Marlin: A fine grained randomization approach to defend against ROP attacks

Aditi Gupta[1], Sam Kerr[1], Michael S. Kirkpatrick[2], and Elisa Bertino[1]

[1] Purdue University, West Lafayette IN, USA,
{aditi,stkerr,bertino}@purdue.edu
[2] James Madison University, Harrisonburg VA, USA,
kirkpams@jmu.edu

Abstract. Code-reuse attacks, such as return-oriented programming (ROP), bypass defenses against code injection by repurposing existing executable code toward a malicious end. A common feature of these attacks is the reliance on the knowledge of the layout of the executable code. We propose a fine grained randomization based approach that modifies the layout of executable code and hinders code-reuse attack. Our solution, *Marlin*, randomizes the internal structure of the executable code, thereby denying the attacker the necessary *a priori* knowledge for constructing the desired sequence of gadget addresses. Our approach can be applied to any ELF binary and every execution of this binary uses a different randomization. Our work shows that such an approach is feasible and significantly increases the level of security against code-reuse based attacks.

1 Introduction

The evolution of software exploits, such as buffer overflows and string format vulnerabilities, shows a pattern of an arms race. On one side, stack smashing attacks gave way to heap-based code injection. Defenders countered with canary words, instruction set randomization, base address randomization, and related techniques [9, 10, 28, 1]. Attackers found ways to bypass these defenses [34, 33] and jump to their injected malicious code. Defenders then responded with Write-or-Execute ($W \oplus X$), which prevents the execution of injected code. To get around $W \oplus X$, return-into-*libc* and return-oriented programming (ROP) [31, 5] attacks were launched that leverage *existing* code rather than injecting their own. In the former case, a corrupted return address is used to jump to a *libc* function, such as `system`. In the latter, the attacker strings together *gadgets* (small sequences of binary instructions) to perform arbitrary computation.

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), only the start address of the code segment is randomized. However, 32-bit machines provide insufficient entropy, as there are only 2^{16} possible starting addresses, making the system vulnerable to brute-force [32]. 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 the area of 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 [30].

Our approach is to revisit the granularity at which randomization is performed. Rather than randomizing only a single parameter, our technique (*Marlin*) breaks an application binary into blocks of code and shuffles them. This significantly increases the entropy of the system; for instance, an application with 500 code blocks allows for $500! \approx 2^{3767}$ permutations, making brute-force infeasible. Our approach, which can be applied to any ELF binary without requiring source code, is performed transparently at load time to ensure every execution instance is unique. Finally, by paying a (quite reasonable) performance cost up front, Marlin avoids the overhead of on-going monitoring of critical data, such as return addresses, which other systems impose.

We are not the only researchers to have investigated software diversity as ROP attack mitigation. While Section 2.2 offers a detailed comparison, existing approaches suffer from one or more of the following limitations. First, diversification is not done frequently enough. Second, source code or other additional information is required. Third, the granularity of randomization is insufficient, leading large code chunks unrandomized. Fourth, on-going monitoring imposes significant run-time overhead by introducing additional data structures. Marlin provides strong and efficient defense while addressing these limitations.

After surveying code-reuse attacks and defenses in Section 2, we describe our design of Marlin in Section 3. Section 4 discusses our prototype, which consists of an off-line tool to randomize the binary image of an executable. We have constructed working ROP exploits and confirmed that these fail after shuffling the binary image using Marlin. Section 5 shows the results of various evalution experiments. Our evaluation of the time to randomize compiled binaries of the SPEC CPU2006 benchmark suite shows the average performance penalty is reasonable. Section 6 highlights both the merits and limitations of Marlin, and we conclude in Section 7.

2 Background & Related Work

The focus of our work is on ROP attacks, which are a special case of code-reuse attacks that leverage existing code in the application binary to execute arbitrary instructions. In this section, we start with a brief summary of these attack techniques and existing defenses. We then summarize critical factors of code-reuse attacks and define our threat model.

2.1 Return-oriented programming

Return-oriented programming (ROP) 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 [4] and unaligned CISC architectures [31]. ROP techniques can be used to create rootkits [19], can inject code into Harvard architectures [16], and have been used to perform privilege escalation in Android [12]. Initiating a ROP attack is made even easier by the availability of architecture-independent algorithms to automate gadget creation [15]. Additionally, the same technique of stringing together gadgets has been used to manipulate other instructions, such as `jmp` and their variants [5, 8, 3].

2.2 Defenses

Address obfuscation [1], ASLR (*e.g.,* PaX [28]), and Instruction Set Randomization (ISR) aim to defend against code-reuse attacks by introducing randomness into processes' memory images. They randomize with coarse granularity and are subject to brute force attacks [33, 32], especially on 32-bit architectures. While upgrading to 64-bit increases the randomization, information leakage can allow an attacker to bypass the defense [30]. Furthermore, for some settings (*e.g.,* embedded devices), upgrading to 64-bit is simply not feasible. While [1] suggests randomizing function blocks as a potential technique (which we employ in Marlin), no further implementation, discussion, or evaluation was attempted.

Researchers have also considered dynamic monitoring defenses. For instance, DROP [6] dynamically compares the execution of `ret` instructions with statistically defined normal program behavior. DynIMA [13] combines TPM memory measurement capabilities with dynamic taint analysis to monitor process integrity. Other approaches store sensitive data (*e.g.,* return addresses) in a protected shadow stack [14, 7]. These techniques impose a non-zero performance cost for every checked instruction, yielding non-trivial cumulative overhead. In contrast, Marlin imposes a one-time cost at process start-up and no additional on-going penalty.

Other approaches introduce randomness at compile time. For instance, compilers can be modified to generate code without `ret` instructions [25, 22]. These mechanisms, however, fail to handle attacks leveraging `jmp` instructions; furthermore, if a new type of gadget is proposed, the compiler would have to be modified yet again. Alternatively, app store-based diversification [17] and linkage techniques for performance optimization [23, 24] can be applied to produce unique executables. However, these techniques do not stop an attacker with a known singular target image, do not help legacy systems, and, in the case of the former, rely on centralized control of software deployment. In contrast, proactive obfuscation [29] applies a semantics-preserving transformation to compiled server applications. Marlin is similar to this work in spirit, but the former aimed at diversifying replicas in distributed systems; as such, their threat model and techniques differed from our own.

Other techniques similar to Marlin have also been proposed to randomize processes. ASLP [21] rewrites the ELF headers and shuffles sections, functions, and variables. As such, ASLP requires relocation information (or recompilation of source code), as well as user input. In contrast, as Marlin randomizes function *blocks* within the text segment, this additional information is not necessary. Bhatkar *et al.* [2] associates a pointer with every function and adds a layer of indirection to every function call. Unlike Marlin, the function reordering is not done at load time. ILR [18] randomizes the location of every instruction and uses a process-level virtual machine, which imposes a significant on-going performance cost, to find the called code. Pappas *et al.* [26] use in-place randomization that probabilistically breaks 80% of the useful gadgets. However, by shuffling the entire memory image, Marlin provides stronger guarantees, probabilistically breaking all sequences. Furthermore, [18] and [26] do not randomize the binary at *every execution*, which Marlin does. XIFER [11] is very similar to Marlin, except that it provides randomization at the basic block granularity, rather than at the function level. The finer granularity incurs more overhead than Marlin; however, we show that block-level randomization is sufficient to defeat brute force attacks, and the additional granularity is unnecessary.

2.3 Enabling factors for code-reuse attacks

Based on our survey of ROP attacks and defenses, we have identified a number of distinct characteristics and requirements for a successful exploit. We argue that a defensive technique that undermines these invariants will present a robust protection mechanism against these threats. 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.

2.4 Threat Model

The proposed defense, *Marlin*, is aimed to protect a vulnerable application against code reuse attacks, such as ROP attacks. This application may have a buffer overflow vulnerability that can be leveraged by an attacker to inject an exploit payload. The system is assumed to be protected using $W \oplus X$ policy 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 cannot 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.

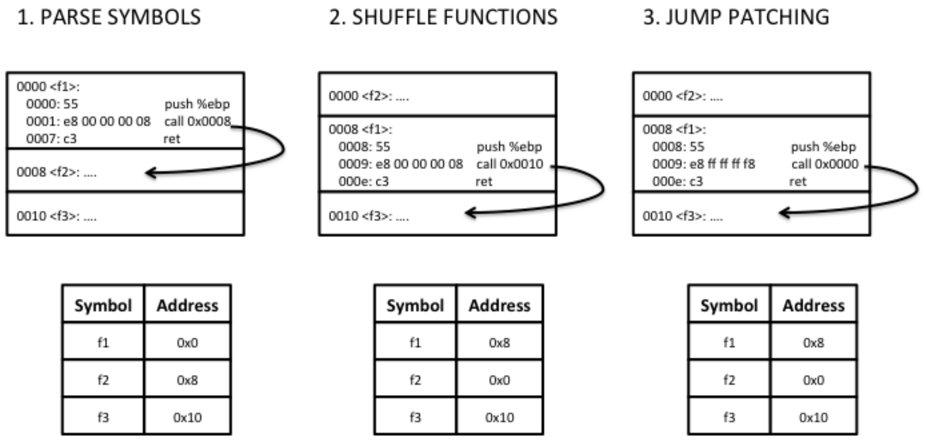
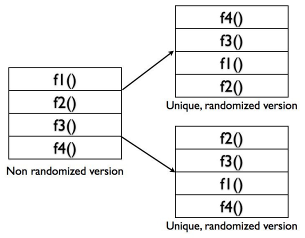
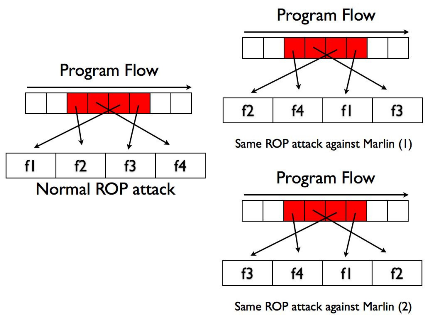


Fig. 1: Processing steps in Marlin



(a) Randomization technique



(b) Failure of ROP attacks

Fig. 2

3 Marlin

Code-reuse attacks make certain assumptions (as discussed in section 2.3) about the address layout of application's executable code and shared libraries. 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 by the current process image. This shuffling is performed at the granularity of function blocks. Marlin 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 Figure 2(a). Figure 2(b) 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 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. Preprocessing phase parses the ELF binary 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* [27]. Once the symbol information is restored and identified, we proceed on to the next stage of Marlin processing that randomizes the application binary.

3.2 Randomization algorithm

Once the function symbols have been identified, Marlin generates a random permutation of this set of symbols. The resulting permutation determines the order in which the `mmap` system calls are issued, which changes the order of the mapped symbols in memory. The function blocks are then shuffled around according to this random permutation. Shuffling the code blocks in an application binary changes the relative offsets between instructions that may affect various jump instructions. These jumps may be either absolute jumps or relative jumps. 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 code 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. While shuffling the blocks, padding is added when necessary to ensure that the resulting binary is page aligned. 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 table*. Note that this jump patching table is discarded before the application is given control, thus preventing attacker from utilizing this information to derandomize the memory layout.

In the second stage, the actual jump patching is done where the algorithm examines the jump table for every jump that needs to be patched. Whenever a relative jump is encountered, it is the algorithm executes `PatchJump()` to redirect the jump to the correct address in the binary. `PatchJump()` method takes the current address of the jump, the address of the destination function, and any offset into the destination function to determine how far away the destination is. Then, it overwrites the original jump with this new offset, so that the jump points to the correct address.

Marlin breaks the basic assumption required by code reuse attacks as mentioned in section 2.3. The run-time shuffling of the code blocks prevents multiple instances of the same program from having the same address layout. 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 attacker. We now discuss the security guarantees offered by Marlin.

Algorithm 1: Code Randomization algorithm

```
Input  : A non-shuffled program, $P$
Output: A shuffled program, $P_S$

$P_S = P$
$L$ = All symbols in $P$.
$F$ = A list of forbidden symbols that should not be shuffled
$L = L - F$
/* Swapping stage */
for Every symbol $S \in L$ do
    $R$ = Randomly select another symbol in $L$
    Swap $S$ and $R$ in $P_S$
    $S.A$ = The previous location of $R$
    $R.A$ = The previous location of $S$
/* Jump patching stage */
for Every symbol $S \in L$ do
    for Every jump $J \in S$ do
        $J.A = S.A$ + the offset of the jump from the start of $S$
        $J.D$ = The destination address of the jump
        $J.S$ = The symbol that J is jumping into
        if $J.D$ is a relative jump to within $S$ then
            /* No action needed */
        else if $J.D$ is a relative jump to outside $S$ then
            $J.D = J.D - S.A + (J.S).A$
            PatchJump($J.A$, $J.D$)
        else if $J.D$ is an absolute jump then
            PatchJump($J.A$, $J.D$)
```

3.3 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 memory access 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 after the k^{th} 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, $P(k)$ is constant ($= \frac{1}{N}$) and independent of k. Let $P(k) = p$, $\forall\ k$. Then, the average number of attempts before the attack is successful for the first time is

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

Thus, the attacker would have to make an average $n!$ number of attempts to correctly guess the randomized layout and launch 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 first success is achieved at k^{th} 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

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

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 $500! = 2^{3767}$ number of attempts will be required for the first case. This is clearly computationally infeasible. A more extensive evaluation performed using SPEC2006 benchmarks is presented later in Section 5 that demonstrates the effectiveness of our technique.

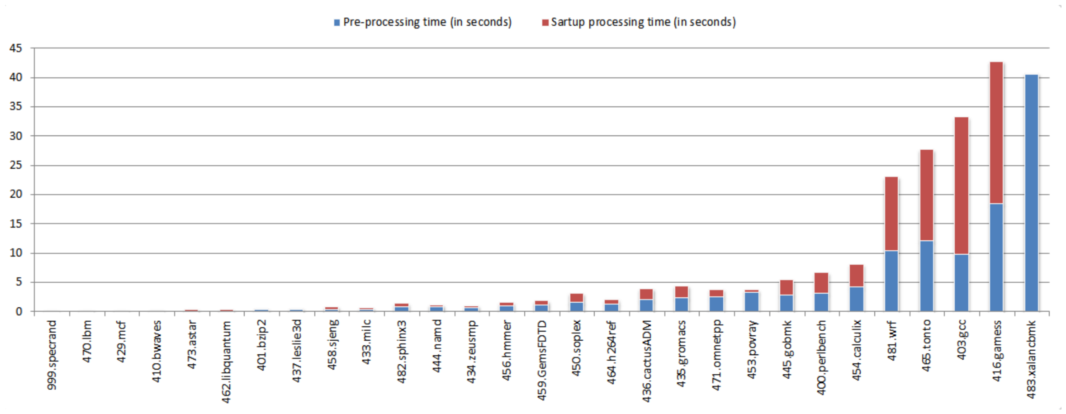


Fig. 3: Overhead measurements of Marlin

4 Prototype Implementation

We have build a prototype implementation of Marlin that can operate on any ELF binary and does not require their source code. As a pre-processing step, we use `objdump` utility to obtain a disassembly listing of the application binary that contains the program instructions as well as its internal symbols. These listings are then used to generate a set of parameter files. These parameter files contain a list of symbols (functions) present in the binary, as well as their starting addresses and lengths. Another file is created which lists the addresses where the relative jumps inside functions of interest are located. It is important to note that not every function is considered to be a function of interest, since randomizing certain functions, such as `_start`, will render the binary inoperable. These parameter files are used as input in the next phase of processing that performs the shuffling and jump patching operations. Upon completion, the parameter files are deleted and the the new "marlinized" binary is ready to be run like a normal executable binary.

5 Evaluation

We now describe various experiments to evaluate our *Marlin* prototype. These experiments test three aspects of Marlin. First, we show that Marlin successfully defends against a ROP attack. Second, we study the brute force effort that would be required to circumvent the protection offered by Marlin. Third, we evaluate the processing costs incurred by using Marlin. The experiments were performed on a Linux machine with Intel Core i7 3.40GHz CPU and 8GB RAM. This machine had ASLR and $W \oplus X$ protection enabled while the experiments were being performed. We used SPEC CPU2006 benchmarksto conduct the various experiments.To launch attacks against Marlin-protected binary, we use ROP-

Benchmark	**Number of Symbols**	**Time for one attempt (sec)**	**Avg. # of attempts for successful attack**[†]	**Avg. time (sec) for successful attack**[†]
999.specrand	10	0.126	3.63×10^{6}	4.57×10^{5}
470.lbm	26	0.269	4.03×10^{26}	1.08×10^{26}
429.mcf	31	0.269	8.22×10^{33}	2.21×10^{33}
410.bwaves	14	0.385	8.72×10^{10}	3.36×10^{10}
473.astar	97	0.559	9.62×10^{151}	5.38×10^{151}
462.libquantum	106	0.529	1.15×10^{170}	6.06×10^{169}
401.bzip2	79	0.691	8.95×10^{116}	6.18×10^{116}
437.leslie3d	29	1.164	8.84×10^{30}	1.03×10^{31}
458.sjeng	142	1.703	2.69×10^{245}	4.59×10^{245}
433.milc	242	1.408	2.37×10^{473}	3.34×10^{473}
482.sphinx3	335	2.302	1.16×10^{702}	2.66×10^{702}
444.namd	142	2.681	2.69×10^{245}	7.23×10^{245}
434.zeusmp	83	2.399	3.95×10^{124}	9.47×10^{124}
456.hmmer	502	3.269	3.07×10^{1139}	1.00×10^{1140}
459.GemsFDTD	102	4.217	9.61×10^{161}	4.05×10^{162}
450.soplex	918	5.386	1.22×10^{2323}	6.59×10^{2323}
464.h264ref	531	5.532	1.50×10^{1218}	8.31×10^{1218}
436.cactusADM	1299	8.347	2.43×10^{3482}	2.03×10^{3483}
435.gromacs	1098	10.373	4.42×10^{2863}	4.59×10^{2864}
471.omnetpp	2023	7.594	3.19×10^{5811}	2.42×10^{5812}
453.povray	1633	9.039	4.06×10^{4539}	3.68×10^{4540}
445.gobmk	2547	32.019	1.29×10^{7571}	4.12×10^{7572}
400.perlbench	1730	12.625	3.22×10^{4852}	4.07×10^{4853}
454.calculix	1324	18.509	2.16×10^{3560}	3.99×10^{3561}
481.wrf	2883	47.437	6.17×10^{8724}	2.93×10^{8726}
465.tonto	4096	51.328	[**] $> 1.97\times10^{9997}$	[**] $> 1.01\times10^{9999}$
403.gcc	4623	50.28	[**] $> 1.97\times10^{9997}$	[**] $> 9.92\times10^{9998}$
416.gamess	2893	91.312	2.49×10^{8759}	2.28×10^{8761}
483.xalancbmk	13848	42.903	[**] $> 1.97\times10^{9997}$	[**] $> 8.47\times10^{9998}$

Table 1: Brute force effort

[†] These correspond to the average number of attempts for Case 1 in section 3.3. The values for Case 2 will be approximately half of the value for Case 1.
[**] We were unable to compute factorial for values larger than 3248. The value used in these columns is 3248!.

gadget [3] [20], an attack tool that automatically creates exploit payload for ROP attacks by searching for gadgets in an application's executable section.

Effectiveness 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

[3] ROPgadget v3.3.3 was used for these experiments.

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 strongly operate under the assumption of a static address layout of executable code. Also, notice that 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 on the unprotected test application.

Attacks on Marlin In section 3.3, we computed the average number of attempts required to successfully attack a "marlinized" binary. We performed an extensive evaluation of this using SPEC CPU 2006 benchmarks. Table 1 shows the number of brute force attempts and the time it takes to craft one exploit. We noticed that around 80% of these benchmarks have more than 80 symbols (indicating an effort of 80! attempts). We observed an average of 1496 symbols and a median of 502 symbols present in these applications. Thus, the number of brute force attempts in a general case can be approximated to $500! \approx 2^{3767}$ attempts which is quite significant. Also, on an average, we observed the time to compute one attack payload as 14.3 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 gadgets 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.

Overhead Analysis We evaluated the efficiency of Marlin by measuring the overhead incurred while loading an application. We use SPEC CPU2006 benchmarks to conduct this performance evaluation. 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 and referred to as *preprocessing* phase. Next phase involves shuffling the code blocks and patching the jumps. This computation is referred to as *startup processing* phase.

Figure 3 shows the overhead incurred during preprocessing and startup processing phase respectively. The benchmark 483.xalancbmk took significantly longer time to process. This is because it contained 13848 symbols in contrast to a median of 500 symbols by other applications. The average time taken by preprocessing phase was 4.2 seconds, while average time taken by the startup processing phase was 3.3 seconds. It is quite evident from these numbers, that the preprocessing phase is the major contributor to these performance costs.

Since preprocessing phase is independent of individual randomizations, it can be executed just once per application and the results can be stored in database. The randomization phase, that runs with every execution, can read and process information from this database. This simple optimization can greatly improve efficiency of Marlin. Also, the performance hit due to Marlin is incurred only at the load time of the application. Once the application binary has been randomized, it executes like a normal application binary.

6 Discussion

Our proposed solution to defend against code-reuse attacks was to increase the entropy by randomizing the code 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 names, 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* [27] 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 don't require symbol information to be identified. Moving forward, we will explore using basic block level instead of function level as the unit of randomization for Marlin.

7 Conclusion

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 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.

References

1. Bhatkar, E., Duvarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: In Proc. of the 12th USENIX Security Symposium. pp. 105–120 (2003)

2. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: Proc. of the 14th conference on USENIX Security Symposium - Volume 14. pp. 17–17. SSYM'05 (2005)
3. Bletsch, T., Jiang, X., Freeh, V.: Jump-oriented programming: A new class of code-reuse attack. Tech. Rep. TR-2010-8, North Carolina State University (2010)
4. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to risc. In: Proc. of the 15th ACM conference on Computer and communications security. pp. 27–38 (2008)
5. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proc. of the 17th ACM conference on Computer and communications security. pp. 559–572 (2010)
6. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: DROP: Detecting return-oriented programming malicious code. In: Proc. of the 5th International Conference on Information Systems Security. pp. 163–177 (2009)
7. Chen, P., Xing, X., Han, H., Mao, B., Xie, L.: Efficient detection of the return-oriented programming malicious code. In: Proc. of the 6th international conference on Information systems security. pp. 140–155 (2010)
8. Chen, P., Xing, X., Mao, B., Xie, L.: Return-oriented rootkit without returns (on the x86). In: Proc. of the 12th international conference on Information and communications security. pp. 340–354 (2010)
9. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: Pointguard: Protecting pointers from buffer overflow vulnerabilities. In: In Proc. of the 12th Usenix Security Symposium (2003)
10. Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: In Proc. of the 7th USENIX Security Symposium. pp. 63–78 (1998)
11. Davi, L., Dmitrienko, A., Nürnberger, S., Sadeghi, A.R.: Xifer: A software diversity tool against code-reuse attacks. In: 4th ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3 2012) (Aug 2012)
12. Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M.: Privilege escalation attacks on android. In: Proc. of the 13th international conference on Information security. pp. 346–360 (2011)
13. Davi, L., Sadeghi, A.R., Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In: Proc. of the 2009 ACM workshop on Scalable trusted computing. pp. 49–54 (2009)
14. Davi, L., Sadeghi, A.R., Winandy, M.: ROPdefender: a detection tool to defend against return-oriented programming attacks. In: Proc. of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 40–51 (2011)
15. Dullien, T., Kornau, T., Weinmann, R.P.: A framework for automated architecture-independent gadget search. In: Proc. of the 4th USENIX conference on Offensive technologies. WOOT'10 (2010)
16. Francillon, A., Castelluccia, C.: Code injection attacks on harvard-architecture devices. In: Proc. of the 15th ACM conference on Computer and communications security. pp. 15–26 (2008)
17. Franz, M.: E unibus pluram: massive-scale software diversity as a defense mechanism. In: Proc. of the 2010 workshop on New security paradigms. pp. 7–16. NSPW '10 (2010)
18. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: Ilr: Where'd my gadgets go? In: Proc. of the 2012 IEEE Symposium on Security and Privacy. pp. 571–585 (2012)

19. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: Proc. of the 18th conference on USENIX security symposium. pp. 383–398. SSYM'09 (2009)
20. Jonathan Salwan: ROPgadget tool. `http://shell-storm.org/project/ROPgadget/`
21. Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: Proc. of the 22nd Annual Computer Security Applications Conference. pp. 339–348 (2006)
22. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with "return-less" kernels. In: Proc. of the 5th European conference on Computer systems. pp. 195–208 (2010)
23. MSDN Microsoft: /ORDER (Put Functions in Order). `http://msdn.microsoft.com/en-us/library/00kh39zz.aspx`
24. MSDN Microsoft: Profile-guided optimizations. `http://msdn.microsoft.com/en-us/library/e7k32f4k.aspx`
25. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: defeating return-oriented programming through gadget-less binaries. In: Proc. of the 26th Annual Computer Security Applications Conference. pp. 49–58 (2010)
26. Pappas, V., Polychronakis, M., Keromytis, A.D.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: IEEE Symposium on Security and Privacy. pp. 601–615 (2012)
27. Paradyn Project: UNSTRIP. `http://paradyn.org/html/tools/unstrip.html` (2011)
28. PaX Team: PaX. `http://pax.grsecurity.net/`
29. Roeder, T., Schneider, F.B.: Proactive obfuscation. ACM Trans. Comput. Syst. 28, 4:1–4:54 (July 2010)
30. Roglia, G., Martignoni, L., Paleari, R., Bruschi, D.: Surgically returning to randomized lib(c). In: Computer Security Applications Conference, 2009. ACSAC '09. Annual. pp. 60 –69 (dec 2009)
31. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proc. of the 14th ACM conference on Computer and communications security. pp. 552–561. ACM (2007)
32. Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proc. of the 11th ACM conference on Computer and communications security. pp. 298–307 (2004)
33. Sovarel, A.N., Evans, D., Paul, N.: Where's the feeb? the effectiveness of instruction set randomization. In: Proc. of the 14th conference on USENIX Security Symposium - Volume 14. pp. 10–10 (2005)
34. Tyler Durden: Bypassing PaX ASLR protection. Phrack Magazine 59(9) (June 2002)