Profile-guided Automated Software Diversity

Andrei Homescu
University of California, Irvine
ahomescu@uci.edu

Steven Neisius
University of California, Irvine
sneisius@uci.edu

Per Larsen
University of California, Irvine
perl@uci.edu

Stefan Brunthaler
University of California, Irvine
s.brunthaler@uci.edu

Michael Franz
University of California, Irvine
franz@uci.edu

Abstract

Code-reuse attacks are notoriously hard to defeat, and most current solutions to the problem focus on automated software diversity. This is a promising area of research, as diversity attacks the common denominator enabling code-reuse attacks—the software monoculture. Recent research in this area provides security, but at an unfortunate price: performance overhead.

Leveraging previously collected profiling information, compilers can substantially improve subsequent code generation. Traditionally, profile-guided optimization focuses on hot program code, where a program spends most of its execution time. Optimizing rarely executed code does not significantly impact performance, so few optimizations focus on this code.

We use profile-guided optimization to reduce the performance overhead of software diversity. The primary insight is that we are free to diversify cold code, but restrict our diversification efforts in hot code.

Our work investigates the impact of profiling on an expensive diversification technique: NOP insertion. By differentiating between hot cold and cold code, we optimize NOP insertion overheads from a maximum of 25% down to a negligible 1%, while preserving the security properties of the original defense. Consequently, using our profile-guided diversification technique, even randomization techniques having a high performance overhead become practical.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Code Generation, Optimization

General Terms Performance, Security, Measurement

Keywords Profiling, Automated Software Diversity, Compilers, Cold-Code, NOP insertion

CGO'13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

1. Motivation

Borrowing a term from biology, we refer to the prevalent practice of shipping identical binaries to all customers as the *software monoculture*. Dating back to 1993, Cohen [6] identified that this practice has detrimental effects for computer security—which Forrest reinforced in 1997 [9]. Due to a particularly malign class of attacks—known as *code reuse attacks*—community interest surged around protections using automated software diversity [12, 13, 17, 27].

While all of the proposed techniques successfully protect against code-reuse attacks, they suffer from performance impacts ranging from 1% to 13%. Interestingly, this spectrum blends well with the security guarantees of the approaches, i.e., the slower ones have the strongest security properties. Taking a closer look at the implementations of automated software diversity, we notice that all of them use a "one-off" design: they take an input program and diversify it in one pass. This resembles the state-of-the-art in compiler construction *before* the advent of profile guided optimizations triggered by Pettis and Hansen's profile guided code positioning of 1990 [30].

Feedback directed, or profile-guided, approaches have been a major line of research in compilation, in particular for generating and optimizing native machine code. A profiling run separates frequently executed—or *hot*—parts of code from infrequently executed—or *cold*—parts. Subsequently, a second compilation run uses these information to optimize the generated code, e.g., by co-locating frequently executed basic blocks.

Previous research focuses almost exclusively on optimizing the hot code parts. Our idea is to combine profiling information with automated software diversity. By doing so, we substantially reduce the costs of even expensive diversi-

fying transformations. One such transformation inserts NOP instructions in between intentionally emitted native machine instructions, a technique known as NOP insertion. To add software diversity to an input program, we have to insert NOPs probabilistically, i.e., depending on some random information, we decide whether to insert a NOP instruction or not. While blind insertion has a positive impact on security, it also affects performance. This is not surprising, because the random NOP insertion trial has no information about the whereabouts of frequently executed code. Profiling information gives us these clues: it tells us that we can diversify as much as we want to in cold code, and reduce diversification overhead in hot code. A similar insight led to successful research in the areas of code compression [8] and hardware error detection [18].

We present the design and implementation of profile-guided automated software diversity using the LLVM 3.1 compiler and its profiling infrastructure. Specifically, our contributions are:

- We introduce a technique that uses profiling information to optimize away overhead introduced by automated software diversity.
- We describe a heuristic formula controlling randomization based on basic block execution frequencies.
- We present the results of a thorough evaluation of our implementation. Our results indicate:

  Performance We are able to reduce overhead of probabilistic NOP insertion for SPEC CPU 2006 down to a negligible 1% performance overhead. This is an important result allowing us to use stronger diversifying transformations without sacrificing performance.

  Security We briefly describe a way to objectively measure the success of diversifying a binary. Our profiling-driven optimization preserves the security properties of code layout randomization. In addition, we show that our transformation thwarts concrete attacks.

2. Background

2.1 Code Reuse Attacks

For performance and practical reasons, a large part of the modern software stack is written in low-level systems programming languages. Since the programmer is assumed not to make any mistakes, little error checking happens at runtime. Consequently, even simple programming mistakes can lead to security vulnerabilities for attackers to exploit.

Originally, attackers would perform arbitrary code execution by injecting a binary payload (for example, x86 code) into a vulnerable application running on the target, then use some other vulnerability in the application to force it to execute the payload. This requires that the application be able to write data to a buffer, then execute code from that same buffer. In other words, the processor has to be able to (or be allowed to) execute instructions from an area of memory that the application can directly write to. However, modern processors allow for a page to be marked as non-writable or non-executable.

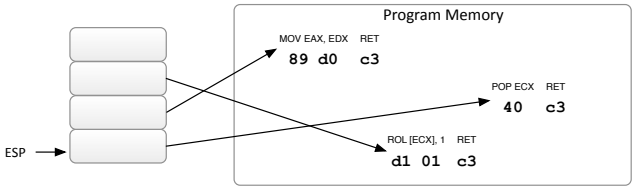

Figure 1. Return-oriented programming attack example.

Modern operating systems use this feature to prevent code injection attacks. A page cannot be both executable and writable at the same time, unless specifically requested by the application. This restriction is known as W$\oplus$X (implemented by the PaX [28] kernel patch on Linux). In practice, W$\oplus$X renders most code injection attacks ineffective. Since the payload is stored in a writable but non-executable buffer, the processor cannot execute its contents. In addition, some operating systems refuse to execute code that was not digitally signed by a central authority.

A recent class of attacks, known as *code reuse attacks*, circumvents these restrictions by using code from the application itself to perform the attack. Instead of injecting new code into the application, the attacker reuses fragments or entire functions from within the program. Return-into-lib(c) [24] attacks, for example, redirect the program's execution to a function (usually `system`) in libc or some other library, after manipulating the stack so the attacker controls the function parameters. Later work [36] proved that return-into-lib(c) is Turing-complete, and therefore able to perform arbitrary computations, by implementing a Turing machine using libc functions.

The later *borrowed code chunks* [20] and *return-oriented programming* (ROP) [33] techniques use code snippets (called *gadgets* in the ROP paper) from the executable section of the program as the attack payload itself. These code snippets are much shorter than functions, but still have one common trait: they all end in a `RET` [1] instruction. The attacker chains pointers to gadgets on the stack consecutively, then redirects execution to the first gadget. Each gadget, at the end of its execution, returns into the following gadget. Figure 1 shows a high-level example of this process. A later version of ROP, called *jump-oriented programming* [3], lifts the requirement that gadgets end in a return, using a jump-based dispatcher gadget instead.

ROP uses gadgets to implement the instructions of a virtual machine (VM). The attacker writes their payload as a program written in this virtual machine, then converts each VM operation into the address of the equivalent gadget. The ROP VM usually offers a significant number of useful op-

[1] The encoding for `RET` on x86 is the `C3` byte.

erations, like simple arithmetic, memory loads and stores, and conditional jumps. In the original ROP design, the operations formed a Turing-complete set; these provide an attacker with a simple and effective way to perform arbitrary computations inside the target. However, it is possible to successfully launch an attack with even a restricted subset of operations, without providing Turing-completeness. Often, the attacker only needs to call some system function (like `mmap`), store a payload into a memory area and then redirect control flow to that memory region.

2.2 Software Diversity

All code reuse attacks have one trait in common: the attacker must have knowledge of the program code itself. At present, there are no known "blind" code reuse attacks, where the attacker does not rely on any kind of assumptions about program code. One practical restriction is that an attacker can only use code from the executable sections of the program, due to W⊕X. Therefore, one way to defend against code reuse attacks is to prevent the attacker from gaining any useful information about the program itself, such as the addresses of known gadgets inside the program.

Cohen [6] proposes the following strategy to defend against attacks:

> *The ultimate defense is to drive the complexity of the ultimate attack up so high that the cost of attack is too high to be worth performing.*

Cohen proposes program evolution as a defense strategy, where programs "evolve" into different, but semantically equivalent versions, of the original program. He then demonstrates several different evolution techniques, such as equivalent instruction replacement, instruction reordering and garbage insertion. As these techniques all change the program code in different ways, by either adding new instruction or shifting the existing ones around, they have a potentially large impact on both the locations and order of instructions in the program. Since code reuse attacks require the attacker to have exact knowledge of the contents of the binary, performing such attacks becomes much more difficult.

Many arbitrary code execution attacks require the attacker to redirect the execution of the program from its regular path to some malicious behavior under the attacker's control. Often, the attacker does this by overwriting some function pointer or jump target in memory. In return-oriented programming, for example, the attacker overwrites the return address of the current function and all following stack locations. Often, attackers use a buffer overflow vulnerability to overwrite these locations, by writing past the end of a buffer stored in a function frame. Stack frame randomization [9] is an implementation of software diversity targeted at stack-based attacks, using variable reordering and stack frame padding to diversify the binaries. A performance and security analysis showed that this approach successfully prevents buffer overflow attacks at negligible performance cost.

With the growing popularity of code reuse attacks, the need arises to introduce diversity into the program's binary code. By preventing the attacker from having a-priori information about the code layout of a program, we significantly raise the cost of a code reuse attack. Recent work looks at implementations of code layout randomization. This can be done in several places: in the compiler (by randomizing the code inside the compiler [12, 17]), in the operating system loader (disassembling the program and applying diversifying transformations to it; also, it can be done by replacing the loader) [13] or in between, as a separate step [27]. These implementations use techniques similar to the ones described by Cohen and show that code layout randomization is very effective at preventing code reuse attacks, preventing many ROP attacks currently at large.

Modern operating systems also implement a form of code layout randomization, called Address Space Layout Randomization (ASLR). This works by randomizing the base addresses of objects in the program, like the stack, heap, program code and dynamic libraries. However, as this randomization changes the location of most program objects, the operating system loader must now perform a significant number of relocations. An alternative implementation, currently used in Linux, changes the program so that it does not require fixed locations for any of its objects; instead, the program determines its own randomized base address at run time, then accesses all its objects by offsets from the base address. This approach has a performance penalty on 32-bit x86 systems [29], so it is currently disabled for most applications. Consequently, the code section of a program is always loaded at the same address (`0x8048000` on Linux), and only libraries are loaded at random addresses.

Another weakness of ASLR is that it only diversifies the base addresses of large program objects, such as code and data sections. There is no diversity inside the sections. If an attacker were to gain this base address by some information leak, they would have all the code layout information for a code reuse attack (assuming the attacker also possesses their own copy of the binary). On 32-bit systems, ASLR also suffers from lack of entropy; recent work [31, 34] shows that ASLR can be defeated in a matter of minutes.

3. Our Approach

A NOP [2] is an instruction that the processor fetches and executes without any effect on the processor register or machine memory. Compilers insert NOPs in programs for various purposes: (i) to enforce aligment of basic blocks, functions or other code blocks (for performance and security); (ii) to add timing delays to code fragments (for contention mitigation) [35]; (iii) to compensate for microarchitectural limitations, such as those in the branch predictor [15].

[2] NOP is the x86 architecture mnemonic for a "no-operation" instruction.

Data: The list $IList$ of instructions. The probability of insertion p_{NOP}. $NOPTable$, the list of candidate NOPs.
Result: The list $IList$ with NOPs inserted.
begin
  $numNOPs \leftarrow |NOPTable|$
  for $i \in IList$ **do**
    $roll \leftarrow random\,(0.0, 1.0)$
    if $roll < p_{\text{NOP}}$ **then**
      $nopIndex \leftarrow random\,(0, numNOPs)$
      $insert(i, NOPTable\,[nopIndex])$
    end
  end
end

Algorithm 1: NOP insertion algorithm.

In this paper, we use randomized NOP insertion to randomize the code layout of a program. At each program instruction, we insert a randomly chosen number of NOPs (zero or one in our current implementation), so that all following instructions are displaced by a random number of bytes. As the algorithm inserts NOPs through the program, the displacements accumulate, so that later instructions are pushed forward by increasingly larger amounts (Figure 2).

For every assembly instruction in the program, we decide whether to prepend a NOP before the instruction, with probability p_{NOP} of success. In case a NOP is inserted, we then pick one of the NOP candidates at random. Consequently, there are two sources of randomness in this transformation: whether to insert and what to insert. Algorithm 1 shows the algorithm in pseudocode.

Table 1 shows a list of eligible NOP instructions. We picked only instruction candidates that preserve the processor state at all times (as opposed to a weaker version of NOPs which change some minor part of processor state, e.g., an operation that adds zero to a register, not affecting registers or memory but changing the CPU flags). Second, we selected candidates with return-oriented programming in mind, carefully picking those that minimize the likelihood of creating new gadgets. In the case of the two-byte instructions, the second byte decodes to an operand or opcode that the attacker cannot use for nefarious purposes. For example, the `IN` instruction causes the processor to read from an input/output port. However, `IN` requires the processor to be in privileged mode to work correctly, causing unprivileged software to fault.

Table 1 shows seven NOP instructions that can be inserted as padding, but our implementation only uses five of them. The two `XCHG`-based NOPs, while perfectly suited to our goals, have a larger performance impact than the others. This is because, on current implementations of the x86 architecture, the `XCHG` instruction locks the memory bus [16]. None of the other NOPs require this, so we use them in our insertion pass. The two extra NOPs provide some extra diversity in the generated code, so they can be enabled at compile-time.

Instruction	Encoding	Second Byte Decoding
`NOP`	`90`	–
`MOV ESP, ESP`	`89 E4`	`IN`
`MOV EBP, EBP`	`89 ED`	`IN`
`LEA ESI, [ESI]`	`8D 36`	`SS:`
`LEA EDI, [EDI]`	`8D 3F`	`AAS`
`XCHG ESP, ESP`	`87 E4`	`IN`
`XCHG EBP, EBP`	`87 ED`	`IN`

Table 1. NOP insertion candidate instructions.

While our goal is to displace the original program instructions by random amounts, these NOPs also have another useful side effect. The x86 architecture is highly irregular, with instructions as short as 1 byte and as long as 20 bytes. Therefore, inserting one or two extra bytes inside an x86 instruction can change its decoding significantly, in many cases even changing the instruction's length. This effect is even more pronounced on ROP gadgets, where changing the decoding of one instruction can cause the next one to start at a different location. This offset propagates to the return instruction (encoded as the `C3` byte) so that the gadget no longer ends in this instruction. This effectively removes that gadget from the binary. Figure 2 illustrates this. NOP insertion not only meets our original design goal of displacing instructions by a random offset, but also has the added benefit of removing some gadgets entirely.

3.1 Profile-Guided Diversification

The approach to NOP insertion described in Section 3 uses the same probability for all instructions. The technique inserts the same expected number of NOPs inside loops and other frequently executed parts of code as in the rest of the program. In practice, most of a program's execution time is spent in a very small part of the code, usually a loop. Therefore, it makes much sense to alter the NOP insertion strategy for these regions, to minimize the performance impact of the extra instructions. To change the probability of NOP insertion according to the execution frequency of the current code region, we need a source of information for that frequency.

Run-time profiling is one source of execution counts. When optimizing using this approach, the compiler generates a special, instrumented version of the input program. The developer then runs this instrumented version on a training set of inputs. The purpose of this run is to collect execution statistics from the training run. These statistics include values such as execution counts for control flow edges and histograms for variable values. The compiler later uses this information for optimizations during a second compilation.

Most modern compilers support some form of run-time profiling and profile-guided optimization. In every case, the profile contains per-basic-block execution counts, or similar

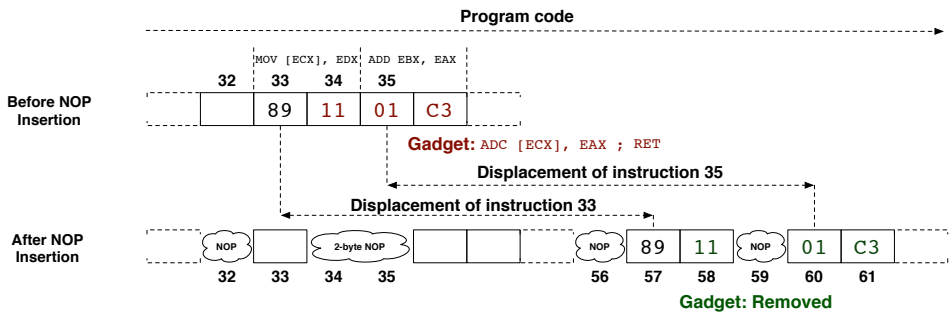

Figure 2. Effect of NOP insertion on program code.

information. LLVM, for example, only inserts counters for the minimal required subset of edges on the control flow graph [25]. The compiler derives all basic block execution counts from that minimal set of per-edge counters.

This approach provides accurate information on which parts of the code are executed most frequently, assuming that the training set is a proper sample of real-world usage of the input program. Note that per-basic-block execution counts are sufficient for our purpose; all instructions in a basic block are executed the same number of times. Therefore, we propagate basic-block execution counts to all instructions, and use the same probability of inserting a NOP for all instructions inside a basic block. More frequently executed blocks will correspond to lower NOP insertion probabilities.

To achieve this, we replace the singular NOP insertion probability with a range of probabilities. The hottest basic blocks get the lowest value in this range, while the coldest blocks get the highest probability. Blocks between these extremes need some probability linked to their execution frequency. The simplest way to model this is a linear function:

$$p_{NOP}(x) = p_{max} - (p_{max} - p_{min}) \frac{x}{x_{max}}$$

where x is the execution count of the current basic block, x_{max} is the maximum execution count in the program and $[p_{min}, p_{max}]$ is the probability range. The compiler uses this function to compute the NOP insertion percentage of each basic block.

This is a simple and effective heuristic that improves our NOP insertion strategy. However, in practice, execution counts of basic blocks are not distributed linearly. Not only do loops often have large number of iterations, but inner loops are occasionally contained in one or more outer loops. In the latter case, the total number of iterations is the product of the iterations of the inner and outer loops. This means that the counts grow exponentially in the number of loops, while the base itself can be a large number. A linear function would simply polarize the probabilities toward either the maximum or the minimum, since the maximum execution count has a very large value. For example, our measurements on a profiled run of SPEC CPU 2006 show that the maximum execution count ranges from 14 million (for 403.gcc) to 4 billion (for 456.hmmer). Also, for many applications, the execution counts are spread out between the minimum and maximum count. The 473.astar benchmark is one example of this: the median of all basic block execution counts is $117,635$, well under the maximum of 2 billion.

This means that a linear function is not an appropriate fit; a logarithm-based function would serve our purpose much better. The updated function for our profile-guided NOP insertion is:

$$p_{NOP}(x) = p_{max} - (p_{max} - p_{min}) \frac{\log(1+x)}{\log(1+x_{max})}$$

Using this function, the numerator and denominator of the fraction become much smaller. For an x_{max} of 4 billion and a logarithm base of 10, the denominator is approximately 10. Therefore, intermediate values get much smaller logarithms as well (in our case, logarithms range between 0 and 10). This also guarantees that counts that are smaller than the maximum by several orders of magnitude are still placed well inside the probability interval, not toward the minimum. The median from 473.astar, for a probability range of $[10, 50]$, now gets an approximate probability of $p_{NOP} \approx 50 - 40 * \frac{5}{10} = 30\%$ instead of $p_{NOP} \approx 50 - 40 * \frac{10^5}{10^{10}} \approx 50\%$. Therefore, using logarithms offers us a much better distribution of probabilities inside the interval, given the particular distribution of the execution counts gathered from profiling.

4. Implementation

One major design choice in the implementation of NOP insertion is the stage of the compilation process at which to insert the extra instructions. Figure 3 shows the stages a program goes through, from source code to final image in process memory. Most modern compilers take a program as source code, parse it into an abstract syntax tree, then flatten that tree into an intermediate representation (IR). One such IR is the LLVM IR used by the LLVM compilation framework [21]. The compiler performs a suite of optimizations on this IR, then lowers the IR into a lower-level representation (LR), such as the Register Transfer Language from GCC [10]. After performing even more optimizations (such

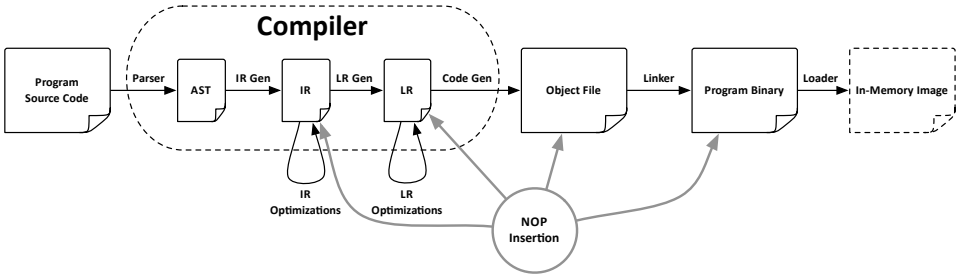

Figure 3. Life of a program, from source code to execution, along with possible stages where we can insert NOPs.

as register allocation), the compiler generates native code directly from the LR, into an object file. A linker later links one or more object files together into the final program binary. The operating system loader loads the program into memory and executes it.

It is theoretically possible to perform NOP insertion during any of these steps. However, each choice comes with its particular set of disadvantages. Inserting NOPs inside native code, either in object files or in the final program binary, requires complete information about the locations of all program instructions. As modern processors allow for read-only data to be mixed with code in the same stream, accurate disassembly has been proven impossible in the general case [6]. Another option is to use debug information, but that requires the program to be compiled with such information present. Inserting NOPs too early also presents a similar challenge: some IR operations might be lowered to more than one native instruction, while some other operations might disappear completely. Fortunately, most LR operations in a compiler have a one-to-one correspondence to the native code instructions in the object files. Therefore, our strategy is to insert NOPs into the lower-level representation, after the compiler performs all optimizations and just before it emits native code.

Our profiling-based randomization uses execution counts for basic blocks to tune the probability that a NOP is inserted. We derive the per-basic-block information from edge execution counts generated through instrumentation. Fortunately, LLVM currently implements a profiling framework which provides instrumentation for profiling [25]. At present, LLVM does not perform any optimizations that use the profiling information, but such optimizations can be easily added. The profiler adds a counter to each control-flow graph edge, and the counter is incremented at run-time each time that path is taken during program execution. Using this profiling framework, we implemented NOP insertion as a new backend pass in LLVM 3.1.

5. Evaluation

We performed all our tests on a 2.66 GHz Intel Xeon 5150 with 4GiB of memory. We used Ubuntu 12.04 as the host operating system, running Linux kernel version 3.2.0.

5.1 Performance Evaluation

To evaluate the performance impact of our technique, we built and ran the SPEC CPU 2006 benchmark suite with our diversifying compiler. We used the test system described above to run the benchmarks. We compiled all tests using the `-O2` optimization level. For the profile-guided diversification tests, we collected profiling information by running SPEC on the train input set.These inputs were designed specifically to provide an accurate profile for each program.

As the NOP insertion process uses a random number generator, there is some potential noise in the measurement due to the performance differences between different randomized versions. To account for this, we evaluated five different versions of each individual benchmark. We ran each version three times and averaged the results.

Figure 4 shows the performance impact of NOP insertion, with and without using profiling information and for a few combinations of probability parameters. First, we evaluated $p_{\mathrm{NOP}} = 50\%$, as this is the parameter that offers the maximum diversity; the number of possible versions of a program is maximized at this value. However, smaller values of p_{NOP} can also provide sufficient diversity, while also lessening the performance impact. For this reason, we also ran our tests with $p_{\mathrm{NOP}} = 30\%$. To measure the impact of profiling, we evaluated three sets of probability ranges for the profile-guided version of NOP-insertion: $p_{\mathrm{NOP}} = 25-50\%$, $p_{\mathrm{NOP}} = 10 - 50\%$ and $p_{\mathrm{NOP}} = 0 - 30\%$. In each range, the first parameter is the minimum probability that a NOP is inserted, while the second parameter is the maximum probability. We used the logarithm-based function to derive the per-basic block probability from the execution count.

Our results confirm that profiling has a significant impact on the performance overhead of NOP insertion. The performance overhead is around 8% for $p_{\mathrm{NOP}} = 50\%$ and a little less than 5% for $p_{\mathrm{NOP}} = 30\%$, but profile-guided randomization reduces that to 2.5% for $p_{\mathrm{NOP}} = 10 - 50\%$ and 1% for $p_{\mathrm{NOP}} = 0 - 30\%$ (a reduction factor of 5x compared to naive NOP insertion). For the benchmarks where the NOP insertion overhead is highest (400.perlbench and 482.sphinx3), profiling also has the highest impact. For the latter, the impact is reduced from 25% to 5% (for $p_{\mathrm{NOP}} = 10 - 50\%$) or as low as 1% (for $p_{\mathrm{NOP}} = 0 - 30\%$). The

470.lbm benchmark has the smallest overhead from NOP insertion; our measurements actually showed a very small performance gain from NOP insertion (under 0.5%), which we attribute to measurement noise. As LLVM does not currently perform any profile-guided optimizations, the performance gains come solely from inserting fewer NOPs in frequently executed code.

Our results also show that both ends of the probability range have an equally significant impact on the performance overhead of NOP insertion. A side-by-side comparison of $p_{\text{NOP}} = 10-50\%$ and $p_{\text{NOP}} = 25-50\%$ shows that reducing the minimum probability from 25% to 10% decreased the average overhead by half, as Figure 4 shows.

5.2 Security Impact

We evaluated the security impact of our diversifying compiler by examining how it affects gadgets in diversified executables. We measured this by counting how many functionally equivalent gadgets remain at the same location in the binary after diversification. Our comparison algorithm (called *Survivor*) extracts the `.text` sections from executables after diversification and compares them to the `.text` sections in unmodified original executables. *Survivor* scans through the sections, looking for common instruction sequences—*candidate* matches—ending in free branches such as returns, indirect calls, or jumps. A candidate match is a pair of instruction sequences with identical offsets; one from the original binary and one from the diversified one. For each candidate, we ensure that both sequences decompile to valid x86 code having no control-flow instructions except a free branch at the end. The algorithm then compensates for the effects of our diversifying transformations by removing all potentially inserted NOP instructions from both instruction sequences. Since this step potentially makes the two instruction sequences more similar, our algorithm conservatively *overestimates* the number of gadgets surviving diversification. If the normalized instruction sequences are equivalent, then the algorithm has identified the candidate as a surviving gadget.

Using this strategy, we determined how many functionally equivalent gadgets exist at the same location in a pair of executables. These two properties are a requirement for a code reuse attack like ROP to work on multiple executables without modification. Because we used `.text` section offsets and not absolute addresses, we were able to perform our analysis in an environment where protections such as address space layout randomization (ASLR) [28] do not interfere with results.

We ran our surviving gadgets algorithm on 25 different versions of each SPEC CPU benchmark. Table 2 shows the results of our scans. The benchmarks are sorted by total number of gadgets in the original binary (the **Gadgets Baseline** column). Our scans show that, as the binary gets larger and contains more gadgets, the effectiveness of randomization also increases. On the largest benchmark, 483.xalancbmk, which contains over half a million gadgets, only 0.05% of the original gadgets survive (a reduction in gadgets of around $2000\times$). The impact of profiling on surviving gadgets is small, with the percentage of extra gadgets between 1% and 30% (473.astar is an outlier, as $p_{\text{NOP}} = 0-30\%$ gains a massive 254% extra gadgets; this is due to the large difference between $p_{\text{NOP}} = 50\%$ and $p_{\text{NOP}} = 30\%$, not from the profiling optimization itself). However, the increase in surviving gadgets is 30% out of 0.05% of the gadgets available in the original undiversified binary. Finally, we note that 407.lbm is very small; the undiversified C library assembly code is comparatively large to the program itself, increasing the relative percentage of surviving gadgets. This could be easily fixed in practice by also diversifying the C library code. Overall, the absolute impact of profiling on the number of surviving gadgets is negligible.

In practice, it is possible that the attacker is satisfied with successfully attacking some subset of targets. To that end, they will try to find the largest subset of gadgets common to as many binaries as possible, ignoring the undiversified program. To determine how much diversity there is in the binaries, we measured how many gadgets survive at the same location in at least 2 (10% of the population), 5 (20%) and 12 (50%) of the 25 versions. Table 3 shows the results of our scans.

The data provide several interesting insights. First, there are more gadgets in total in at least two binaries than there are in the original, undiversified binary. This is because a gadget at offset O in the original binary can be displaced to offset O_1 in some subset of binaries and to offset O_2 in another subset. Therefore, the same baseline gadget is counted several times in the diversified population, once for each offset. Second, adjusting p_{NOP}has a significant impact on gadgets surviving in at least two binaries (for example, for 400.perlbench, going from $p_{\text{NOP}} = 50\%$ to $p_{\text{NOP}} = 0-30\%$ increases the number of gadgets from $6,827$ to $11,117$, an increase of 62%). However, this is less significant when looking at a larger subset of the population; although 471.omnetpp shows an increase from 113 to 390 gadgets (a large percentual increase) surviving in at least five versions, this is a very small increase compared to the initial number of gadgets in the binary. Third, we observe that the number of gadgets surviving in at least half the binaries is essentially constant, regardless of the program size or diversification parameters. The remaining gadgets (around 40 in total) come from the small C library object files that the linker adds to the binary (which we can also diversify, given their source code).

To verify that profile-guided NOP insertion is effective against a concrete attack, we tested our diversification on a vulnerable application. We picked a popular network-facing application (PHP version 5.3.16) and ran two separate ROP gadget scanners to build ROP attacks against the undiver-

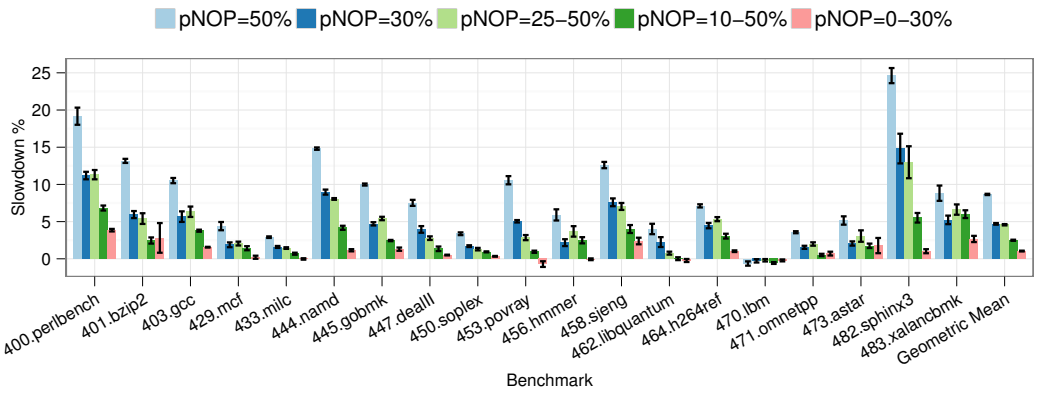

Figure 4. SPEC CPU 2006 performance overhead of NOP insertion.

Benchmark	Gadgets Baseline	p_{NOP} 50%	$25-50\%$	$10-50\%$	30%	$0-30\%$	Gadgets Extra%	Gadgets Surviving%
470.lbm	344	61.60	61.92	61.80	62.88	62.92	**2%**	**18.29%**
429.mcf	579	55.60	56.92	56.84	55.88	56.68	**1%**	**9.79%**
462.libquantum	709	52.32	52.28	52.28	52.28	52.92	**1%**	**7.46%**
401.bzip2	1191	16.00	16.24	16.24	16.36	17.40	**8%**	**1.46%**
473.astar	1362	16.64	18.56	22.24	46.20	59.04	**254%**	**4.33%**
433.milc	3149	17.16	16.92	17.28	17.16	17.44	**1%**	**0.55%**
458.sjeng	3317	15.08	16.00	16.04	17.24	17.44	**15%**	**0.53%**
456.hmmer	4535	15.56	15.60	15.84	15.84	16.20	**4%**	**0.36%**
444.namd	5322	38.48	39.12	39.60	42.72	43.24	**12%**	**0.81%**
482.sphinx3	6599	15.76	16.28	16.64	16.32	16.56	**5%**	**0.25%**
464.h264ref	16233	16.32	16.44	15.68	16.76	18.76	**14%**	**0.12%**
450.soplex	23885	23.32	23.88	24.84	25.96	24.48	**4%**	**0.10%**
447.dealII	24654	21.20	22.52	22.80	24.92	26.28	**23%**	**0.11%**
453.povray	41954	21.56	22.68	22.36	24.40	26.08	**20%**	**0.06%**
400.perlbench	43065	24.68	25.32	24.20	24.08	25.68	**4%**	**0.06%**
445.gobmk	56032	37.60	39.92	39.52	42.84	43.72	**16%**	**0.08%**
471.omnetpp	75246	45.28	47.20	48.08	49.56	59.16	**30%**	**0.08%**
403.gcc	105039	38.08	37.24	40.40	42.24	48.28	**26%**	**0.05%**
483.xalancbmk	566342	246.80	254.36	253.68	271.24	274.16	**11%**	**0.05%**

Table 2. Surviving gadgets on SPEC CPU 2006 binaries, showing the average number of surviving gadgets for each benchmark and NOP insertion strategy. **Gadgets Baseline** is the number of gadgets in the undiversified binary, **Extra%** is the extra number of gadgets in $p_{\text{NOP}} = 0-30\%$ compared to $p_{\text{NOP}} = 50\%$ (best-to-worst comparison) and **Surviving%** is the ratio of gadgets surviving diversification in $p_{\text{NOP}} = 0-30\%$.

sified PHP binary. We used two publicly documented ROP attack frameworks: ROPgadget [32] and microgadgets [14].

As a preliminary step, we verified that the undiversified PHP binary is indeed vulnerable to both these attacks; with both scanners, the program contained enough gadgets to allow the attacker to execute arbitrary code. Next, we built 25 diversified versions of the PHP interpreter, using the highest-performance, lowest-security setting: $p_{\text{NOP}} = 0-30\%$. We ran *Survivor* on each diversified version, then reran both ROPgadget and the microgadgets scanner on the surviving gadgets to verify the feasibility of an attack. On all diversified versions of PHP, a ROP-based attack was no longer possible, as the remaining gadgets did not provide the required operations for the attack.

Since there exists no standard profiler-friendly training input for PHP, we profiled several different PHP programs, then built 25 diversified versions for each profile. We used several benchmarks from the Computer Language Benchmarks Game [11]: binarytrees, fannkuchredux, mandelbrot, nbody, pidigits, spectralnorm and fasta. Each benchmark stresses different parts of the PHP interpreter (function calls, arrays, loop operations). None of the profiles produced any binary that we could successfully attack, using either ROPgadget or microgadgets.

6. Discussion and Future Work

Both the naive and the profile-guided NOP insertion are driven by probability parameters. At compile-time, the user of the diversifying compiler chooses the parameter. This choice presents several trade-offs. First, for software diversity to be effective, a sufficient number of versions must be available; the probability where a maximum number of ver-

	$p_{\text{NOP}}\%$														
	At least 2 versions					**At least 5 versions**					**At least 12 versions**				
Benchmark	50	$25-50$	$10-50$	30	$0-30$	50	$25-50$	$10-50$	30	$0-30$	50	$25-50$	$10-50$	30	$0-30$
470.lbm	586	608	614	602	723	140	173	56	175	180	50	50	46	50	50
429.mcf	1614	1722	1563	1663	1850	79	166	155	212	196	45	45	45	45	45
462.libquantum	871	819	849	1082	1229	137	47	50	152	62	41	41	41	43	41
401.bzip2	913	1085	1195	1145	1910	46	49	43	53	150	44	42	42	42	44
473.astar	1335	1373	1551	1580	2165	48	54	62	56	100	45	44	44	41	48
433.milc	2022	2014	2358	2496	3443	44	51	48	54	65	44	42	42	44	41
458.sjeng	1502	2110	2008	2927	3593	47	48	45	55	78	41	44	44	42	42
456.hmmer	1721	1829	1898	2427	2779	44	44	45	49	50	44	44	44	44	44
444.namd	2189	2449	2524	3509	4225	77	76	71	81	87	54	64	63	64	67
482.sphinx3	2315	2521	2426	5277	4589	42	45	45	51	55	42	44	44	44	44
464.h264ref	3639	4343	5163	7138	7216	46	42	47	46	56	44	41	42	43	49
450.soplex	6652	7300	6952	10314	11013	110	138	104	142	157	44	48	42	50	49
447.dealII	5764	7647	7723	8759	10550	48	53	55	65	66	44	44	44	44	47
453.povray	9878	9658	11002	12702	13425	70	66	67	63	77	44	44	44	41	49
400.perlbench	6827	10380	7935	8361	11117	47	55	46	52	68	44	48	44	42	40
445.gobmk	10896	11974	11898	14574	17739	58	53	63	81	108	42	44	43	44	42
471.omnetpp	17156	17523	17914	60388	29870	113	106	154	419	390	48	47	47	44	48
403.gcc	16825	16572	20443	19243	25343	90	161	160	113	104	16	16	43	16	16
483.xalancbmk	76765	79688	82053	102370	109543	77	108	103	181	186	42	42	16	16	44

Table 3. Surviving gadgets on SPEC CPU 2006 binaries, on a sample of 25 different binaries. The columns show the number of gadgets surviving in at least 2, 5 and 12 out of the 25 versions for each combination of benchmark and randomization parameter.

sions are available is $p_{\text{NOP}} = 50\%$. The number of versions decreases for both larger and smaller values of p_{NOP}. Second, the performance overhead is directly proportional to the probability of NOP insertion. While profiling reduces that overhead significantly, the developer must still choose an appropriate probability range.

We implemented and evaluated NOP insertion for 32-bit x86 microprocessors, using the available NOPs on this architecture. Essentially all instruction sets provide some flavor of NOP instruction, so our approach could easily be ported to other architectures. Many RISC architectures use fixed-lengths instructions together with aligned jumps, so that it is not possible to resume execution at arbitrary function offsets. This restricts the set of gadgets an attacker can use, making ROP-based attacks more complicated. However, recent work has showed that practical ROP attacks are feasible on RISC machines [4, 7, 19].

Compilers may implement other techniques, such as instruction, basic block and function reordering, basic block shifting, register randomization and equivalent instruction substitution. A compiler may use all these available techniques to improve security, as most of them are orthogonal and operate at different granularities. In addition, profile-guided optimization can be used to reduce the performance impact, by taking into account the heat of the code when randomizing registers or reordering instructions. Also, some techniques can be used in code sections where other techniques provide less security. For example, NOP insertion adds much less diversity at the beginning of the program than its later parts, due to the way displacements from NOPs accumulate over consecutive instructions. To offset this, a compiler can also perform basic block shifting, which inserts a dummy basic block of random size at the beginning of each function, or adjust its NOP insertion to add more NOPs at the beginning. If the function jumps over the initial basic block of NOPs, its performance impact should be minimal. However, its presence should prevent the attacker from exploiting the low diversity at the beginning of the binary.

7. Related Work

Most compilers use profiling information to optimize frequently used sections of code. This has the goal and effect of increasing program performance and, in some cases, reducing program size. However, there are other applications of profiling where the interest lies in infrequently executed code. One such application is code compression. Debray and Evans [8] analyze the use of profiling in tandem with binary code compression, using basic block execution frequency to decide whether to compress particular regions of code. They showed that focusing compression on cold code produces significant reductions in program size.

Another use case for identifying cold code is intentional code duplication. Recent research uses this technique to identify computational errors due to transient faults in processors [18]. The work by Khudia et al. duplicates values and instructions in a program, to account for transient processor errors. The duplicates perform the same computations as the originals, so their results should match. At specific points in the program, the duplicates are checked against the originals to detect errors. The compiler uses edge profiling to reduce number of duplicated instructions, minimizing the performance impact of this approach. When only duplicating cold instructions, they show a reduction of 41% in overhead from previous techniques.

Software diversity is one strategy to defend against code reuse attacks. Other strategies rely on detection or prevention of such attacks during the execution of the targeted program,

or using static compile-time techniques. Detection software runs alongside the program and attempts to detect whenever a ROP or other code reuse attack is in progress. DROP [5] dynamically instruments a running program to analyze the frequency of taken returns; whenever the number of returns taken in a short amount of time exceeds a threshold, DROP considers a ROP attack is in progress and takes measures against it.

DieHard [2] is an implementation of software diversity targeted at memory errors. By randomizing the layout of the application heap, DieHard provides probabilistic protection against attacks like heap buffer overflows and double frees. As memory randomization focuses on a different stage of an attack from code randomization, the two techniques complement each other.

"Return-less kernels" [22] target ROP attacks specifically during compilation, by reordering program instructions and re-allocating registers so that free branch instructions (a prerequisite for return-oriented programming and similar code reuse attacks) never appear inside a binary. G-Free [26], control-flow locking [3], control-flow integrity [1] and software fault isolation (SFI) [23, 37] take a combined compile- and run-time approach to security. The compiler adds code to the program that restricts the control flow to the program's original control flow edges, by instrumenting all branch instructions. While these techniques have proven effective at defending against ROP, any inherent weakness they have can be exploited by attackers. Also, some are specifically targeted at return-oriented programming, so are unable to defend against newer code reuse attacks. This incurs a maintenance cost on whoever implements these techniques, as each implementation must be updated for any new code reuse attack, if at all possible.

Some of these techniques incur much larger overhead than our profile-guided NOP insertion technique. However, many of them rely on code insertion at specific places, just like our approach. Therefore, it is possible that they too might benefit from profile-guided optimization, by inserting checks and guards selectively.

8. Conclusion

Software diversity, in the form of code layout randomization, has a significant impact on the outcome of code reuse attacks. Attacks against a sufficiently diverse program have a high chance of failure. NOP insertion in particular is an effective form of code layout randomization. However, a naive implementation suffers from a moderate performance overhead. Selective insertion of NOPs using profiling data reduces this overhead by as much as $5\times$. NOP insertion is particularly efficient against return-oriented programming attacks, reducing the number of available gadgets by a factor as high as $2000\times$. The remaining gadgets are, in practice, not sufficient for an attack. This holds true even when using profile-guided NOP insertion. Profile-guided software diversification has a minimal impact on performance, while hardening programs against a code reuse attacks.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback and suggestions, Stephen Crane for assistance and comments on earlier drafts of this paper, and Todd Jackson for providing his original version comparison implementation.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. D11PC20024. 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 Defense Advanced Research Projects Agency (DARPA) or its Contracting Agent, the U.S. Department of the Interior, National Business Center, Acquisition Services Directorate, Sierra Vista Branch.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, 2005.

[2] E. Berger and B. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, PLDI '06, pages 158–168, 2006.

[3] T. Bletsch, X. Jiang, V. 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*, ASIACCS '11, pages 30–40, 2011.

[4] 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*, CCS '08, pages 27–38, 2008.

[5] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In A. Prakash and I. Sen Gupta, editors, *Information Systems Security*, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer Berlin / Heidelberg, 2009.

[6] F. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, Oct. 1993.

[7] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy. Return-oriented programming without returns on ARM. Technical report, System Security Lab, Ruhr University Bochum, Germany, 2010.

[8] S. Debray and W. Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 95–105. ACM, 2002.

[9] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on*

Hot Topics in Operating Systems, HotOS '97, pages 67–72, 1997.

[10] Free Software Foundation, Inc. GCC compiler internals, 2012. URL `http://gcc.gnu.org/onlinedocs/gccint/`.

[11] B. Fulgham. The computer language benchmarks game. `http://shootout.alioth.debian.org/`, 2012.

[12] 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 Security Symposium*, pages 475–490, 2012.

[13] 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*, S&P '12, pages 571–585, 2012.

[14] A. Homescu, M. Stewart, P. Larsen, S. Brunthaler, and M. Franz. Microgadgets: Size does matter in Turing-complete return-oriented programming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies*, WOOT '12, 2012.

[15] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani. Mao – an extensible micro-architectural optimizer. In *Proceedings of the 9th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, 2011.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, August 2012.

[17] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium*, pages 459–474, 2012.

[18] D. S. Khudia, G. Wright, and S. Mahlke. Efficient soft error protection for commodity embedded microprocessors using profile information. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, pages 99–108, 2012.

[19] T. Kornau. Return-oriented programming for the ARM architecture. Master's thesis, Ruhr University Bochum, Germany, 2009.

[20] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation techniques, 2005. URL `http://www.suse.de/~krahmer/no-nx.pdf`.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '04, pages 75–87, 2004.

[22] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 195–208, 2010.

[23] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th USENIX Security Symposium*, pages 209–224, 2006.

[24] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine*, 11(58), 2001. `http://www.phrack.org/issues.html?issue=58\&id=4`.

[25] A. Neustifter. Efficient profiling in the LLVM compiler infrastructure. Master's thesis, Faculty of Informatics, Vienna University of Technology, 2011.

[26] 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*, ACSAC '10, pages 49–58, 2010.

[27] 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*, S&P '12, pages 601–615, 2012.

[28] *Homepage of The PaX Team*. PaX, 2009. `http://pax.grsecurity.net`.

[29] M. Payer. Too much PIE is bad for performance. Technical report, ETH Zurich, 2012. URL `http://nebelwelt.net/research/publications/tr-pie12/`.

[30] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, 1990.

[31] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69, 2009.

[32] J. Salwan. ROPgadget tool, 2012. URL `http://shell-storm.org/project/ROPgadget/`.

[33] 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*, CCS '07, pages 552–561, 2007.

[34] H. Shacham, M. Page, B. Pfaff, E. 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*, CCS '04, pages 298–307, 2004.

[35] L. Tang, J. Mars, and M. L. Soffa. Compiling for niceness: mitigating contention for QoS in warehouse scale computers. In *Proceedings of the 10th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '12, pages 1–12, 2012.

[36] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, RAID '11, pages 121–141, 2011.

[37] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, S&P '09, pages 79–93, 2009.