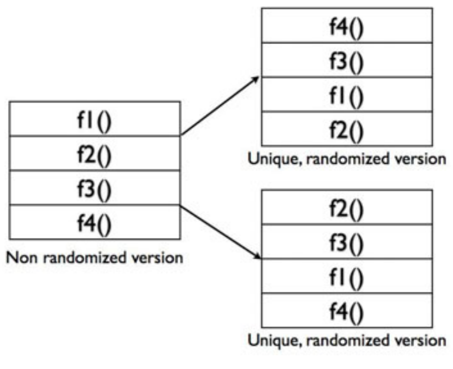
Fig. 1. Processing steps in Marlin.

system is assumed to be protected using write or execute only policy ($W \oplus X$) and the attacker can not inject arbitrary executable code in the stack or the heap. The attacker is assumed to have access to the target binary that has not yet undergone Marlin processing. The attacker is also assumed to be aware of the functionality of Marlin. However, the attacker can not examine the memory dump of the running process and is unaware of how exactly the code is randomized for the currently executing process image. Our approach protects against both remote and local exploits as long as the attacker is not able to examine the memory of the target process. For instance, in this threat model, a local attacker can not attach a debugger to a process that is running as root and obtain its memory dump.

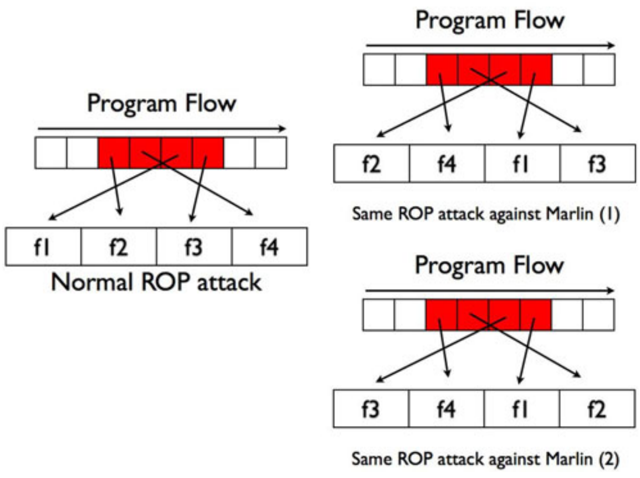
3.2 Granularity of Randomization

Code can be randomized at various levels of granularity such as instruction level, basic block level, function level, segment level or just the base address. Choosing the right granularity of randomization is a tradeoff between effectiveness (measure of security offered) and efficiency (measure of overhead incurred) of the resulting defense scheme. While randomizing at finer granularity (such as basic block or instruction level) provides higher entropy, it may also incur higher overhead as it breaks the principle of locality. That is, the basic blocks that comprise a function might be moved to different pages and the system would have to load multiple memory pages to execute a single function. Randomization at basic block also involves handling many more types of jumps and calls which require precise control flow graph information. Such precise control flow information typically cannot be completely extracted from an executable. This makes basic block based randomization more difficult to handle in the absence of complete control flow graph information.

On the other hand, randomization at the function level granularity eases the handling of several jumps and calls. As the function body remains intact during the shuffling phase, relative jumps are not affected as their target lies within the same function. The same holds true for certain computed jumps. This avoids having to patch the target address for near jumps that occur within a function body. Also, by keeping all the basic blocks of a function body together, the overhead of loading multiple pages per function can be avoided. For these reasons, we chose to



(a) Unique output with every run



(b) Mitigation of ROP attack

Fig. 2. Effect of function block randomization.

implement randomization at the function level granularity in Marlin. We show later in the evaluation section that even with this coarse granularity, our approach offers strong protection against brute force attack. We now discuss the various steps involved in application code randomization.

3.3 Preprocessing Phase

As mentioned above, Marlin randomizes the application binary at the granularity of function blocks. This requires identifying the function blocks in the application binary. In preprocessing phase, the ELF binary is parsed to extract the function symbols and associated information such as start address of the function and length of the function block. However, traditional binaries are typically stripped binaries and do not contain symbol information. In such cases, we first restore the symbol information using an external tool, *Unstrip* [39]. Once the symbol information is restored and identified, we proceed on to the next stage of Marlin processing that randomizes the application binary.

3.4 Randomization Algorithm

Once the function symbols have been identified, Marlin generates a random permutation of this set of symbols. The function blocks are then shuffled around according to this random permutation. Shuffling the function blocks in an application binary changes the relative offsets between instructions that may affect various jump and call instructions. The target destination for these jumps/calls can be specified either as an absolute address or as a relative offset. Relative jumps increment or decrement the program counter by a constant value as opposed to absolute jump that directly jump to a fixed address. When the function blocks are randomized, these jumps will no longer point to the desired location and must be 'fixed' to point to the proper locations. We achieve this by performing *jump patching*.

The randomization algorithm described in Algorithm 1 involves two stages. In the first stage, the function blocks are shuffled according to a certain random permutation. During this shuffling, we keep a record of the original address of the function and also the new address where the function will reside after the binary has been completely randomized. This information is stored in a *jump patching table*. Note that this jump patching table is discarded before the application is given control, thus preventing attacker from utilizing this information to de-randomize the memory layout. In the second stage, the actual jump patching is executed where the jump patching table is examined for every jump that needs to be patched. Whenever a relative jump is encountered, the algorithm executes the `PatchRelativeJump()` method to redirect the jump to the correct address in the binary. The `PatchRelativeJump()` method takes the current address of the jump and the address of the jump destination to determine the new offset and patch the jump target. The second case is the computed jumps where the contents of a register specify the absolute address of the destination, for example call to function pointers. We handle these cases by doing a backward analysis and fixing the instruction where the function address is being loaded into a register. If the destination address is obtained from `.data` section (for example, in case of global function pointers), then we patch the `.data` section with the new value. This processing is done by the `PatchAbsoluteJump()` method shown in Algorithm 1.

Algorithm 1. Code Randomization algorithm

```
Input: Original program, $P$
Output: Randomized program, $P_R$
$L$ = All symbols in $P$
$F$ = A list of forbidden symbols that should not
      be shuffled
$L = L - F$
$O_L$ = Ordered sequence of symbols in $L$
$S.Addr_P$ = Address of symbol $S$ in program $P$
$J.Addr_P$ = Address of jump instruction $J$ in program $P$
$J.Dest_P$ = Destination address of jump $J$ in program $P$
$J.Sym$ = Symbol that $J$ is jumping into
/* Permutation stage */
for Every symbol $S \in L$ do
    $R$ = Randomly select another symbol in $L$
    Swap $S$ and $R$ in $O_L$
$P_R$ = Permuted program according to symbol
        order in $O_L$
/* Jump patching stage */
for Every symbol $S \in L$ do
    for Every jump $J \in S$ do
        if $J$ is a relative jump to within $S$ then
            /* No action needed */
        else if $J$ is a relative jump to outside $S$ then
            $J.Dest_{P_R} =$
            $J.Dest_P + (J.Sym.Addr_{P_R} -$
            $J.Sym.Addr_P) - (S.Addr_{P_R} - S.Addr_P)$
            PatchRelativeJump($J.Addr_{P_R}$, $J.Dest_{P_R}$)
        else if $J$ is an absolute jump then
            PatchAbsoluteJump($J.Addr_{P_R}$, $J.Dest_{P_R}$)
```

The run-time shuffling of the function blocks assures that multiple instances of the same program have different address layouts. Thus, to defeat Marlin, an attacker would need to dynamically construct a new exploit *for every instance of every application* which is not possible since the randomized layout is not accessible to the attacker. We now discuss the security guarantees offered by Marlin.

3.5 Security Evaluation

We now show that our randomization technique significantly increases the brute force effort required to attack the system. In a brute force attack, the attacker will randomly assume a memory layout and craft exploit payload according to that address layout. A failed attempt will usually cause a segmentation fault due to illegal instruction and the crashed process or thread will need to be restarted. We now compute the average number of attempts required by an attacker to succeed. A successful attack is assumed to be equivalent to guessing the correct permutation used for randomization.

In the discussion that follows, let n denote the number of symbols (excluding forbidden symbols) in an application binary. The total number of possible permutations that can be generated for this application is $N = n!$. Let $P(k)$ denote the probability that the attack is successful for the first time at the kth attempt. Let X be a random variable denoting the number of brute force attempts after which the attack is successful for the first time (that is, the attacker guesses the correct permutation). We will now estimate the *average* value of X. We consider the following two cases.

Case 1. A failed attempt crashes the process and causes it to be restarted.

In this event, the process will be restarted with a new randomization. The subsequent brute force attempts by an attacker will be independent since he would learn nothing from the past failed attempts. That is, the probability of success at kth attempt is constant and independent of k. Let $p = \frac{1}{N}$ denote the probability of success at any attempt. Then, the average number of attempts before the attack is successful for the first time is

$$\mathrm{E}[X] = (p * 1) + (1 - p) * (1 + \mathrm{E}[X]) = \frac{1}{p} \Rightarrow \mathrm{E}[X] = n!$$

Thus, the attacker would have to make an average $n!$ number of attempts to correctly guess the randomized layout and launch a successful ROP attack.

Case 2. A failed attempt crashes a thread of the process and causes only that thread to be restarted.

In this event, since the process is still executing, the memory layout will remain same. Every failed attempt will eliminate one permutation. The probability that the first success is achieved at kth attempt is

$$P(k) = \left(\prod_{i=1}^{k-1} \frac{N-i}{N-i+1} \right) * \frac{1}{N-k+1} = \frac{1}{N}.$$

The average number of attempts before first success can be computed as

$$\mathrm{E}[X] = \sum_{x=1}^{N} x * P(x) = \sum_{x=1}^{N} x * \frac{1}{N} = \frac{N+1}{2} \Rightarrow \mathrm{E}[X] = \frac{n!+1}{2}.$$

So, the attacker will need an average $\frac{n!}{2}$ number of brute attempts to correctly guess the randomization and launch successful ROP attack. Given enough time and resources, the attacker can try all possible permutations one after the other and will require at most $n!$ attempts for a successful brute force attack.

As an example, to launch a successful ROP attack against an application with 500 symbols that is protected using Marlin, an average $500! = 2^{3,767}$ number of attempts will be required for the first case. This is clearly computationally infeasible. A more extensive evaluation performed using `coreutils` applications is presented later in Section 5 that demonstrates the effectiveness of our technique.

3.6 Discussion

Having described our randomization techniques above, it is necessary to offer a few words about how Marlin applies these techniques while addressing specific implementation challenges that have been identified [6] in regard to memory image randomization. Against ROP attacks, randomization is, by far, the more effective technique. By significantly increasing the entropy of the application image, randomization creates negligible probability that an adversary can craft a chain of gadgets for short-lived applications, as every new process will have a different configuration of function blocks. Specifically, the large number of possible permutations significantly increases the number of attempts needed for a single ROP gadget chain to work.

Shacham et al. [6] correctly point out that full randomization eliminates sharing memory pages between processes. For strong security guarantees, *eliminating sharing is actually desirable*. That is, for some critical applications, it is more important to guarantee integrity than optimal performance. However, in other cases, such strong security guarantees are not required. To accommodate a wide range of tradeoffs, several approaches are possible. First, an executable could be marked as *critical*, which would then be fully randomized. Next, *normal* applications would first detect if another instance of the same executable is already running. If so, the new process would share read-only access to the shuffled code image. Such options can be implemented using flags that get passed to Marlin.

3.7 Optimization Techniques

A straightforward performance optimization for Marlin would be to perform the pre-processing for jump patching only once for each application and store the result in a database maintained by the system. The jump patching algorithm can reuse the information about function blocks from this database in subsequent executions. The database would only need to be updated when the application code changes.

The impact of the code randomization can be reduced by taking the permutation generation off-line. To do so, each application will have a dedicated file containing the next instance's permutation. When a binary is executed, the custom shell sends a signal to a trusted daemon process that runs with low priority and returns the next permutation. The application's function blocks are then shuffled accordingly.

4 Implementation Details

We have implemented a Marlin prototype that can operate on any ELF binary without requiring its source code. The implementation was done for 32-bit x86 architecture on a system running Ubuntu operating system. Implementation of Marlin involved two major components. First part consisted of randomizing the executable code and generating the randomized binary. The second part dealt with integrating this into an existing system such that binary randomization occurs seamlessly with every execution. We discuss the details of Marlin implementation below.

4.1 Code Randomization

Randomizing an application's executable code segment consists of two stages. First is the preprocessing stage that can be done just once per binary and is independent of subsequent executions. This stage involves disassembling a binary and extracting information about the function blocks and also the control flow. The second stage is the actual randomization stage when the function blocks are shuffled and the jump/call targets are patched. We now discuss each of this in further detail.

4.1.1 Preprocessing Stage

Before we randomize the binary, we need to identify the function blocks. We do this by disassembling the binary using `objdump` disassembler and then parsing the dissembler output to extract the function symbols and the relevant information. For each function symbol, we gather information about its location in the executable, the length of the function block and the information on any jumps or calls originating from this function. This information is collected for functions in the PLT table as well in addition to the application defined functions.

While we use `objdump` disassembler, other commercial disassemblers such as IDAPro can be used to obtain more accurate disassembly. Also, several production level binaries are available only as stripped binaries, that is the symbol information has been removed from them. We restore the symbol information using `Unstrip` utility [39] before disassembling it using `objdump`.

4.1.2 Randomization Stage

In this stage, the actual shuffling of the function blocks is performed. The first step is to generate a random permutation of symbols and shuffle the list of symbols to obtain a new order of symbols. The new binary is re-written according to this new symbol order. In our preliminary implementation [8], we did not shuffle certain symbols such as `_start` that were referred to as *forbidden symbols*. Our revised implementation no longer has this limitation and all the symbols within `.text` section are now randomized, including `_start` symbol. This `_start` symbol is the first instruction that executes after the binary is loaded into the memory by the ELF loader. This entry address is stored in ELF header of the binary. Once the application is randomized, we patch the ELF header with the new entry address which is the new location of `_start` symbol.

4.1.3 Fixing Jumps and Calls

The jump and call patching is performed in the same pass when the new randomized binary is written. This is done by using the patch list information that is generated during the preprocessing stage. For each call that needs to be patched, the patch information consists of the name of the parent symbol, the name of symbol being patched to and the offset from the beginning of the parent symbol where the patching needs to be done.

The calls and jumps can be of the following types:

- Call instructions
  - *Call to an application defined function.* In normal function calls (*call* <$f1$>), the target address of the callee function is specified as relative address offset from the address of the call instruction. We fix this target address in the call patching phase using the patch information collected during the preprocessing phase.
  - *Call to a dynamically linked function.* Functions in dynamically linked libraries that are called in the application's code appear in the PLT section of the application's code. Calls to these linked functions (*call* <$f2@plt$>) are also specified as relative offset from the address of the call instruction to the function's PLT entry. These targets are also fixed in the call patching stage by correcting the relative offset.
  - *Call to a function pointer.* Call to function pointers are handled as indirect calls, that is the absolute address of call target is loaded into a register, say $\%eax$, and then the call is made as $call * \%eax$. To fix these types of calls, the absolute address of the callee should be patched at the instruction that loads its address into the $\%eax$ register. This is done by doing a backward analysis starting from an indirect call instruction and tracing backwards until we reach the instruction where the value of function pointer is loaded. In case of global function pointers, the address of function is stored in the data section and eventually loaded from this data section into a register. In these cases, the data section is patched with the corrected function address after shuffling.
- *Jump instructions.* In x86 architecture, jumps can be either conditional jumps or unconditional jumps. Conditional jumps are near jumps while unconditional jumps can be either near or far jumps. We don't need to patch the near jumps as they are within the same function body and specified using the function offset. However, unconditional far jumps transfer program control to the target address without a return. For example, this can happen in the case of goto statement where the jump specifies an absolute address. If the jump destination is outside the application's code, for example a shared library, then this does not need patching. However, if the destination of a far jump is within the application code, then the code needs to be patched. We patch certain far jumps, for instance the jump tables that are created due to some `switch-case` blocks. These jump tables are stored in `.rodata` section of the code. We patch the jump table in this `.rodata` with the new jump targets after randomizing the code.

4.2 System Integration

Software diversification can be applied at various stages in an application's lifecycle ranging from compile-time diversification to runtime-diversification. In Marlin, our goal is to randomize the target binary with each execution. That is, we want to invoke the Marlin functionality whenever the target binary is executed. We considered several approaches to achieve this as discussed below.

The first approach that we considered was modifying the dynamic loader (`ld`). In this approach, the application code that is mapped in the memory by `mmap` call will be randomized just before the control is passed to the target binary. This has the advantage that all the transformations are done in the memory and they are done immediately before the control flow jumps to application's `_start` symbol. This ensures that every application is randomized with each execution and no intermediate files are generated that can be potentially exploited by the attacker. However, this approach incurs several complications and redundant processing. For example, the loader library (`ld-linux.so`) is self contained and does not use any shared library which makes it difficult to integrate Marlin's code into the loader. More importantly, the work that the loader has done to load the normal application code (resolving references etc) is wasted since the functionality needs to be re-executed after randomizing the code again. For these reasons, we decided against using this approach.

The second approach that we considered was modifying the `execve` system call such that it first executes Marlin to randomize the target executable and then executes the randomized code. However, since `execve` is used by almost every execution in the system including the kernel code, modifying this function can introduce a lot of instability into the system, especially during the testing phase. Also, to prevent the recursive invocations of Marlin onto itself, one would need to modify the `execve` definition which is not a good solution since `execve` is called at several places in both user and kernel code. Thus, we decided against using this approach as well.

Finally, the third approach that we considered was a custom secure shell approach. In this approach, we modified the normal shell code to create a secure shell that would randomize the target binary before executing it. This is the approach that we adopted in implementing the Marlin prototype. We modified the source code for bash shell, specifically the `shell_execve` function that is responsible for making a call to `execve` method. We created a hook just before the `execve` call to randomize the code for the target binary. This approach has the advantage that it allows us to test our system without interfering with the existing system functionality. In the deployment of the production version, one can easily replace the normal bash shell with our secure shell to ensure that all the executions invoked by this shell are randomized.

Further, we implemented a whitelist that allows for selectively randomizing only certain application binaries. For example, one may wish to randomize only those applications that have user interactions, that is they accept input from a user (can be a remote user). Our implementation supports specifying the whitelist entries in three different ways. First, the entry can be an absolute path of a directory in which case all the files contained in this directory and its sub-folders will be randomized. Second, the entry can be specified as the absolute path of an application in which case this application is randomized whenever it is executed using secure shell. Finally, the entry can be specified just using the name of the executable in which case any executable with the specified name, irrespective of its path, will be randomized before execution. This whitelist is protected and can only be modified by the superuser.

5 Evaluation

We now describe various experiments that were performed to evaluate *Marlin* technique. These experiments test the effectiveness of Marlin technique and also the performance overhead incurred due to randomization. The experiments were performed on a Linux virtual machine with two processor cores and 4 GB RAM (host machine processor was Intel Core i5 2.4 GHz with 6 GB RAM). This VM had ASLR and $W \oplus X$ protection enabled while the experiments were being performed. We used `coreutils` binaries, some commonly used application binaries (see Table 1) and byte-unixbench [40] benchmarks to conduct various experiments. To launch attacks against Marlin-protected binary, we use ROPgadget (v3.3.3) [41], an attack tool that automatically creates exploit payload for ROP attacks by searching for gadgets in an application's executable section.

TABLE 1
List of Applications Used in Evaluation

Application	Version
Apache	2.4.7
Bash	4.20
Brasero	3.11.0
Cups	1.7.0
Coreutils (105 applications)	8.22.1
Dhclient	4.2.5
Emacs	24.3
Gcc	4.8.1
Gedit	3.8.3
Ghex	3.8.1
Gimp	2.80
Git	1.8.5
Gnome-terminal	3.0.1
Gtkpod	2.1.4
Gzip	1.60
Lame	3.99.5
Make	3.81
Mono	–
Nano	2.3.2
OpenSSH	6.4
Qemu	1.7.0
Subversion	1.8.5
Tar	1.27
Vim	7.4
Vlc	2.1.2
Wine	1.1.27
Wireshark	1.11.2

5.1 Effectiveness

First, we tested the effectiveness of Marlin using a test application that has a buffer overflow vulnerability. This application, `ndh_rop`, was included as a part of the ROPgadget test binaries. We used ROPgadget on this target application and found 162 unique gadgets. These were sufficient to craft a shell code exploit payload. When this exploit payload was provided as an input to the unprotected binary, it gave us a shell. Next, we randomized this application using Marlin technique and tried to attack it using the same input payload. The attack did not succeed and failed to provide us with a shell.

This highlights the sensitivity of these attacks to slight changes in the address layout. ROP attacks operate under the strong assumption of a static address layout of executable code. In our threat model, the attacker only has access to the unprotected binary and is not aware of the exact permutation that has been used for randomization. So he can only run ROPgadget tool on the unprotected test application.

5.1.1 *Brute Force Effort*

In Section 3.5, we computed the average number of attempts required to successfully attack a randomized binary. This

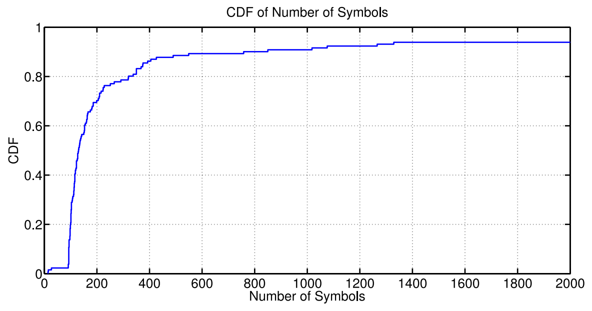


Fig. 3. CDF for number of symbols.

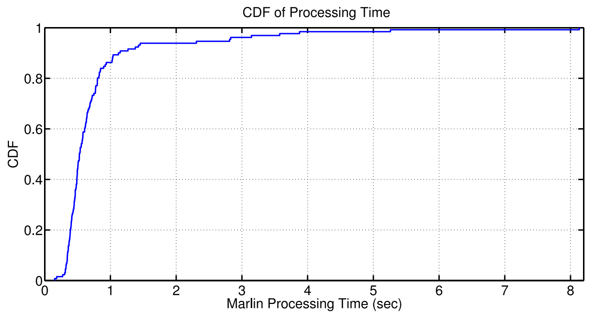


Fig. 4. CDF for Marlin processing time.

brute force effort is approximately `factorial(n)` where `n` is the number of symbols in a binary. We performed an extensive evaluation of this using 131 ELF binaries corresponding to 105 `coreutils` applications and 26 commonly used applications. Fig. 3 shows the CDF of number of symbols present in these applications. We noticed that around 97.7 percent of these benchmarks have more than 80 symbols (indicating an effort of $80!$ attempts). We observed an average of 470 symbols and a median of 130 symbols present in these applications. Thus, the number of brute force attempts in a general case can be approximated to $130! \approx 2^{730}$ attempts which is quite significant. Also, on an average, we observed the time to compute one attack payload is 15.48 seconds.

It is interesting to note that the effectiveness of protection offered by Marlin depends on the modularity of the program. An application that has several function modules will be more secure against brute force attempts when protected with Marlin. If the entire code of an application is organized in few functions, then irrespective of the size of the binary, it will still be quite susceptible to brute force attacks since it would contain large chunks of unrandomized code. Randomizing at finer granularity, for example at the granularity of basic blocks or instructions, will solve this issue. However, we believe that randomization breaks the locality principle and the randomized binary may suffer a performance hit. Thus, as a trade off, we chose to randomize at the granularity of function block.

5.1.2 *Gadget Displacement*

Next, we studied the entropy introduced by our randomization approach by measuring the gadget displacement. That is, we measured how many gadgets are moved due to randomization by Marlin technique. To measure this, we extend the ROPgadget tool to compare the original binary with the randomized binary and compute the number of unique gadgets that were found in former and are no longer present at the same address in latter. This experiment was also performed on the same set of 131 application binaries as used in Section 5.1.1 with 20 iterations per binary.

We measure two types of gadget displacement. First, we measure the displacement of unique gadgets that are found by ROPgadget in the executable sections of the target binary. Note that these gadgets are not necessarily from `.text` section and may belong to other executable sections such as `.plt` section. In this case, we observed an average of 71.8 percent and a median of 72.5 percent gadgets were displaced in the randomized binaries. Since we randomize only `.text` section, the gadgets found in other executable sections were unmoved. Next, we restricted the search for unique gadgets to only `.text` section and measured the number of gadgets that were displaced by randomization. In this case we observed an average of 99.78 percent gadget displacement (with median as 100 percent displacement). Thus, nearly all gadgets in the `.text` section are displaced.

We can conclude from above observations that randomizing at function level granularity leads to high gadget displacement which is quite effective against ROP attacks. This eliminates the need for randomizing at a more finer granularity such as basic blocks or instruction level.

5.2 Overhead Analysis

We evaluated the efficiency of Marlin by measuring two variables. First, we measured the processing cost incurred by Marlin while loading an application as discussed in Section 5.2.1. Second, we measured the runtime overhead of the randomized binaries. This is discussed in more detail in Section 5.2.2.

5.2.1 *Marlin Processing Overhead*

When an application is loaded, Marlin identifies the function blocks and records information about them (such as start address, length) that is used later in jump patching. This computation is independent of the individual randomizations. Next phase involves shuffling the function blocks and patching the jumps. Marlin processing cost is the combined overhead of these two phases. We measure Marlin processing overhead for the same set of 131 ELF binaries used in above experiments with 20 randomizations per binary.

Fig. 4 shows the CDF of processing cost for Marlin for these 131 applications. We notice that 95 percent of these applications incurred less than 1.43 seconds processing time. This is quite reasonable since this is a one time overhead incurred only at the application load time. We observed that applications with larger number of symbols incur more processing overhead. For instance, the application gimp took significantly longer time to process (8.13 seconds). This is because it contained 10,760 symbols in contrast to a median of 130 symbols by other applications. The average time taken by Marlin processing was 0.87 seconds with a median of 0.53 seconds.

Thus, we observe that the processing overhead due to Marlin is very minimal. Also, the performance hit is incurred only at the load time of the application. Once the

TABLE 2
Comparison with Other Defense Techniques

Defense	ROP types mitigated	Technique	Stage	Platform	Benchmark	Runtime overhead	Space overhead	Gadget Displacement	Runtime data-structures
Marlin	ROP, JOP	Function shuffling	Load time	Linux (x86 ELF)	Linux coreutils, byte-unixbench, COTS	0%	0%	99.78%	No
ILR [35]	ROP, JOP	Instruction shuffling	Installation, Execution	Linux (x86 ELF) Linux (x86 ELF)	SPEC CPU 2006	13-16%	14MB - 264MB	99.96%	Yes (Fall-through map)
STIR [38]	ROP, JOP	Basic-block shuffling	Load time	Linux (x86 ELF), Windows (PE)	SPEC CPU 2000, Linux coreutils, COTS	4.6% (SPEC) 0.3% (coreutils) 1.6% (overall)	73% (file size) 37% (process size)	99.99%	No
XIFER [37]	ROP, JOP	Code piece shuffling	Load time	Linux (x86 ELF), ARM	SPEC CPU 2006	1.2-5%	1.76% (file size), 5% (at runtime)	100%	No
IPR [42]	ROP, JOP	Instruction reordering, Equiv. instruction substitution	Offline tool	Windows (PE)	Wine test suite	0%	N/A	76.9%	No
DROP [19]	ROP only	Check gadget sequence length with threshold	Execution	Linux (x86 ELF)	COTS	430%	N/A	N/A	No
ROPDefender [22]	ROP only	Instrumentation to check return address	Execution	Linux Windows	SPEC CPU 2006	117% (SPECint) 49% (SPECfp)	Yes (Statistics not reported)	N/A	Yes (shadow stacks)
ROPecker [43]	ROP, JOP	Check for long gadget chain	Installation, Execution	Linux (x86 ELF)	SPEC CPU 2006, Bonnie++, Apache	2.6% (SPEC CPU) 1.5% (disk I/O) 0.08-9.72% (Apache)	210MB	N/A	Yes (instruction and gadget database)
CCFIR [25]	ROP, JOP	Restrict jump target using white-list	Installation, Execution	Windows (x86 PE)	SPEC CPU 2000	3.6% (SPECint 2000) 0.59% (SPECfp 2000) 4.2% (SPECint 2006)	Yes (statistics not reported)	N/A	Yes (Spring-board)
CFL [24]	ROP, JOP	Control flow locking	Compilation, Execution	Linux (x86 ELF)	SPEC CPU 2000, SPEC CPU 2006, COTS	0-21%	Yes (Statistics not reported)	N/A	No

application binary has been randomized, it executes like a normal application binary.

5.2.2 Runtime Overhead

We measured the runtime overhead of randomized binaries to see if shuffling the functions affects the execution time of a binary. For this purpose we use the byte-unixbench benchmarks. We used the execution time of un-randomized benchmarks as a baseline to compare with randomized benchmarks. We performed 20 randomizations per benchmark and took the average of these values. We observed that the execution time of the benchmarks was same before and after randomization and was not affected due to Marlin. This supports our initial hypothesis that the overhead is incurred only during the randomization phase of Marlin and after that the binary executes as a normal binary with no runtime overhead.

5.3 Comparison with Existing Defense Techniques

Table 2 compares Marlin with other approaches with respect to the defense techniques, the properties, and the metrics. We compared these approaches based on nine comparison dimensions. First, we looked at the types of code reuse attacks mitigated by these techniques. Marlin defends against both return-based and jump-based attacks, unlike [19], [22] that can stop only return-based attacks. Next we look at the techniques adopted by these defense approaches. These techniques employ either some form of diversification or use execution monitoring. For diversification based approaches that use code randomization, the randomization can be performed at various granularities. While randomizing at finer granularity such as instruction level [35], [38] increases entropy, it decreases the runtime performance by breaking memory locality. Control flow integrity [24], [25] and other approaches [19],

[22], [43] that monitor runtime execution also incur significant runtime overhead. Marlin adopts a tradeoff approach by using a coarse-granularity of randomization that achieves high entropy with no runtime overhead.

These defense techniques can be applied at different stage of software cycle such as installation, loading, execution stage etc. Marlin is applied at application load time which provides the advantage of frequent randomization without affecting runtime performance. Techniques such as [19], [22], [24], [25], [35], [43] that are deployed at execution stage incur a runtime overhead as shown in columns 4 and 7 in Table 2. Techniques that use runtime data structures also incur memory overhead in addition to runtime overhead. As shown in column 10, [22], [35], [43] use runtime data structure, while Marlin technique does not require any such additional data structure. Thus, Marlin incurs no runtime or memory overhead. These overheads for other approaches are indicated in columns 7 and 8. Some of the defense approaches such as [22], [38] use binary instrumentation to integrate their technique and this results in a increase in file size (see column 8). Marlin does not use any instrumentation and does not incur any space overhead.

We also compared Marlin with other approaches based on gadget displacement which measures how many gadgets are displaced in the target binary after applying diversification techniques. This metric is not applicable for techniques such as [19], [22], [24], [43] that do not diversify the binary, hence the gadget displacement is zero. Marlin displaces 99.78 percent of the gadgets in `.text` section by using function level randomization. Thus, we show that the coarse granularity randomization is sufficient and finer granularity randomization such as instruction level or basic block level randomization does not offer any additional benefits and in some cases may lead to unnecessary overheads.

6 Conclusion and Future Work

Our proposed solution to defend against code-reuse attacks was to increase the entropy by randomizing the function blocks. One may apply this randomization technique at various levels of granularity—function level, block level or gadget level. The level of granularity to choose is a trade off between security and performance. In our implementation, we implemented the randomization at the function level which is the most coarse granularity amongst the three mentioned above. However, we show that even this coarse level of granularity provides substantial randomization to make brute force attacks infeasible.

Our prototype implementation requires the binary disassembly to contain symbol information, i.e. a non-stripped binary. In practice however, binaries may be stripped and not contain the symbol information. We address this by using external tools such as *Unstrip* [39] that restore symbol information to a stripped binary. Another approach to process stripped binaries is to randomize at the level of basic blocks since they do not require function symbols to be identified. However, randomizing at basic block granularity will likely incur higher runtime overhead as it would break the principle of locality.

One limitation of Marlin is that it is unable to correctly rewrite certain binaries if these target binaries have certain compiler optimizations enabled or if they are obfuscated. This is because Marlin requires the $.text$ section in the target binary to be organized as function blocks and for these function block to be clearly identifiable using a disassembler.

In this work, we proposed a fine-grained randomization based approach to defend against code reuse attacks. This approach randomizes the application binary with a different randomization for *every run*. We have implemented a prototype of our approach and demonstrated that it is successful in defeating real ROP attacks crafted using automated attack tools. We have integrated this into a custom bash shell that randomizes a binary before executing it. We have also evaluated the effectiveness of our approach and showed that the brute force effort to attack Marlin is significantly high. Based on the results of our analysis and implementation, we argue that fine-grained randomization is both feasible and practical as a defense against these pernicious code-reuse based attack techniques.

Acknowledgment

The work reported in this paper has been partially supported by the Purdue Cyber Center, by the National Science Foundation under grants CNS-1111512 and CNS-1016722, and by a grant by Sypris Electronics.

References

[1] Aleph One, “Smashing the stack for fun and profit,” *Phrack Mag.*, vol. 49, no. 14, Nov. 1996.
[2] (2003). PaX Team. PaX [Online]. Available: http://pax.grsecurity.net/
[3] Solar Designer, “Getting around non-executable stack (and fix),” Aug. 1997, http://seclists.org/bugtraq/1997/Aug/63.
[4] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 552–561.
[5] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *Proc. 17th ACM Conf. Comput. Commun. Security*, 2010, pp. 559–572.
[6] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proc. 11th ACM Conf. Comput. Commun. Security*, 2004, pp. 298–307.
[7] G. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, “Surgically returning to randomized lib(c),” in *Proc. Annu. Comput. Security Appl. Conf.*, Dec. 2009, pp. 60–69.
[8] A. Gupta, S. Kerr, M. Kirkpatrick, and E. Bertino, “Marlin: A fine grained randomization approach to defend against ROP attacks,” in *Proc. 7th iNt. Conf. Netw. Syst. Security*, 2013, vol. 7873, pp. 293–306.
[9] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, “Marlin: Making it harder to fish for gadgets,” in *Proc. ACM Conf. Comput. Commun. Security*, 2012, pp. 1016–1018.
[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, “When good instructions go bad: Generalizing return-oriented programming to RISC,” in *Proc. 15th ACM Conf. Comput. Commun. Security*, 2008, pp. 27–38.
[11] R. Hund, T. Holz, and F. C. Freiling, “Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms,” in *Proc. 18th Conf. USENIX Security Symp.*, 2009, pp. 383–398.
[12] A. Francillon and C. Castelluccia, “Code injection attacks on harvard-architecture devices,” in *Proc. 15th ACM Conf. Comput. Commun. Security*, 2008, pp. 15–26.
[13] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, “Privilege escalation attacks on android,” in *Proc. 13th Int. Conf. Inf. Security*, 2011, pp. 346–360.
[14] T. Dullien, T. Kornau, and R.-P. Weinmann, “A framework for automated architecture-independent gadget search,” in *Proc. 4th USENIX Conf. Offensive Technol.*, 2010.

[15] P. Chen, X. Xing, B. Mao, and L. Xie, "Return-oriented rootkit without returns (on the x86)," in *Proc. 12th Int. Conf. Inf. Commun. Security*, 2010, pp. 340–354.

[16] T. Bletsch, X. Jiang, and V. Freeh, "Jump-oriented programming: A new class of code-reuse attack," Dept. Comput. Sci., North Carolina State Univ., Raleigh, NC, USA, Tech. Rep. TR-2010-8, 2010.

[17] E. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proc. 12th USENIX Security Symp.*, 2003, pp. 105–120.

[18] A. N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? The effectiveness of instruction set randomization," in *Proc. 14th Conf. USENIX Security Symp.*, 2005, vol. 14, p. 10.

[19] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in *Proc. 5th Int. Conf. Inf. Syst. Security*, 2009, pp. 163–177.

[20] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proc. 2007 ACM SIGPLAN Conf. Programming Language Design Implementation (PLDI '07)*, San Diego, California, USA, 2007, pp. 89–100.

[21] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *Proc. ACM Workshop Scalable Trusted Comput.*, 2009, pp. 49–54.

[22] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proc. 6th ACM Symp. Inf. Comput. Commun. Security*, 2011, pp. 40–51.

[23] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie, "Efficient detection of the return-oriented programming malicious code," in *Proc. 6th Int. Conf. Inf. Syst. Security*, 2010, pp. 140–155.

[24] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. 27th Annu. Comput. Security Appl. Conf.*, New York, NY, USA, 2011, pp. 353–362.

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

[26] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. 22nd USENIX Conf. Security*, 2013, pp. 337–352.

[27] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: Defeating return-oriented programming through gadget-less binaries," in *Proc. 26th Annu. Comput. Security Appl. Conf.*, 2010, pp. 49–58.

[28] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with 'return-less" kernels,' in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 195–208.

[29] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proc. Workshop New Security Paradigms*, 2010, pp. 7–16.

[30] T. Roeder and F. B. Schneider, "Proactive obfuscation," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, pp. 4:1–4:54, Jul. 2010.

[31] MSDN Microsoft./ORDER (Put Functions in Order) [Online]. Availabel: http://msdn.microsoft.com/en-us/library/00kh39zz.aspx

[32] MSDN Microsoft. Profile-guided optimizations [Online]. Available: http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx

[33] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proc. 22nd Annu. Comput. Security Appl. Conf.*, 2006, pp. 339–348.

[34] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. 14th Conf. USENIX Security Symp.*, 2005, vol. 14, p. 17.

[35] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 571–585.

[36] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 601–615.

[37] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Security*, 2013, pp. 299–310.

[38] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. ACM Conf. Comput. Commun. Security*, 2012, pp. 157–168.

[39] Paradyn Project. (2011). UNSTRIP [Online]. Available: http://paradyn.org/html/tools/unstrip.html

[40] (2011). byte-unixbench: A Unix benchmark suite [Online]. Available: http://code.google.com/p/byte-unixbench/

[41] J. Salwan. ROPgadget tool (2012). [Online]. Available: http://shell-storm.org/project/ROPgadget/

[42] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proc. IEEE Symp Security Privacy*, 2012, pp. 601–615.

[43] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Proc. 21st Annu. Netw. Distrib. Syst. Security Symp.*, 2014.

Aditi Gupta is currently working toward the PhD degree in the Department of Computer Science at Purdue University under the supervision of Dr. Elisa Bertino. Her research interests include access control, security in context aware systems, malware defense, and building secure architectures.

Javid Habibi is currently working toward the Master of Science (MS) degree in the Department of Computer Science at Purdue University.

Michael S. Kirkpatrick is currently an assistant professor of computer science at James Madison University. His main research interests include security engineering, access control, applied cryptography, privacy, and secure operating systems. Prior to earning graduate degrees from Purdue University and Michigan State University, he spent several years working in the field of semiconductor optical proximity correction for IBM Microelectronics (later Server & Technology Group) in Essex Junction, VT.

Elisa Bertino is a professor of computer science at Purdue University, and serves as a director of Purdue Cyber Center and Research Director of CERIAS. Prior to joining Purdue, she was a professor and the department head at the Department of Computer Science and Communication of the University of Milan. She has been a visiting researcher at the IBM Research Laboratory (now Almaden) in San Jose, at the Microelectronics and Computer Technology Corporation, at Rutgers University, at Telcordia Technologies. Her research interests include database security, digital identity management, policy systems, and security for web services.