Remix: On-demand Live Randomization

Yue Chen, Zhi Wang, David Whalley
Florida State University
{ychen,zwang,whalley}@cs.fsu.edu

Long Lu
Stony Brook University
long@cs.stonybrook.edu

ABSTRACT

Code randomization is an effective defense against code reuse attacks. It scrambles program code to prevent attackers from locating useful functions or gadgets. The key to secure code randomization is achieving high entropy. A practical approach to boost entropy is on-demand live randomization that works on running processes. However, enabling live randomization is challenging in that it often requires manual efforts to solve ambiguity in identifying function pointers.

In this paper, we propose Remix, an efficient and practical live randomization system for both user processes and kernel modules. Remix randomly shuffles basic blocks within their respective functions. By doing so, it avoids the complexity of migrating stale function pointers, and allows mixing randomized and non-randomized code to strike a balance between performance and security. Remix randomizes a running process in two steps: it first randomly reorders its basic blocks, and then comprehensively migrates live pointers to basic blocks. Our experiments show that Remix can significantly increase randomness with low performance overhead on both CPU and I/O intensive benchmarks and kernel modules, even at very short randomization intervals.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

ASLR; Code Reuse Attack Defense; Live Randomization

1. INTRODUCTION

With the ubiquitous deployment of data execution prevention (DEP) that can foil direct code injection [20, 21, 24], code reuse attacks have become a popular attack method. Instead of injecting foreign code, they reuse existing code to bypass DEP. These attacks could reuse either whole functions (e.g., return-to-libc or return-to-plt) or short code fragments called gadgets (e.g., return-oriented programming [12, 15, 45] or jump-oriented programming [11]). In a typical scenario, the attacker first launches a code-reuse attack to disable DEP by calling functions like `mprotect`, and then injects the malicious code into the victim process for more complex tasks. Control flow integrity (CFI) is an effective defense against code reuse attacks [1]. CFI guarantees that the runtime control flow follows the static control flow graph (CFG). Consequently, the attacker cannot arbitrarily manipulate the control flow to reuse the existing code. However, CFI has not been widely adopted. Early CFI systems have high performance overhead because CFI requires to instrument every indirect branch instruction. Recent implementations improve the performance by sacrificing preciseness [55, 54] and, in some cases, security [23, 28].

Code randomization is another effective defense against code reuse attacks. Unlike CFI, code randomization scrambles the reusable code by randomizing the code location, the code layout, or the instruction encoding [7, 22, 30, 33, 42, 49]. Many code reuse attacks rely on the exact locations or contents of the victim process. Code randomization causes these attacks to behave unpredictably. Most popular operating systems support a simpler form of code randomization called address space layout randomization (ASLR), in which (position-independent) executables are loaded at random base addresses [3, 5, 37]. ASLR offers limited randomness, especially on the 32-bit architectures [46]. Moreover, ASLR is particularly vulnerable to information leak attacks – a single leaked code or data pointer can de-randomize the whole process since every code section has a fixed offset to the base. To address this problem, fine-grained code randomization techniques have been proposed, for example, to rearrange functions [33], basic blocks [49], or instructions [30, 42]. A high entropy is the key to the security of code randomization.

One effective boost to randomness is on-demand live randomization. Live randomization works on a live, running process. It can be applied many times at undetermined periods of time, making the process a moving target for the attacker. Live randomization can eliminate the predictability associated with the compile-time or load-time randomization schemes. It can significantly improve the randomness for 32-bit architectures, which many computers and embedded devices still use. However, live randomization is challenging to implement correctly: when the code is changed, it is necessary to update all the code and the data that depend on the changed code to guarantee correctness. For example, if a `call` instruction is moved to a different address, we have to update every branch instruction that targets this

CODASPY'16, March 09 - 11, 2016, New Orleans, LA, USA

ISBN 978-1-4503-3935-3/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2857705.2857726

instruction (or its preceding instructions), and search the stack for the corresponding return address and update it to the new one. Run-time changes to function entry points are even harder to fix – it is non-trivial to locate all the affected function pointers in the whole address space, including the code, the data, the stacks, and the heap. In particular, a linear search of the function address has false positives and could take a prohibitively long time to complete. Function addresses could also be stored by and in the OS kernel. For example, a process can register a handler for each signal of interest. The kernel saves this data in the kernel memory, unreachable by the process. If the handler is moved, the kernel must be notified with the updated address. To achieve that, one has to intercept the system calls that register signal handlers and re-register the handlers when necessary. Therefore, live randomization is challenging to implement. An existing live-randomization system customizes the compiler to generate enough meta-data to facilitate its job [27]. However, it still has yet to overcome the aforementioned challenges. For example, there is unsolvable ambiguity (e.g., pointers in unions or pointers stored as integers) in pointer migration that requires developers' manual effort.

In this paper, we propose Remix, an efficient, practical live randomization system for both user processes and kernel modules. Remix randomly shuffles the process' basic blocks *within* their respective functions to change the run-time code layout (a basic block is a linear sequence of instructions that has only one entry point and one exit point [50]. An exit point is often a branch instruction, such as `jmp` or `ret`. It could also be a non-branch instruction that falls through to the next basic block.) That is, functions remain at their original, expected locations, while basic blocks are moved around but never cross the function boundaries. This design can significantly reduce the complexity of live randomization: *first*, there is no need to fix function pointers because function entry points are not moved. This avoids the complicated pointer migration that may involve unresolvable ambiguity [27] (function addresses are still randomized once at the load time by ASLR). Basic block addresses may still appear in both the code and data sections (e.g., jump tables). But these appearances are mostly limited to the local scopes and thus are relatively easy to fix. *Second*, it is straightforward to support partial randomization since each change is confined to a local scope. For example, Remix can be used to randomize selected kernel modules. Randomized and non-randomized kernel modules can co-exist in a single kernel in harmony. *Third*, compared to systems that globally rearrange basic blocks [49], Remix maintains better locality. Compilers make an effort to optimally lay out the code for better performance. Global rearrangement of basic blocks could potentially lead to poor locality and substantial performance loss. Remix instead shuffles basic blocks locally. It can also bundle closely-related basic blocks together (e.g., tight loops) to further reduce the performance overhead. Simplicity and efficiency are two major advantages of Remix. They make Remix an ideal technique to compose with other defenses. For example, Remix should be used with ASLR so that functions are randomized at least once (during program startup). Other examples of compatible techniques include defenses against JIT-ROP [47] or Blind-ROP [10] attacks [6, 19, 26] and function-level re-randomization [27]. Remix can significantly increase the unpredictability of those systems with on-demand, live randomization of basic blocks.

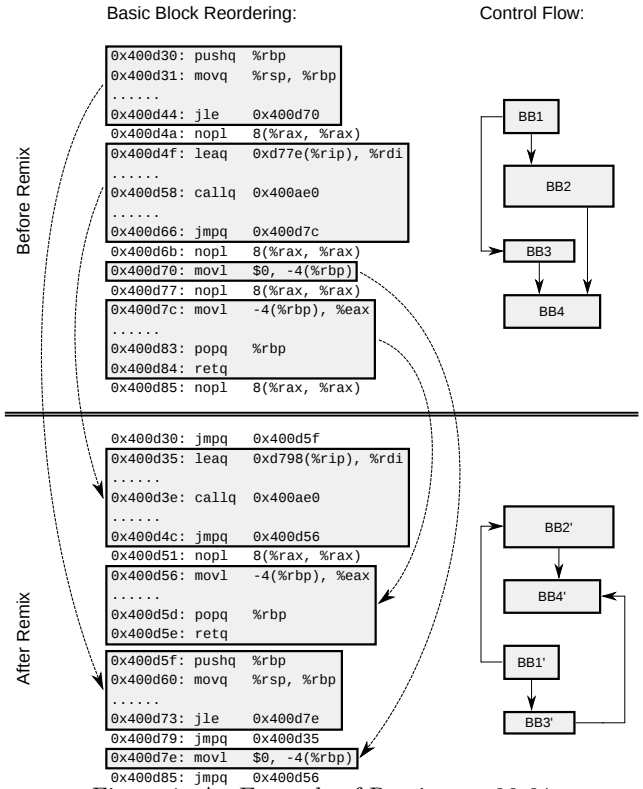

Figure 1: An Example of Remix on x86-64

We have implemented a prototype of Remix for Linux applications and FreeBSD kernel modules. Our prototype uses a slightly modified LLVM compiler to reserve spaces needed for basic block reordering (it can also support binary-only programs by leveraging existing `NOP` instructions used to align instructions, albeit with less gain in randomness.) Our experiments with standard benchmarks and applications show that Remix can substantially improve the randomness with a minor overhead (e.g., 2.8% average performance overhead and 14.8% average increase in binary size for SPEC CPU2006).

The rest of the paper is organized as the following. We first present the design and implementation of Remix in Section 2 and Section 3, respectively. The evaluation results are given in Section 4, followed by a discussion of potential improvements to Remix in Section 5. Finally, we compare Remix to the state-of-the-art in Section 6 and conclude the paper in Section 7.

2. DESIGN

2.1 Overview

Remix aims at increasing randomness for protected processes through live randomization of basic blocks while keeping function entry points unmoved. Figure 1 shows an example of applying Remix to a simple 64-bit x86 (x86-64) function. After Remix, the basic blocks have been reordered. Any gadgets discovered before Remix immediately become obsolete, and their execution likely will cause exceptions like illegal opcode or general protection error. Even though it is conceptually straightforward, reordering basic blocks faces a number of challenges:

First, the function might not have enough space to accommodate the reordered basic blocks. For example, some basic blocks end with a short jump instruction that takes a single byte of offset. Their targets could be moved by Remix beyond the reach of one byte. It is thus necessary to substitute the short jump with a long jump, which takes four bytes for the offset. In addition, some basic blocks do not end with a branch instruction. They instead fall through to the next basic block. The `movl $0, -4(%rbp)` instruction in Figure 1 (at address `0x400d70`, before Remix) is such an example. The instruction at `0x400d7c` starts a new basic block because the instruction at `0x400d66` jumps to it, making it an entry point. Remix has to add a new jump instruction to connect the fall-through basic blocks. To accommodate reordered basic blocks, we modify the compiler to emit a five-byte NOP instruction after each basic block. This provides enough space to insert a long jump (also five bytes) for each basic block. This errs on the safe side – there is always enough space to accommodate the reordered basic blocks. Remix can also support binary-only programs without recompilation by leveraging the existing NOP instructions in functions.

Second, when a basic block or its succeeding blocks are moved to other positions, it is necessary to fix their exit points to maintain the correct control flow: if the exit instruction is a direct branch, we only need to update its offset to the new address of its successors (a basic block has two successors if it is a conditional branch.) For example, the `jle` instruction (Figure 1) has two branches. When it is moved, Remix adds a direct `jmp` instruction (at `0x400d79` after Remix) because the original branch falls through to the `movl` instruction at `0x400d70`. If the exit instruction is an indirect branch, Remix analyzes its structure and handles it accordingly. For example, indirect calls can be left alone because function entry points are not moved by Remix. Indirect jumps are more complicated with several possibilities (Section 2.3.2). They are in fact related to the third challenge, how to migrate basic block pointers.

Third, there exist pointers to basic blocks in the process' code and data sections. For example, the stack consists of local variables and return addresses. A return address points to the instruction after the originating call instruction (the return site). If a call instruction is moved by Remix, we have to substitute the original return address on the stack with the new one. In addition, the compiler generates jump tables to speed up `switch`/`case`-like structures. A jump table contains basic block pointers to handle its cases. It has to be patched when basic blocks are moved. Jump tables have several possible structures. Remix must handle all those different cases. The kernel has its own set of basic block pointers that have to be converted to maintain the correct control flow. In the rest of this section, we present in detail how Remix solves these problems.

2.2 Basic Block Reordering

Remix shuffles basic blocks within their respective functions to increase run-time randomness. Algorithm 1 gives a high-level overview of this process. Specifically, Remix first parses the code into basic blocks, and generates a random ordering of these basic blocks to guide the process. Remix then lays out the basic blocks according to that ordering, and saves the mapping between their old and new positions in a table (m). This table is used to convert basic block pointers. Note that the first instruction of a function (i.e., the function entry point) is replaced by a direct jump to the first basic block. As shown in Figure 1, Remix does not terminate a basic block with a call instruction. We choose this design for two reasons: first, Remix keeps functions at their original locations. Call instructions thus do not require complicated handling. Second, by design, a call instruction falls through to the next instruction. An extra jump must be inserted after the call instruction if the fall-through instruction is moved by Remix. Many applications use a large number of call instructions. This would substantially increase the binary size and reduce the performance.

Algorithm 1 Basic Block Reordering

```
1: for each function f do
2:     s = GenerateRandomOrder(f);
3:     m = LayoutBasicBlocks(s);
4:     for each instruction i do
5:         if i ∈ DirectCall then
6:             FixDispS(i, m);
7:         else if i ∈ DirectJump then
8:             FixDispS(i, m);
9:             addr = CalcPrevTarget(i);
10:            FixDispD(i, m, addr);
11:        else if i ∈ IndirectJump then
12:            if IsJumpTable(i) then
13:                AddToJumpTableList(jt, i);
14:            end if
15:        end if
16:        if i ∈ PC-RelativeInsn then
17:            FixDispS(i, m);
18:        end if
19:    end for
20:    ConvertBasicBlockPointers (m, jt);
21: end for
```

Reordering basic blocks changes their positions. Some instructions need to be updated to maintain the original control flow. They consist of instructions that have a program-counter (PC) relative operand (e.g., the various branch instructions). Most of them have a constant displacement that can be adjusted to offset the position changes made by Remix. We need to consider two types of position changes – the instruction itself and the destination of the instruction. We use two functions, `FixDispS` and `FixDispD`, in Algorithm 1 to handle these two cases, respectively. The majority of the instructions to be patched are branch instructions, i.e., indirect/direct calls and jumps (line 5-15 in Algorithm 1):

- **Indirect Call:** an indirect call invokes a function indirectly through a function pointer. Function pointers remain valid because Remix does not move function entry points. As such, indirect calls can be left unchanged.
- **Direct Call:** a direct call targets the function at a certain displacement to itself. Even though the function stays at its position in the memory, the call instruction could have been moved to a different place. Accordingly, direct calls should be fixed with the `FixDispS` function.
- **Direct Jump:** a direct jump often targets another basic block. Both the source and the destination instructions

might change the positions. To fix a direct jump, Remix first adjusts the instruction's displacement to offset the source instruction movement with `FixDispS`. It then calculates the original target and adjusts the displacement to offset the target instruction movement. In addition, a conditional jump has two branches, one for the true condition and the other for the false condition. One of the branches is a fall-through to the next instruction. Remix handles this case by treating the fall-through as an implicit jump to the next basic block. The same approach is applied if a basic block falls through to the next one without a branch instruction (e.g., BB3 in Figure 1).

- **Indirect Jump:** indirect jumps are more complicated to handle than the other branch instructions. They can target both functions and basic blocks. The former does not need any changes, but the latter can involve several different cases that must be handled by Remix. We elaborate these cases in Section 2.3.2.
- **PC-relative Addressing Mode:** in addition to branch instructions, we also need to patch instructions with the PC-relative addressing mode, which are often used by the compiler to generate position-independent code. A program must be compiled as a position-independent executable (PIE) to benefit from ASLR (on Linux). A PIE program can run at any location in the address space. To achieve that, it calculates the run-time addresses of its code and data relative to the current program counter. The newer x86-64 architecture natively supports the PC-relative addressing mode. For example, instruction `lea 0x200000(%rip),%rbp` adds `0x20,0000` to the current program counter and saves it to the `rbp` register. The older x86-32 architecture has no native support for this addressing mode. Instead, the compiler uses a simple built-in function to retrieve the return address from the stack, which has been pushed to the stack earlier by the caller. Accordingly, this function returns the address of the return site (i.e., PC+5). To ensure correctness, Remix needs to update these instructions and functions. Fortunately, the compiler uses this mode (almost) exclusively to calculate the run-time function and data addresses, both of which are not changed by Remix. Only the PC-relative instructions and functions (on the x86-32 architecture) may have been moved. This can be easily compensated with `FixDispS`.

When updating instructions, the new displacement might grow larger than what can fit in the original instruction. For example, x86-64 has two formats of relative jumps – short jumps with a one-byte displacement and long jumps with a four-byte displacement (x86-32 also supports short jumps with a two-byte displacement.) It is rather easy to overflow short jumps especially in large functions. One feasible solution is to restrict the moving distances of short jumps within the one-byte limit. However, this could quickly become over-complicated if several short jumps are close to each other. We might end up with several basic blocks unchanged or only moved by a short distance. Remix instead configures the compiler to always generate the equivalent long jumps with four-byte displacements. This is also the case for call instructions which have either a two-byte or a four-byte displacement.

Figure 1 gives an example of applying Remix to a short x86-64 function. After Remix, four basic blocks are moved to new positions. Branch and PC-relative instructions, including `jle`, `callq`, `jmpq` and `leaq`, are updated to maintain the control flow. Moreover, two `jmpq` instructions (`0x400d79` and `0x400d85`, after Remix) are inserted for the fall-through of basic blocks. Another `jmpq` instruction (`0x400d30`) is placed at the function entry point targeting the first basic block.

2.3 Basic Block Pointer Conversion

User-space programs built by compilers often do not need or have direct access to basic blocks. Accordingly, most programs have no explicit pointers to basic blocks. However, the compiler might spontaneously create such pointers when compiling the source code. For example, a return address on the stack points to the instruction following the corresponding `call` instruction. Besides, the compiler often uses jump tables to speed up the `switch`/`case` statements. After Remix reorders basic blocks, these pointers become invalid and thus have to be updated. In the rest of this section, we discuss these cases in detail.

2.3.1 Return Address Conversion

A call instruction automatically pushes its return address to the stack so that the callee can continue the execution from there upon return. The return address points to the instruction following the call instruction, i.e., the return site. When Remix performs live randomization of the process, the stack has already contained return addresses. If these addresses are not subsequently updated, the process will return to wrong locations, eventually causing exceptions such as illegal opcode or segmentation fault.

To convert return addresses, we traverse the whole stack (starting at the top of the stack in register `rsp`), and search for and update every address that points to a valid return site. With this condition, the chance of a stack variable being accidentally treated as a return address is very slim. In addition, return address conversion is straightforward and deterministic if the program maintains stack frame pointers. A stack frame is a continuous block of memory on the stack that keeps data for an active function. If frame pointers are maintained, each frame contains a pointer to the previous frame, and the return address is stored at a known location in the frame. Therefore, we can traverse stack frames and update all and only return addresses. By default, modern compilers like `gcc` do not generate code to maintain frame pointers in an optimized compilation.

2.3.2 Indirect Jump Related Conversion

Indirect jumps are used by the compiler and standard libraries for a number of purposes. They can target either functions or basic blocks. No change is needed for the former, but the latter requires us to update the associated basic block pointers.

Function Pointers: the compiler uses indirect jumps (to functions) mostly to support shared libraries, C++ vtable, and tail/sibling calls. For example, the compiler generates the PLT and GOT tables for calls to external functions in a shared library [35]. The library is loaded at a random address unknown until the program runs. At the run-time, the linker resolves the address of each called external function and saves it in a GOT entry. A PLT entry is an executable trampoline that represents the actual function. It essentially is an indirect jump to the function address saved in its associated GOT entry. The PLT table is placed in a special

```
(A) jmpq    *0x480000(,%rax,8)

(B) jmpq    *0x8(%rax,%rcx,8)

(C) movslq  (%r9,%rbp,4),%rcx
    add     %r9,%rcx
    jmpq    *%rcx
```

Figure 2: Jump Table Examples

section. Remix leaves this section unchanged. Tail/sibling call optimization is also interesting. The compiler normally allocates a new stack frame for each function call. However, there are cases where the callee can safely share the caller's stack frame. Such a call is dubbed the tail call or the sibling call, depending on the location of the call instruction. A typical example of the tail call is tail-recursive functions [52], but compilers like gcc support the broader definition of tail/sibling call. They can identify these cases and reuse the callers' stack frames. If the callee is a function pointer, the compiler generates an indirect jump (instead of an indirect call) in order to reuse the stack frame. Remix does not need to change indirect jumps introduced by tail/sibling call optimization.

Saved Context: indirect jump is also used by the standard C library to restore saved context. For example, the `setjmp` and `sigsetjmp` functions save their calling context to a jump buffer, while the `longjmp` and `siglongjmp` functions restore the context saved by `setjmp` and `sigsetjmp`, respectively. Both functions use an indirect jump to continue the execution at the saved instruction pointer. After reordering basic blocks, Remix needs to update all the jump buffers. The most efficient solution is hooking the functions that save the context and record the locations of the jump buffers. Note that the saved registers in the jump buffer are encoded by glibc in a special format (PTR_MANGLE). The alternative approach to search the whole address space for jump buffers incurs unnecessary performance overhead as these functions are seldom used.

Jump Tables: jump tables are often generated by the compiler to speed up `switch`/`case` statements. If some cases are continuous, the compiler stores their handlers in a table, and uses the switch variable as an index to quickly locate the corresponding handler. On x86-64, various patterns of jump tables can be used [17] as shown in Figure 2. They all have a base address, an index register, and a scale. An entry in the jump table can be addressed by $(base + index * scale)$. For example, the bases of case A, B, and C are constant `0x480000`, register `rax`, and register `r9`, respectively, and the indexes are in the `rax`, `rcx`, and `rbp` respectively (in case C, `rbp` is used as a general-purpose register, not the stack frame base pointer.) Interestingly, while case A and B store the actual handler addresses in the table since they directly jump to the selected entry, case C stores the offsets between the table base and the handlers. Each offset is only four bytes (a pointer is 8 bytes on the x86-64 architecture.) To calculate the handler address, the code reads the offset into register `rcx` and adds it to the table base in register `r9`.

Handlers for a `switch`/`case` statement are some basic blocks of the enclosing function. Remix thus has to update them after reordering basic blocks. The first two cases are rather straightforward to handle: jump tables are typically placed in the `.rodata` section. We search this section looking for at least 3 consecutive addresses pointing to the code section. If these addresses are close enough to each other (e.g., no more than 1MB apart) and all point to a valid instruction, Remix updates them accordingly. Even though false positives are possible, we did not find it to be a problem during our experiments. This approach does not work on the third case whose jump table consists of offsets, not instruction addresses. A simple solution is to export some meta data (e.g., the table base and length) from the compiler for Remix to patch the table at the run-time. Remix then can locate each handler and adjust its offset by the displacement between the old handler address and the new one. Our prototype uses this approach. Another viable solution is to use pattern matching to locate the code similar to case C (registers might be different) and use an intra-procedural, backward program slicing [2, 56] to locate the table base and length. For example, the index (register `rbp` in case C) is often compared to the table's upper and lower limits to make sure that it is within the table's boundary. This gives us the valid range of the index and hence the table length. As for the table base, the compiler generates case C mostly for position-independent code (e.g, shared libraries). The table base is calculated at the run-time using the PC-relative addressing mode, which has its own patterns (Section 2.2). As such, the table base can be calculated using the program counter and an offset. This approach is more complicated but it is the only choice if the source code is not available.

Exception tables can be similarly patched. Each exception table entry consists of a code range and a handler. If an exception happens in that range, it should be handled by the associated handler. However, Remix might move a faulting instruction out of the range and cause no handler or a wrong handler to be called. To address that, we can either revert the faulting instruction to its original location or avoid moving basic blocks into and out of the range. Our prototype has yet to implement this feature. Nevertheless, we can complete our experiments (including the Apache server and a kernel file system) without any problem. Even though exception handling is exploited by malware or DRM software to *obfuscate* control flows, regular applications do not use it that way (i.e., they use it for exceptions, not regular control flows.) since exception handling is relatively slow.

2.4 Live Randomization of Kernel Modules

Live randomization of the kernel code faces many of the same challenges as that of user applications. For example, the kernel can be compiled to use jump tables for tight `switch`/`case` statements. The kernel may also use exception tables – the kernel often needs to access the user memory. To protect itself from untrusted applications, the kernel must verify every user address it accesses, an expensive operation that requires traversing the process' paging structures. Moreover, the vast majority of user addresses are benign and safe to access. To avoid unnecessary verification, the kernel accesses the user memory without a prior verification. Instead, it registers a page fault handler that will be called by the kernel if the memory access fails. These cases can be similarly handled as in the user-space.

Nevertheless, there are some differences between the kernel and user applications. For example, the kernel often embeds manual assembly code, which may not follow the paradigms of the compiled code. That code has to be handled case-by-case (for once). The return address conversion

is more complicated than the user space because the changed return addresses could exist in any of the active kernel stacks (if a process is running in the user-space, its kernel stack is empty.) All these stacks need to be updated at once. In addition, a hardware interrupt can interrupt any instruction in the kernel or the user space. The interrupted address is saved on the kernel stack. If the interrupted instruction is in the kernel and has been moved, we can directly update the saved interrupt context. However, if the interrupted instruction is in the user space, Remix cannot update the kernel interrupt context, which is protected from the user space. Consequently, in the user space, Remix should not move an instruction that may be interrupted, i.e., the instruction that is currently executing. In our prototype, we stop the whole process (to guarantee consistency) and use a small agent to reorder basic blocks. The agent does not randomize itself. Even though it is possible to randomize the whole kernel, our prototype currently supports live randomization of kernel modules (e.g., the ReiserFS file system).

2.5 Performance Optimization

In this section, we present our strategies to improve the run-time performance of protected processes and to reduce the randomization latency.

2.5.1 Probabilistic Loop Bundling

Compilers make an effort to optimize the layout of the generated code for better performance. For example, gcc has an option to align loops to a power-of-two boundary (`-falign-loops`). If enabled, gcc inserts a number of NOPs before the loops to properly align them in the cache. If the loops are executed many times, the performance gain from the alignment outweighs the time wasted in executing NOPs. Remix, as well as other basic block randomization systems, disrupts this careful layout of the code. Because Remix randomly rearranges basic blocks, its final performance impact is somewhat unpredictable due to the complex interactions between the program and the cache hierarchy. For example, our early experiments find that Remix incurs low overhead for most SPEC CPU2006 benchmarks, but there are a couple of outliers with more than 15% overhead. To address that, we propose probabilistic loop bundling.

Loops are critical to the overall performance. A process often spends most of its execution time in loops. Changing the layout of loops might incur the largest impact to the performance. Accordingly, Remix focuses its optimization on the loops. It can probabilistically bundle the basic blocks of loops. A bundled loop has the same internal layout of basic blocks as the original, non-randomized loop. Within the boundary of a function, we consider the destination of a backward jump as the beginning of a loop and the jump as its end (even though this loop detection is quite rough, it is sufficient for our purpose.) We also control the size of a bundled loop by limiting the number of the jump and return instructions it contains. This avoids bundling large loops – for some functions, their bodies consist of a single large loop. Before the first randomization, Remix detects loops in the original code and records the layout of their basic blocks. During the live randomization, Remix flips a coin with certain probability to decide whether or not to bundle a loop. If a loop is bundled, its basic blocks are restored to the original, compiler-generated layout. The whole bundle is then treated as a single basic block and takes part in the randomization. In other words, a bundled loop is still moved around but its internal basic blocks remain relatively static. If possible, we make bundled loops to be 16-byte aligned. Our prototype bundles loops with a probability of $\frac{1}{3}$. i.e., about $\frac{2}{3}$ of the loops are randomized. Finally, we want to emphasize that each live randomization individually selects which loops to bundle. No loops will always be bundled.

Software	glibc	httpd	nginx	lighttpd	OpenSSL
NOP Space	42.9	19.3	26.2	22.1	19.9

Table 1: Average NOP Space per Function

2.5.2 Meta-data Maintenance

Remix reorders basic blocks from time to time to make the code layout unpredictable. This is a time consuming process especially for large programs. In addition, Remix has to stop the whole process during randomization to ensure consistency. Otherwise, a multi-threaded process might have unsuspecting threads executing partially randomized functions. To this end, Remix maintains some meta-data to facilitate live randomization. For example, it builds an index for basic blocks and some important instructions (e.g., `call` instructions and jump tables). The meta-data is built from the ground up in the first run and kept updated afterwards. With the meta-data, Remix can significantly reduce the randomization latency. To protect the meta-data from being leaked, we allocate its memory at a random location. Even though the meta-data is stored in the process' address space, it is isolated from the process itself because no pointers to the meta-data exist in the process (our prototype stores the base address for the meta-data out of the process. See Section 3.) Information leak vulnerabilities in the process cannot disclose the meta-data location or its content. To be more cautious, we could move the meta-data to random locations at undetermined intervals.

2.6 Binary-only Program Support

If the source code is available, Remix uses a (slightly) customized compiler to reserve enough space for extra jumps necessary to connect reordered basic blocks (Section 2.1). However, the source code is not always available, especially for commercial or legacy programs. Remix has a compatibility mode to support binary-only programs by leveraging the existing NOP padding in the code. As previously mentioned, compilers often insert NOP instructions to align functions and loops to a power-of-two boundary. As such, there are NOPs between and inside functions. Table 1 shows the average NOP space per function (in bytes) for several popular software packages. Remix can use the NOP space for its purpose. We treat small and large functions differently: small functions naturally contain less NOP instructions, but short jumps (2 bytes each, one byte for the opcode and the other byte for the displacement) are often enough to chain basic blocks; Large functions have more NOP space available, but basic blocks might be moved far apart from each other. To chain two basic blocks, we use short jumps whenever possible and long jumps otherwise. If the space runs short, we bundle some basic blocks together to reduce the extra jumps needed (similar to the loop bundling). During each live randomization, Remix picks different sets of basic blocks to bundle together. This ensures that a different code layout is generated each time.

3. IMPLEMENTATION

We have implemented a prototype of Remix for the Linux applications and the FreeBSD kernel modules on the x86-64 architecture. The FreeBSD kernel is chosen because it has better support for the LLVM/Clang compiler. In this section, we describe our prototype in detail.

We slightly modify the LLVM/Clang compiler to insert a 5-byte NOP instruction (`nopl 8(%rax, %rax)`) after each basic block. To achieve that, we add one line to the `EmitBasicBlockEnd` function in LLVM. These 5-byte NOPs also serve as delimiters for basic blocks because LLVM itself does not use this type of NOP (it does use other formats of NOPs, such as `xchg %ax,%ax`.) This makes basic block identification straightforward for Remix. To ensure that LLVM only generates long jumps (Section 2.2), we pass `-mc-relax-all` to the LLVM backend. However, it unnecessarily relaxes other instructions, such as `add` and `sub`, to full displacements as well. With more invasive changes to LLVM (likely in the `fixupNeedsRelaxation` function), we could make LLVM relax only branch instructions. We use Capstone, a cross-platform multi-architecture disassembly framework, to disassemble instructions in the memory. Linux enforces $w \oplus x$ for user applications, in which a block of memory is either writable or executable, but not both simultaneously. As such, we use the `mprotect()` system call to temporarily make the `.text` and `.rodata` sections writable. After live randomization, we set their permissions back.

A major implementation challenge is to guarantee the consistency of the process, especially for a multi-threaded process. All the threads should enter a consistent state before live randomization, and have their data updated before the execution is resumed. A viable solution is to use a kernel module and pause all the threads at the system call boundary. In our prototype, we use the `ptrace` interface to stop the whole process (for single-threaded processes, a timer signal can also serve this purpose.) Similarly, we need to put the kernel in a quiescent state and update all the affected kernel stacks consistently.

Ptrace is an interface for process tracing and debugging. It allows one process to inspect and control the execution of another process. We start the target program under the control of a small utility program, which is responsible for initiating live randomization at random intervals (for brevity, we call it the initiator.) When it is time for live randomization, the initiator sends a `SIGSTOP` signal to the target process and waits for it to stop. For each stopped thread, the initiator has full access to its execution context, including the registers and the program counter. Even though we could randomize the code with ptrace, the ptrace interface is too slow for this task – each access to the target process' memory must be conducted through an expensive system call. Instead, we pre-load a small agent in the target process and use ptrace to activate the agent. The agent performs the live randomization and returns the control back to the initiator when it finishes. The initiator can subsequently restores the process' state and resumes its execution at the interrupted instructions. However, these instructions might have been moved to different positions. To fix that, the initiator requests the agent to translate the interrupted program counters to their new values. To avoid interfering with the target process' heap and stacks, the agent uses the `mmap` system call to allocate new memory for its own heap and stack. The agent makes system calls directly instead of using the equivalent libc functions because it might be libc that Remix is currently randomizing (if so, libc is in an inconsistent state.) To prevent the agent from being exploited by code reuse attacks, the initiator relocates the agent from time to time. Moving the agent is much simpler than the live randomization of regular processes because the agent is small, position-independent, and self-contained (i.e., it does not rely on other libraries.)

Software	Apache	nginx	lighttpd
Average Basic Block #	15.3	18.8	14.4
Average NOP Space	19.3	26.2	22.1

Table 2: Statistics of Three Web Servers

In the FreeBSD kernel, live randomization is triggered by a timer. When the timer expires, we call the `smp_rendezvous` function to put all the CPUs in a consistent, quiescent state. `Smp_rendezvous` sends inter-processor interrupts to signal all the CPU cores. They rendezvous and execute the same set of functions. In our prototype, one core performs live randomization while others wait for it to finish. That core reorders the basic blocks of the target kernel module and searches the kernel stacks and other data structures for the affected basic block pointers. After randomization, all the cores are resumed and continue the interrupted execution.

4. EVALUATION

In this section, we first analyze the security guarantee of Remix and then measure its performance overhead with standard benchmarks.

4.1 Security

Remix randomly reorders basic blocks within their respective functions to increase entropy. It complements the existing ASLR support in commodity operating systems. ASLR randomly places the executable in the address space. It only provides a coarse-grained protection against code reuse attacks. The leak of a single code pointer, such as a function pointer or a return address, is often sufficient to de-randomize the whole executable. The attacker often leverages an information leak vulnerability to de-randomize the victim process before full-on attacks [51]. Remix can significantly improve ASLR's resilience to this type of information leak. It reorders the basic blocks of each function at random intervals. The actual code layout is unpredictable and keeps changing from time to time. Even if two systems run the exactly same programs, their run-time code layouts are different. Table 2 shows the average number of basic blocks per function for three popular web servers, Apache, nginx, and lighttpd. They all have about 16 basic blocks per function on average. Therefore, Remix adds about four bits of entropy to each instruction of these programs. This leads to about 20% to 25% boost in the entropy for 32-bit systems [46]. More importantly, Remix introduces the time as a variable to address space layout, making it a moving target. The compiler often spontaneously inserts NOP instructions to the generated programs to align functions or loops. Table 2 also shows the average NOP space per function (in bytes) for those programs. The NOP space can be leveraged to further increase the entropy by randomly placing NOPs between basic blocks. For short functions with less than 4 basic blocks, we also insert some additional NOP space to improve the entropy.

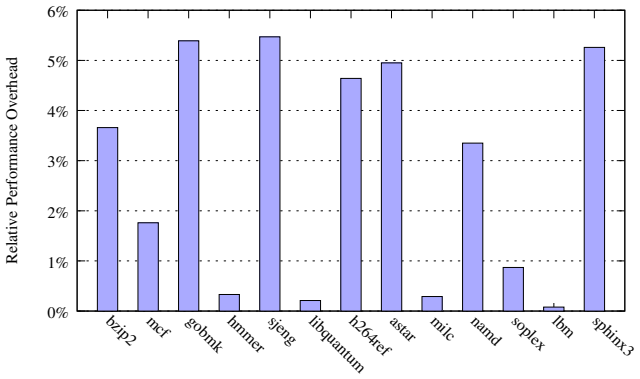

Figure 3: SPEC CPU2006 Performance Overhead

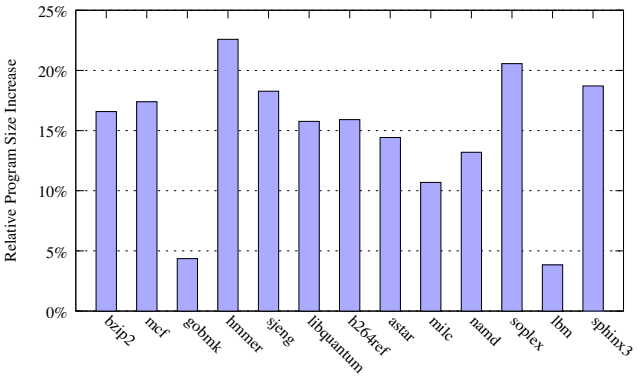

Figure 4: SPEC CPU2006 Size Increase

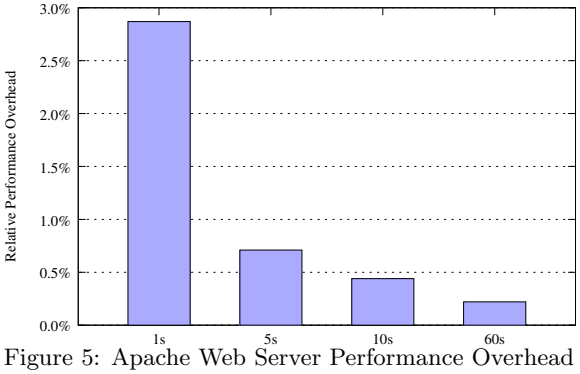

Figure 5: Apache Web Server Performance Overhead

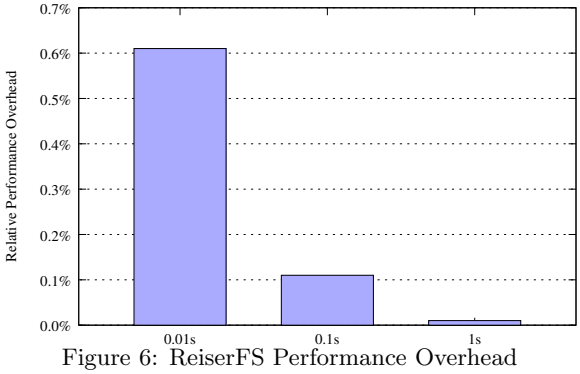

Figure 6: ReiserFS Performance Overhead

Recently, researchers have proposed a few novel attacks against fine-grained code randomization. For example, JIT-ROP (Just-in-time ROP [47]) repeatedly exploits a memory leak vulnerability to recursively map out the victim process' address space, and synthesizes code reuse attacks on demand. JIT-ROP is particularly detrimental to code randomization techniques that randomize the process only once at the compile or load time [49]. Remix's live randomization could potentially disrupt the JIT-ROP attack (if the code happens to be randomized by Remix during the attack), but it is not always effective. A sure defense against JIT-ROP is execute-only memory, in which the code can only be executed but not read. Fortunately, execute-only memory is being adopted by major CPU architectures [4, 19, 31] and can be emulated in the software [6, 26]. Remix should incorporate the execute-only memory as a defense-in-depth solution. For performance reasons, our prototype implants an agent and its meta-data into the target process. However, it is unlikely that JIT-ROP could find these artifacts. Even though they exist in the process' address space, they are isolated from the process itself because the process has no pointers to them. In addition, we could move them to random locations from time to time. JIT-ROP carefully maps the victim process' address space to avoid accessing invalid memory. Blindly probing the Remix memory most certainly will trigger a general protection exception and be foiled. BROP [10] is another attack against fine-grained code randomization, which exploits a victim process many times to essentially brute-force it in order to locate useful gadgets (it assumes the process would be restarted upon crash.) During this long process, Remix likely has randomized the process a few times, making the probed gadgets useless.

4.2 Performance

The performance impact of Remix mostly comes from the following two aspects: first, live randomization has to stop the whole process or the kernel to ensure consistency. This introduces some latency to the whole process. Second, Remix rearranges the code layout. Modern computer architectures rely heavily on the cache for performance. Changing the process' code layout can affect its cache profile and by extension the performance. We measure both aspects with standard benchmarks (SPEC CPU2006) and a number of popular applications. All the experiments are performed on a third-generation Intel Core i7 machine with 16 GB of memory. The operating system is the 64-bit Ubuntu 14.04.2 LTS. LLVM version 3.6 is used as the base compiler.

To measure the execution overhead, we randomize the SPEC CPU2006 benchmarks once (with a probability of $\frac{1}{3}$ for loop bundling) and compare their performance to the baseline built with the unmodified LLVM compiler. All the experiments are repeated 20 times. The standard deviation of the experiments is negligible. Figure 3 shows the performance overhead caused by Remix (C++ benchmarks with exceptions currently are not supported yet.) The overhead for most benchmarks are less than 5% with an average of 2.8%. To reserve space for reordering basic blocks, Remix inserts a 5-byte NOP instruction after every basic block. It also relaxes various instructions to use larger constants (e.g., `jmp` and `call`). This could substantially increase the program binary size. Figure 4 shows the size increase of SPEC CPU2006. The average increase in size is 14.8%. We use ApacheBench to measure the performance impact of live randomization intervals. We run the Apache server and ApacheBench on two directly connected machines. We

use ApacheBench to send $5 * 10^6$ requests to the server with a concurrency of 10. We set Remix to periodically randomize the server with an interval of 1, 5, 10, and 60 seconds, respectively. As expected, one-second interval incurs the highest overhead (2.9%). The overhead gradually decreases as the time interval increases. At the ten-second interval, the overhead is only 0.44%.

Remix not only supports user-space applications but also kernel modules. Our experiments are based on the FreeBSD kernel as it has better support for the LLVM/Clang compiler. We use Remix to live randomize the ReiserFS kernel driver [44]. IOZone, a user-space file system benchmark [40], is used to measure the performance of ReiserFS under different randomization intervals. We test the stride read of a large file in the automatic mode with a record size from 4KB to 512MB. The performance overhead of Remix is negligible even with a randomization interval of 0.01 seconds (Figure 6). This is expected as the performance bottleneck is in the disk I/O. We also test the read/re-read operations and get very similar results.

5. DISCUSSION

In this section, we discuss some possible improvements to Remix. *First*, Remix reorders basic blocks within their respective functions. The entropy increase by Remix is thus limited by the number of basic blocks in the function. Smaller functions have fewer basic blocks and thus benefit less from Remix. In our prototype, we insert extra NOP space in small functions to increase the entropy. Furthermore, we could incorporate fine-grained code randomization [30] specifically for these small functions. One of the key benefit of reordering basic blocks within functions is that function entry points remain at their intended location. Consequently, there is no need to migrate stale function pointers, which in general is an unsolvable problem. However, this does not necessarily require that basic blocks remain within function boundaries. We could randomly place basic blocks in the whole address space, and use a long jump at each function entry point to jump to its first basic block. This system has the benefits of binary stirring's higher entropy gain and Remix's simpler live randomization. However, its spatial locality of the randomized code is even more fragmented than our current design. It is necessary to carefully study the optimal basic block layout for better performance and security. Meanwhile, Remix does not *lively* move functions to avoid the complex runtime fixing of stale function pointers. Functions are nevertheless randomized at least once during the program startup by ASLR. Remix is a highly composable technique. It can be naturally integrated with systems that lively randomize functions [27] or with other techniques.

Second, some programs contain code that cannot be automatically randomized by Remix, such as inline assembly code, which sometimes does not follow the (relatively) clean paradigm as the compiled code. For example, kernels often use inline assembly in trampolines for interrupt handlers. A trampoline prepares the kernel stack to handle pending interrupts. The addresses of these trampolines are stored in the interrupt vector table. When an interrupt is triggered, the CPU indexes into this table and dispatches the corresponding handler. If these trampolines are reordered, we need to update the interrupt vector table. In addition, some programs have code that cannot be cleanly disassembled (e.g., the obfuscated code), and programs like just-in-time compiler can dynamically generate binary code. There does not seem to have a universal solution to these diverse problems. We instead have to handle them case-by-case. For example, we could incorporate the design of Remix into the JIT compiler so that dynamically generated code can also be randomized.

Third, Remix performs live randomization of the target process at an undetermined interval. The choice of this interval is a trade-off between randomization latency and security. As mentioned earlier, Remix provides probabilistic protection against information-leak based attacks such as JIT-ROP [47] (Section 4.1). That is, the protection is in effect if Remix happens to randomize the target process during the attack. As such, an interesting criterion to decide the live randomization interval is how likely Remix can disrupt these attacks. Our prototype uses an interval of ten seconds as a trade-off between randomization latency and security. Like other code randomization systems, Remix is, after all, a probabilistic defense. A more complete, defense-in-depth system should combine Remix with specific defenses against those attacks (e.g., execute-only memory to prevent JIT-ROP).

Lastly, Remix inserts an extra NOP instruction after each basic block to reserve space for reordering basic blocks. A program built by Remix is still a valid one that can be executed standalone. It is just larger (14.8% average size increase for SPEC CPU2006) and probably runs slower. Our tests show that Remix-built programs run mostly as fast as or only slightly slower than the original programs. This result is expected as modern processors have an efficient and intelligent instruction prefetching system. However, there are a few outliers that execute even faster than the baseline. This is probably caused by the complex interaction between the instruction alignment and the cache hierarchy. Native client (NaCl) shows similar results [53]. NaCl is a software fault isolation system that can safely execute native code in the web browser. In NaCl, the untrusted code is divided into equal-sized fragments, and no instructions can cross the fragment boundary. NOP instructions are used to pad the fragments if necessary.

6. RELATED WORK

In this section, we present the state-of-the-art in the research of code reuse attacks and defenses.

ROP Defenses: the first category of related work is various defenses against return-oriented programming (ROP). ROP exploits short snippets of the existing code, called gadgets, for malicious purposes [12, 45]. Gadgets of ROP end with a return instruction. This allows the attacker to chain a number of gadgets together using a crafted stack. ROP has been demonstrated to be Turing-complete when given a reasonably sized code base. Variations of ROP that do not rely on return instructions have also been proposed [11, 15]. Code randomization and control-flow integrity are two systematic defenses against ROP (we will discuss them in this section later.) Besides, there are a wide variety of diverse ROP defenses. For example, G-free eliminates usable gadgets at the compile time by removing unaligned free-branch instructions [41]. Return-less also leverages a customized compiler to remove intended and unintended return instructions [36]. KBouncer [43] and ROPecker [16] detects ROP attacks by checking whether the path to a sensitive system call contains too many indirect branches to "short" gadgets.

Recent work shows that this approach might not be effective [13] and the detection threshold is difficult to determine [29].

Software Diversity: the second category of closely related work is software diversity, in which the program code or data are diversified to foil attacks that depend on knowing particular program attributes [34]. For example, ROP chains the gadgets together by arranging their addresses on the stack. Each gadget ends with a return instruction, which pops the next gadget address from the stack and executes it. As such, ROP needs to know the gadget addresses. To defeat ROP, code randomization aims at making gadget addresses unpredictable. Code randomization systems differ in the randomization granularity. Address space layout randomization (ASLR) is a popular coarse-grained code randomization system. It places the program binary as a whole at a random base address [48]. Consequently, ASLR has limited entropy on the 32-bit architectures [46]. Because the program internal layout is not changed, ASLR is especially vulnerable to information leak attacks. A single leaked code pointer can de-randomize the whole process. ASLP works at a finer granularity than ASLR [33]. It permutes functions and static data objects in addition to randomizing the section bases. In comparison, Remix works on the basic blocks and also supports live randomization of running processes. Binary stirring is one of the fine-grained code randomization systems. It also works at the basic block level. However, it stirs basic block globally for once at the load time. Remix instead reorders basic blocks within their respective functions. This localizes the changes required to compensate the moved basic blocks. It allows a relatively simple implementation of live randomization. Giuffrida et al. proposes a live randomization system that relies on heavy compiler customization to output meta-data for the pointer conversion [27]. In particular, it needs to migrate function pointers which may involve unsolvable ambiguity and require manual efforts. Remix confines the changes (mostly) to functions, and thus are easier to implement. At an even finer granularity, some systems randomize the instruction set through encoding or encryption to defeat code injection and code reuse attacks [8, 32]. ILR randomizes the location of every instruction [30]. It uses a process virtual machine (Strata) to execute the scattered code. IPR rewrites instruction sequences with equivalent same-length instructions [42]. It can eliminate about 10% useful gadgets and probabilistically break 80% of them. It supports various concrete transformations, such as atomic instruction substitution, instruction reordering, and register reassignment. Data randomization has also been proposed to prevent data-based attacks [9, 18].

Code randomization systems are often vulnerable to information leak attacks. For example, the coarse-grained ASLR could be de-randomized by even one leaked code pointer. Fine-grained code randomization systems like binary stirring are more resilient to leaked code pointers, but are still vulnerable to the leak of memory contents. For example, JIT-ROP repeatedly exploits a memory leak vulnerability to map the victim process' code in order to launch an on-demand ROP attack [47]. A few systems have been proposed to enhance fine-grained code randomization to withstand JIT-ROP attacks [6, 19, 26]. They all utilize the execute-only memory in which the code can only be executed but not read. Remix only provides probabilistic defense against JIT-ROP attacks (Section 4.1). Remix should integrate execute-only memory when it is available in the commodity hardware. Such a combination would significantly raise the bar of code reuse attacks.

Control-Flow Integrity: Control-flow integrity is another effective defense against code reuse attacks [1]. It inserts in-line monitors to confine the run-time control flow to the program's (static) control-flow graph. CFI systems vary in the protection granularity. Fine-grained CFI provides a strong protection against most control-flow hijacking attacks, but often has high performance overhead. It also requires a precise control-flow graph that still is not readily available in the commodity compilers. Recent research effort focuses on reducing CFI performance overhead for commodity programs [54, 55]. They trade the protection granularity for performance, leading to potential vulnerabilities [23, 28]. Opaque CFI uses coarse-grained control-flow integrity to strengthen fine-grained code randomization against certain types of information leak attacks [38]. Instead of validating the exact target address, OCFI ensures that the target is within a certain randomized bound. RockJIT leverages modular CFI to protect the JIT compiler and the dynamically generated code [39]. It builds a fine-grained CFG from the source code of the JIT compiler, and keeps the control-flow policy updated with the new generated code. Even though most CFI systems are implemented in the software, hardware architectural support for CFI has been proposed that can substantially simplify and speed up CFI systems [25].

In addition to control-flow integrity, researchers have proposed other security properties that can prevent code reuse attacks. For example, data-flow integrity (DFI) enforces that run-time data flow must follow the data-flow graph [14]. DFI can prevent many memory vulnerabilities from being exploited. Code-pointer integrity (CPI) separates sensitive data, such as code pointers and pointers leading to code pointers, from regular data to protect them from unauthorized modification.

7. SUMMARY

We have presented the design and implementation of Remix, a live randomization system for user-space applications and kernel modules. Remix randomly reorders basic blocks within their respective functions at undetermined time intervals. it can substantially increase the entropy of ASLR, one of our most important defenses against code reuse attacks. By randomizing the code layout, Remix can significantly enhance ASLR's defense against certain types of information leak vulnerabilities. Remix is a flexible and composable defense technique due to its unique design and efficiency. It brings to the composed systems extra entropy that changes with the time. Our experiments with both standard and application benchmarks show that Remix only incurs a small performance overhead.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments that greatly helped improve the presentation of this paper. This work was supported in part by the US National Science Foundation (NSF) under Grant 1453020. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

9. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[2] H. Agrawal and J. R. Horgan. Dynamic Program Slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.

[3] Apple. OS X MountainLion Core Technologies Overview. http://movies.apple.com/media/us/osx/2012/docs/OSX_MountainLion_Core_Technologies_Overview.pdf.

[4] ARM: the Architecture for the Digital World. http://www.arm.com/.

[5] Linux Kernel Address Space Layout Randomization. http://lwn.net/Articles/569635/.

[6] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[7] M. Backes and S. Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[8] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized Instruction Set Emulation. *ACM Transactions on Information and System Security*, 8(1):3–40, 2005.

[9] S. Bhatkar and R. Sekar. Data Space Randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[10] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking Blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[12] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[13] N. Carlini and D. Wagner. Rop is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[14] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[16] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A Generic and Practical Approach for Defending against ROP Attacks. In *Proceedings of the 21st Network and Distributed Systems Security Symposium*, 2014.

[17] C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Proceedings of 7th International Workshop on Program Comprehension*, 1999.

[18] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*, 2003.

[19] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.

[20] Memory Protection Technologies. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[21] x86 NX support. http://lwn.net/Articles/87814/.

[22] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proceedings of the 22nd Network and Distributed Systems Security Symposium*, 2015.

[23] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[24] Data Execution Prevention. http://en.wikipedia.org/wiki/Data_Execution_Prevention.

[25] U. Erlingsson, M. Abadi, and M.-D. Budiu. Architectural Support for Software-based Protection, Mar. 13 2012. US Patent 8,136,091.

[26] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM conference on Data and application security and privacy*, 2015.

[27] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[28] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[29] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size Does Matter: Why Using Gadget-chain Length to Prevent Code-reuse Attacks is Hard. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[30] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[31] Intel. *Intel 64 and IA-32 Architectures Software Developeŕs Manual*, 2014.

[32] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-injection Attacks with

Instruction-set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[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 *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006.

[34] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated Software Diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy*, 2014.

[35] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 1999.

[36] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits with "Return-less" Kernels. In *Proceedings of the 5th ACM SIGOPS EuroSys Conference*, 2010.

[37] A. I. Mark Russinovich, David Solomon. *Windows Internals, 6th Edition*. Microsoft Press, 2012.

[38] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque Control-Flow Integrity. In *Proceedings of the 22nd Network and Distributed Systems Security Symposium*, 2015.

[39] B. Niu and G. Tan. RockJIT: Securing Just-in-time Compilation Using Modular Control-flow Integrity. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, 2014.

[40] W. D. Norcott and D. Capps. Iozone Filesystem Benchmark. *URL: www.iozone.org*, 2003.

[41] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010.

[42] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[43] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Conference on Security*, 2013.

[44] H. Reiser. ReiserFS, 2004.

[45] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[46] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[47] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[48] P. Team. PaX Address Space Layout Randomization (ASLR), 2003.

[49] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, 2012.

[50] Wikipedia. Basic Block. http://en.wikipedia.org/wiki/Basic_block.

[51] Wikipedia. Pwn2Own. http://en.wikipedia.org/wiki/Pwn2Own.

[52] Wikipedia. Tail Call. http://en.wikipedia.org/wiki/Tail_call.

[53] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[54] 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 *Proceedings of the 34th IEEE Symposium on Security and Privacy*, 2013.

[55] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[56] X. Zhang, R. Gupta, and Y. Zhang. Precise Dynamic Slicing Algorithms. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.