Software Profiling Options and Their Effects on Security Based Diversification

Mark Murphy
University of California, Irvine
msmurphy@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

Imparting diversity to binaries by inserting garbage instructions is an effective defense against code-reuse attacks. Relocating and breaking up code gadgets removes an attacker's ability to craft attacks by merely studying the existing code on their own computer. Unfortunately, inserting garbage instructions also slows down program execution. The use of profiling enables optimizations that alleviate much of this overhead, while still maintaining the high level of security needed to deter attacks. These optimizations are performed by varying the probability for the insertion of a garbage instruction at any particular location in the binary. The hottest regions of code get the smallest amount of diversification, while the coldest regions get the most diversification.

We show that static and dynamic profiling methods both reduce run-time overhead to under 2.5% while preventing over 95% of original gadgets from appearing in any diversified binary. We compare static and dynamic profiling and find that dynamic profiling has a slight performance advantage in a best-case scenario. But we also show that dynamic profiling results can suffer greatly from bad training input. Additionally, we find that static profiling creates smaller binary files than dynamic profiling, and that the two methods offer nearly identical security characteristics.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, optimization*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*unauthorized access*

MTD'14, November 3, 2014, Scottsdale, AZ, USA

ACM 978-1-4503-3150-0/14/11...$15.00
http://dx.doi.org/10.1145/2663474.2663485.

General Terms

Performance, Security, Measurement

Keywords

Static Profiling, Dynamic Profiling, Automated Software Diversity, Code Randomization

1. MOTIVATION

Software profiling is a proven technique for identifying which blocks of program code execute the most often [2, 17]. This information can then be used to implement a variety of optimizations. Likewise, the diversification of binary code [18] through the insertion of garbage instructions has been shown to be an effective defense against code-reuse attacks [14, 15]. These garbage instructions displace the original instructions in the diversified binary, making any particular piece of code difficult to find for an attacker.

Inserting garbage instructions, however, makes the diversified binary larger and slower than the original. Homescu et al. [14] showed that profile-adaptive diversification reduced the slowdowns to a negligible level. In their study, the use of a profile-guided diversifying compiler that inserted garbage instructions into the binary code resulted in run-time overheads of about 1% while successfully implementing code diversification.

Previous work, however, has only demonstrated the use of one form of profiling in this process, dynamic [14, 15]. Dynamic profiling, at its best, can give a very accurate picture of which part of a program is "hot" or "cold", but the profile that it creates can also be very misleading. The accuracy of a dynamic profile depends on the quality of a training run. If the input for that training run is representative of the typical use of a program, then the profile will likely be accurate and optimizations will produce the expected benefits. However, if the input for the training run is not representative of a typical run, the profile will not be accurate, misidentifying which code regions are hot or cold. In consequence, optimizations cannot yield their full potential. The representative input sets, needed for an accurate training run, could be non-representative, could be unavailable, or the program's use may just not lend itself to the creation of an accurate profile from a single set of inputs. If any of these things are true, performance will suffer and the potential improvements from profiling will not be achieved.

Static profiling, although it does not deliver as precise a

profile, does not have the potential problems of dynamic profiling. Since a static profile is formed from the examination of the control structures in the code itself, no training run is needed. We will show that by using static profiling instead of dynamic profiling, the downside of dynamic profiling can be avoided while still realizing many of the positive aspects that it provides. Performance gains comparable to those from the best-case of dynamic profiling can be achieved with static profiling while still maintaining the same level of security through the diversification of binary code.

2. BACKGROUND

2.1 Code-Reuse Attacks

Instead of inserting an attack program into a victim's machine and executing it there, code-reuse exploits use pieces of code that are already on a victim's machine. Return-to-libc attacks place the addresses of existing functions on the stack [23], while return-oriented programming (ROP) attacks place addresses of smaller pieces of code that end with a return (called "gadgets") [27] on the stack. Jump-oriented programming uses gadgets like ROP, but does not rely on the stack and return statements [5]. Blind ROP (BROP) is a method that uses brute force probing of the binary code to disclose code locations and then launches a ROP attack [3]. All of these methods work by getting the victim machine to execute its own code to serve the attacker's purposes. Since return-oriented programming seems to currently be the most commonly used code-reuse attack [22], we will concentrate our discussion on ROP.

In ROP, the attacker's goal is to get the CPU to load gadget addresses into the Program Counter (PC) and to control the flow of execution. Any sequence of machine instructions that leads to a return, whether using the original instructions or discovering new ones from portions of existing code (as in Figure 1) can be used. In order to succeed, ROP attacks rely on the following three things:

- The lack of diversity in distributed software
- The non-aligned instructions in x86 architecture
- The function return mechanism

The first thing that the attacker needs to carry out a ROP attack is the knowledge of what files will be loaded into memory on the target machine and the exact addresses of the gadgets inside those files. This knowledge is actually made possible by the current nature of software. When software developers distribute their products, they deliver the same code to all consumers. Because of this, an attacker can simply examine the files on their own machine to discover sequences of usable gadgets, knowing that there are millions of other machines out in the world with the exact same files containing the exact same code. This lack of diversity in software was identified as a potential threat to computer security as early as 1993 by Cohen [7], and it remains a large factor in making code-reuse attacks possible today.

The second thing that aids in the construction of ROP attacks is the non-aligned nature of x86 architecture. In the x86 architecture, some machine instructions can be as short as a single byte, while others may be many bytes in length. This representation is space-efficient and does not require an entire word of memory to represent an instruction that can be completely described in fewer bytes. But at the same time it means that instruction execution has to be able to start at any given byte address instead of just at aligned word addresses. Figure 1 illustrates this point, showing how gadgets could potentially be found starting at any byte address as long as they eventually lead to a return instruction. The numbers in the middle line of the figure are a hexadecimal representation of binary machine code. The assembly language instructions above the machine code show the sequence that would be executed under normal circumstances. The assembly code written below the machine code shows another possibility, where execution has been directed to start at a location that is inside the original MOV instruction. In this case, the targeted byte also contains legal op-code information, so execution will successfully begin at this location. The new binary sequence that begins at this address (11 01 C3) represents a gadget containing the ADC and RETN instructions. It is perfectly valid code even though it is totally different from the original and so this discovered code can be executed if its location is known. This shows that the ROP attack does not necessarily even need to find all of its tools at the end of existing functions. It can discover code inside other instructions that can be manipulated to act as returns. While the ability to discover new gadgets is not essential to the creation of a successful ROP attack, it does make an attack easier to craft by increasing the number of available gadgets.

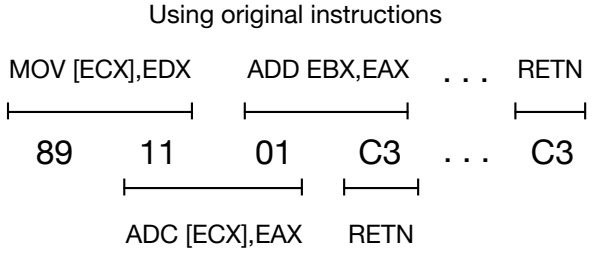


Figure 1: A discovered gadget

In Figure 1, an executable instruction can be found by beginning at any of the bytes that are shown. This is not always the case, as many byte sequences do not form legal instructions, but it illustrates the point that gadgets are very common. In fact attackers have so many gadgets available in existing files that ROP attacks have been shown to be Turing complete, meaning that ROP attacks can piece together gadgets to produce any machine code that a programmer could write.

Once a sequence of usable gadgets has been discovered, the attacker just needs to put the gadget addresses on the victim computer's stack and get that computer to start executing its own code. By having knowledge of the code that already resides on the computer and co-opting the return mechanism, the attacker can redirect the flow of execution to any series of gadgets that reside in the machine's memory.

The third thing that makes a ROP attack possible is the function return mechanism itself. When a function call is encountered in the execution of a program, before the execution of the function begins, that function's return address is placed on the stack. When the function finishes running, the return address, which was stored on the stack, is loaded back into the PC. In this way execution returns to the in-

struction that immediately followed the function call in the code. ROP attacks hijack this mechanism, placing other addresses on the stack that the machine interprets as return addresses. Thus each time the computer attempts to return from a function, it actually winds up jumping to and executing the gadget it finds at the location of the next address on the stack.

2.2 Diversification

Code-reuse attacks depend on predictability. The attacker crafts their exploit using known addresses from an examination of the code on their own computer. If the loaded files on a victim's computer are not the same, at the binary level, as the files that the attacker used to create their attack, then the attack is likely to fail. The targeted gadgets, if they exist at all, are unlikely to be found in the same locations on any two machines.

Address Space Layout Randomization (ASLR) is a diversification process that is currently implemented in some form in all major operating systems. In ASLR, modules are loaded into randomized memory locations instead of being placed in the exact same place every time. With this move towards diversity in computers, all potential victim machines appear to be different to attackers and different from the attacker's own machine as well. However, attackers have found ways to have the operating system leak the information about where the desired modules are located in a victim's machine [28, 11, 26, 30]. And, once the attacker knows the location of the files, the addresses of all of the code gadgets within those files are easily calculated since the code inside all of the files is still identical to the attacker's own version. Fine-grained attempts at randomization take the diversification one step further by relocating the functions that exist inside the files [13]. But once again, the locations of the functions can be leaked and the actual code inside the functions is unchanged, so gadget addresses can be found as offsets from the beginning of the functions [29]. Once the locations of the gadgets are known, the attack can proceed as before.

The introduction of ASLR into operating systems was a step in the right direction for providing security through diversity, but has proved to be an insufficient impediment to code-reuse attacks. By inserting garbage instructions into the binary code, diversity can be extended down to the instruction level, so that vulnerabilities that currently exist in the same code on every machine are broken up or hidden more effectively.

Brute force code reuse attacks, such as BROP, require special handling. Because they can use information leakage to locate gadgets in the binaries, just moving the gadgets around is not enough to stop these attacks. The diversifying compiler currently uses booby traps [9] to counter this type of exploit, but we believe that further diversity could also be used to counter this type of attack. BROP depends on the forking mechanism to produce an exact duplicate of the current process, so diversifying when forking would remove this required feature of the attack.

2.3 Diversity Through NOP-Insertion

The insertion of garbage instructions into a binary can theoretically produce an infinite number of distinct binaries from one original file. We will use the general term of "NOPs" for these garbage instructions. They occupy space

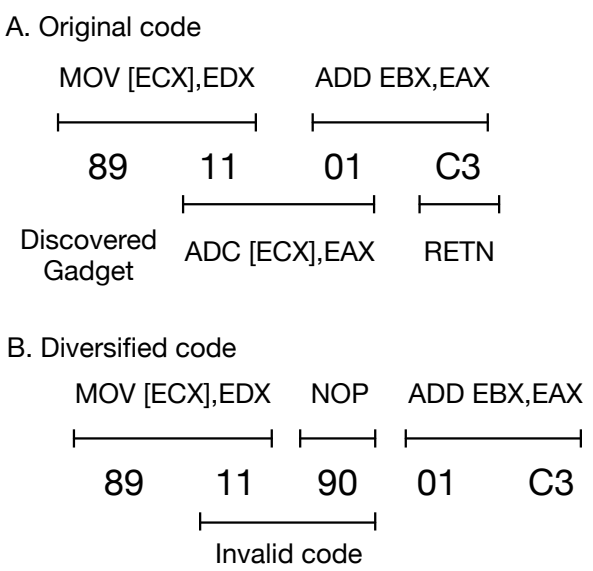


Figure 2: Breaking the discovered gadget

in the code, but do not affect the state of the machine. Therefore the created binaries are all functionally equivalent, performing the same tasks, but internally they are all unique. This massive diversification makes code-reuse attacks that do not include information leakage impractical. Even if the attacker gets around the ASLR employed by the operating system, the code gadgets that they need to carry out their attack are either not present at all or are located in some other unknown place in memory.

Figure 2 demonstrates how the NOP insertion mechanism works to relocate some gadgets and break up others. In part A, the code from Figure 1 is repeated, showing how the original code was vulnerable to ROP attacks. In part B, a NOP instruction has been inserted between the MOV and the ADD instructions. One effect of this insertion is that the gadget that was created by starting execution in the middle of an instruction (11 01 C3) has been removed entirely. That code, that the attacker may have needed in order to perform their attack, no longer exists in this file. The code that now begins at that address is not a valid instruction. It is possible that the inserted NOP could create a new valid gadget, but that new gadget would not exist in the attacker's machine, so they would have no knowledge of it. The next thing to notice from this figure is that a gadget that began with the ADD instruction has been moved forward to a new address in the code. This displacement means that if the attacker wanted to use the ADD instruction as the start of a gadget, they wouldn't find it in its original position. In this particular figure, an attack starting at the original ADD location could still work, since the NOP will be executed and then the desired ADD will still be found as the next instruction. But every time a NOP is inserted, every instruction that follows that NOP is moved to a new address, so, there is a cumulative effect of instruction spacing caused by NOP insertion. Because of this, there is a very small likelihood that execution starting at the address of any original gadget will perform the action desired by the attacker.

The creation of multiple binaries that are all different is

```
for i in instructions:
    pct = findInsertionPct()
    r = random(0.0,1.0)
    if r < pct:
        numNOPs = random(1,MaxNops)
        for k in range(1, numNOPs):
            choice = random(1,size(NopList))
            insert(NopList[choice])
```

Figure 3: NOP insertion algorithm

accomplished through the use of a random number generator. When given the same seed each time, a pseudorandom number generator will produce exactly the same series of values, while still producing a completely different set of values for a different starting seed. So, randomization is achieved while still preserving repeatability.

Figure 3 shows how NOP insertion is implemented. For each instruction, the insertion percentage is determined, and if a random value is less than this percentage, then one or more NOP insertions will occur at that point. Additionally, for each instance of a NOP insertion, another random number is generated to choose which specific NOP should be used in that location. These candidate machine instructions are all just one or two bytes in length. Other machine instructions that function as NOPs were available, but we chose these five for the purpose of minimizing the inserted code's impact on file size and speed of execution.

Having a variety of NOP instructions to choose from further increases diversity. The candidate instructions in the NOP list are shown in Table 1.

Table 1: NOP candidates

Instruction	Machine Code
NOP	90
MOV ESP,ESP	89 E4
MOV EBP,EBP	89 ED
LEA ESI,[ESI]	8D 36
LEA EDI,[EDI]	8D 3F

The insertion percentage, for the fixed-percentage diversifying compiler, is assigned at compile time and used throughout the code. If an instruction occurs early in the code, however, the insertion percentage is raised to increase the diversity level at the beginning of the file. The profile-guided compiler uses a more complex determination for the insertion percentage, as will be discussed in section 3.2.

The diversifying compiler uses a different seed for the random number generator for each different binary version. In this way, the three random choices in the algorithm in Figure 3 get pseudorandom sequences of values, creating diversity, but the choices made at those locations are also completely repeatable by the compiler, using the same seed.

The seed value is also key in the maintenance of programs, enabling updates and patches. Because each machine has its own version of a particular file that is completely different internally from all others, the same patch can't be sent to everyone. Instead the developer uses the seed value from a customer's version to reproduce that binary, and then sends the patch that is specific to that version. Precompiling multiple versions in the cloud is relatively inexpensive [12].

The insertion of a large number of NOPs might be expected to greatly increase the size of diversified files and slow down execution markedly. However, the use of small, quick executing NOPs and the use of profiling minimizes the increases in file size and execution time.

3. PROFILING

3.1 Types of Profiling

There are two types of software profiling that are available in the LLVM compiler [19] : static and dynamic [24]. Static profiling is an analysis that is performed at compile-time, before the target program is executed. It creates a profile that is based directly on the control flow graph (CFG) of the file that is being compiled. The profile is created from the CFG by using heuristics to estimate which basic blocks in the graph are likely to be encountered the most frequently during execution. A simple example of this is that a block of code inside a loop would be identified as hotter than a block of code in a simple if-statement, due to the expectation that the loop block will be executed more than once.

The second method, dynamic profiling, inserts counters at key locations in the code, compiles the code, and then executes a training version of the program. The values in the counters at the end of the training run accurately represent how often each basic block was actually executed in the training run, and are used to form the profile. That profile, based on the observed results of the training run, is then used to optimize the compilation of the full version of the program.

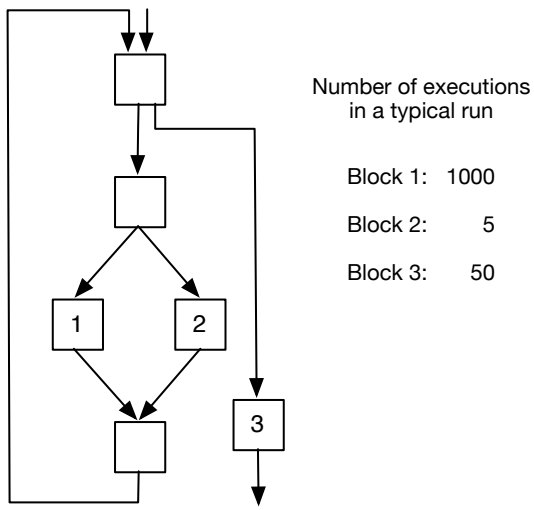


Figure 4: A sample control flow graph

Figure 4 shows a sample control flow graph where some dynamic counts have already been determined. If the CFG from a training run represented these same tendencies, the dynamic profile would indicate that the instructions in basic block 1 were much hotter than those in the other two numbered basic blocks shown in the diagram. As such, the greatest amount of optimization would correctly be focused on the instructions in basic block 1. The least amount of optimization would be applied to instructions in basic block 2, and the optimization level for the instructions in basic

block 3 would fall somewhere in between. Dynamic profiling does not, however, always produce the fastest executing code. If the input for the training run caused the execution to encounter block 2 more often than block 1, then the resulting profile would not be representative of the values shown in Figure 4 for a typical run. Worse yet, the training input set could cause execution to skip the loop entirely in the training run, causing the program to execute only block 3. Either of these last two scenarios would result in an inaccurate profile, causing optimization to be focused on areas that are rarely executed in a typical run. This would result in little or no optimization being performed in the part of the code that would actually be executed the most often. Having a training run and a training input set that is representative of normal use is therefore key to getting the best performance out of dynamic profiling.

If a static profiler were to see this same CFG, it would not already have an estimate of the number of executions for different basic blocks. Instead, it would likely guess that all of the code inside the loop was hot and seek to optimize both blocks 1 and 2. Because program input is not known at compile time, it is difficult for the static profiler to predict which branches in a CFG will be executed often and which will not, so static profiling can sometimes over-predict code hotness, classifying a seldom-executed basic block as hot. Thus, in some cases optimization may be applied to blocks where it will have little effect. However, unlike the dynamic profiling situation where the hottest code segments could miss out on optimization entirely, static profiling will still identify the hottest segments as hot and apply optimization to them.

3.2 Profiled Optimization with NOP insertion

Section 2.3 describes how the diversifying compiler produces software diversity, using random numbers to decide if a NOP should be inserted before any instruction, and if so, then which one. An earlier study [14] showed that the maximum diversity imparted to a file by the diversifying compiler was with a 50% NOP insertion probability. But this rate of NOP insertion causes unacceptable execution slow-downs, so optimizations are needed to allow the diversified binaries to run faster.

The answer to the run-time overhead problem is to use software profiling to determine which basic blocks are hot and which are cold. Then optimization can be focused on the code in the hot blocks. The Pareto Principle, when applied to software optimization, states that 80% of a program's execution time is spent in only 20% of the code [17]. This implies that for best results, optimizations to reduce execution time should be focused mainly on the smaller portion of hot code, since improvements in these sections will have the biggest impact on execution speed.

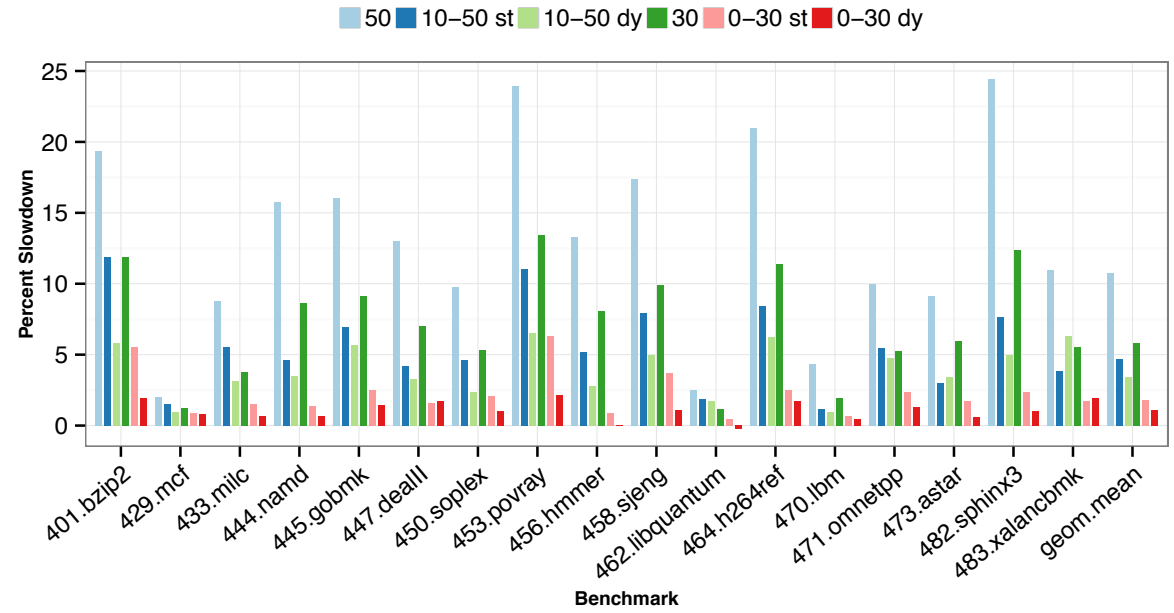
In the diversifying compiler, the optimization that profiling makes possible is the use of ranges of insertion probabilities rather than just fixed probability values. For instance, instead of using a 50% insertion rate for the probability of placing a NOP before a given instruction, a range of insertion probabilities from 10% to 50% can be used. Using profiling results, a logarithm-based formula [14] is used in the `findInsertionPct` stage of the algorithm from Figure 3 to determine exactly which insertion rate to use for each instance. For the code blocks that profiling has identified as containing the hottest code, the 10% insertion level is used. For the blocks determined to be cold, the maximum 50% level is used, and other blocks that are found to be somewhat hot, but not the hottest, get some rate between 10 and 50. So, by the Pareto Principle, an estimated 80% or more of the code gets near maximum diversification, and only a small portion of the code will have a reduced percentage of NOPs inserted. Thus, the blocks of code that execute the most often have a very small increase in size and a very small slowdown, reducing the run-time overhead to acceptable levels. The less frequently executed blocks of code, making up the majority of the program code, get the highest rate of diversification. Thus, compilation with profiled NOP insertion achieves the security goal of code diversification while also reducing execution time.

4. RESULTS

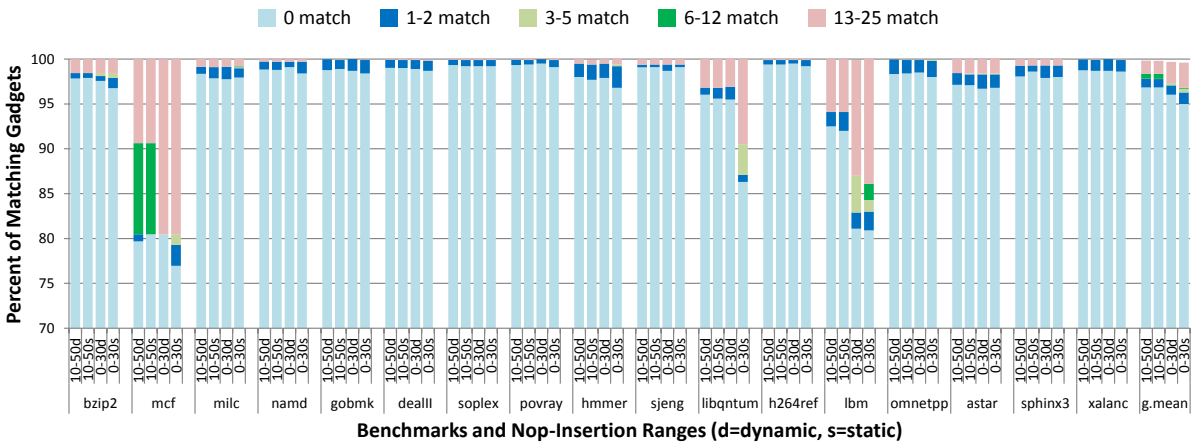
In this study we compare the results from the static and dynamic profiling methods on three criteria: run-time overhead, their effectiveness at removing gadgets, and file size. For experimental data, we compiled seventeen benchmarks from the Spec2006 suite with both dynamic and static profiling. We built each of the benchmarks using five different seed values to create different distributions of the NOPs inside the binaries. Then we ran each benchmark three times for each seed value and NOP insertion range. Previous work [14] demonstrated that the ranges of 10%– 50% and 0%–30% were good ranges for dynamic profiling results, so those same ranges were also used in this study. The diversification was implemented as a back-end pass in LLVM 3.1 in the previous study. Our implementation uses the same compilation method, but with a more recent version, LLVM 3.4. All of the profiling results were found using the options that are available in this compiler.

4.1 Execution time

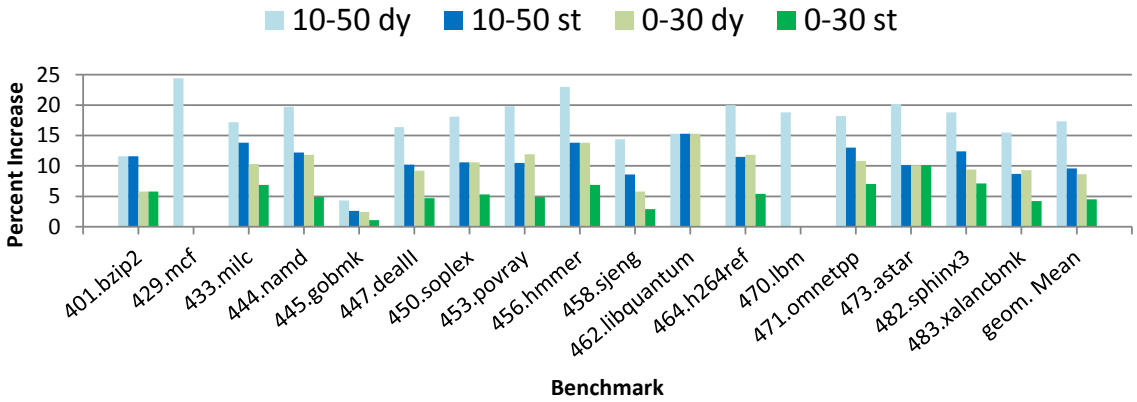
Figure 5a shows the average run-time slowdown results from the test runs, including results achieved with both dynamic profiling and static profiling. The key feature to see from both types of profiling is the relationship between the fixed percentage settings and the ranged settings with the same maximum value (50 vs. 10-50 and 30 vs. 0-30). These results show that both dynamic and static profiling optimizations produce large speedups in execution time by using the probability ranges. In the mean case, the ranged version slowdown was less than half of the fixed percentage slowdown for each range. Several benchmarks that were some of the slowest with the fixed insertion percentages showed the biggest improvements with the use of ranges. These include `namd`, `dealII`, `h264ref`, and `sphinx3`. The data shows that dynamic profiling dropped the execution time overhead for the geometric mean from a 12.8 percent slowdown to 3.95 percent for the 50% insertion rate and 10–50 range respectively. Similarly the overhead was reduced from 7.31 percent to 1.08 percent for the 30% insertion rate and the 0–30 range. For static profiling, the data shows that similar overhead reductions occurred, dropping the geometric mean value for 50% insertion from 12.88 percent to 5.54 with the 10–50 range, and lowering the slowdown for the 30% insertion rate from 6.92 percent to 2.23 for the 0–30 range. Comparing the dynamic profiling results to the static results shows that most of the runs that used dynamic profiling were faster than the corresponding runs using static profiling. Of the 34 runs for each version, the graph shows only four cases



(a) Slowdowns due to diversification



(b) Percentage of gadgets surviving in diversified binaries



(c) Increase in file size due to diversification

Figure 5: Test results from diversified binaries

where the static run was faster than the corresponding dynamic run. The differences between static and dynamic profiled results were small, however, when compared to the differences between unprofiled and profiled results. The mean static profiled run for the 10–50% range had a slowdown of 5.54% compared to the dynamic value of 3.95%. For the 0–30% range, the static mean value was 2.23%, in comparison with the 1.08% from dynamic. So the mean run-time cost of using static profiling instead of dynamic profiling was only about 1.5% of the overall execution time.

4.2 Security

In order to see how well the diversification process performed in preventing gadget transmittal to the diversified binaries, we compiled 25 diversified versions of each benchmark, using a different random seed for each one. We then compared the 25 binary files to the original and correlated how many original gadgets existed in the diversified files. The results of these comparisons are shown in Figure 5b. The vertical axis of this graph shows the percentage of original gadgets that appear in any of the 25 diversified files. Separate sections of the columns represent the percentage of gadgets found in none of the 25 diversified files, in 1 or 2 of the files, in 3–5 files, in 6–12 files, or in 13–25 files. The first thing to note about this graph is that the color representing zero matches (no shared gadgets) dominates for every benchmark. Over 75% of the gadgets in every original benchmark file appear in none of the diversified files. In fact the geometric mean for the 0 matches category is over 95% for every variety of profile-guided diversity.

The graph also shows some outlier behavior. The category that represents the highest amount of gadget sharing (13–25) appears in every column, and three benchmarks, `mcf`, `lbm`, and `libquantum` show a larger percentage of remaining gadgets than the rest of the benchmarks. The actual counts used to make this graph show that at least 22 original gadgets survived in every setting for every benchmark in all 25 runs. We believe that these surviving gadgets are coming from assembly code that exists in the C library. The assembly code is not compiled and thus does not get diversified by the compiler, but still gets linked into the final program. This small number of gadgets that are surviving into the diversified binaries also explains the apparent weaker security in the `mcf`, `lbm`, and `libquantum` benchmarks. These three benchmarks have the smallest file sizes of those tested, and thus have a smaller number of original gadgets. Therefore the ratio of the number of surviving gadgets to the number of original gadgets is higher than in the other larger benchmarks, and so the same number of assembly survivors shows as a larger percentage. Another possible effect of their small file sizes is just that they have less code and so they contain less overall diversification.

Comparing the graph columns representing the dynamic profile runs to those representing the static profile runs indicates that there is little difference between the two forms of profiling with regards to gadget survival. For the 10–50% probability range, both dynamic and static had exactly the same mean results, stopping 96.8% of all original gadgets from appearing in any of the diversified binaries. The 0–30% range showed only a very slight difference in the mean, with dynamic profiling stopping 96% and static stopping 95% of all original gadgets from appearing in the diversified files.

4.3 File Size

Figure 5c shows the changes in file size of the compiled benchmark binaries after diversification, for the same profiling ranges as previously used. Change in file size gives some indication of the amount of NOP code that is being inserted into each build, and thus gives some sense of how much internal diversity is being added. This graph shows that the files from the dynamic profiling builds are the same size or larger than those from the corresponding static profiling builds in every case. This indicates that dynamic profiling is adding more NOP code during the compilation process than static profiling.

For efficiency reasons, the linker aligns on-disk sections of the binary to the memory page size of 4k so that they can be loaded directly into memory. This coarse granularity in reported file size causes some issues with the smaller benchmarks. For example, in `mcf`, with an original file size of only 16k, the dynamic version of the 10–50% build expanded the file size from four pages to five pages, showing a file size increase of 4k. But, even though NOP insertion also occurred in the static version, it was not enough to expand the code beyond the limit of the fourth page, and so the file size is reported as being unchanged. The same zero growth indication is seen in the data for the other small files as well. But, while the file size increments used to make the graph do not yield a totally accurate representation of the amount of binary code in the files, they do still clearly illustrate the overall trend that compilation using dynamic profiling produces larger increases in file size than when using static profiling.

5. DISCUSSION

In the absence of information leakage, the random insertion of NOP instructions into program binaries, as performed by the diversifying compiler, is an effective method for thwarting ROP and other code-reuse attacks (Section 2.2 mentions the extra steps needed to counter attacks that do include information leakage). Gadgets that exist in the original files are relocated or broken up in each binary by the diversification process. However, the insertion of extra code into a file causes that program to run slower. At the 50% insertion setting, Figure 5a shows a mean slowdown of over 10%, with some benchmarks slowing down by over 20%. The use of software profiling eliminates the majority of these slowdowns while still maintaining high levels of security. Figure 5b shows that an average build prevents over 95% of all gadgets from being passed on to diversified files regardless of which profiling method is used.

Comparison of the static and dynamic profiling methods, the two alternatives available in the LLVM compiler, shows that the results are very similar with respect to execution speed. Static profiling, results in binaries that run slightly slower than those created using dynamic profiling, but the difference in slowdown is only about 1.5% of the total program run time, a small amount when compared to the overall speedup that profiling provides in the diversification process. Also, these results for dynamic profiling were found using benchmarks that have very good training data, yielding accurate profiles. A poor training set could produce a profile that does not represent how the program normally runs, and optimization using that profile would result in a binary that would run nearly as slowly as an unprofiled version.

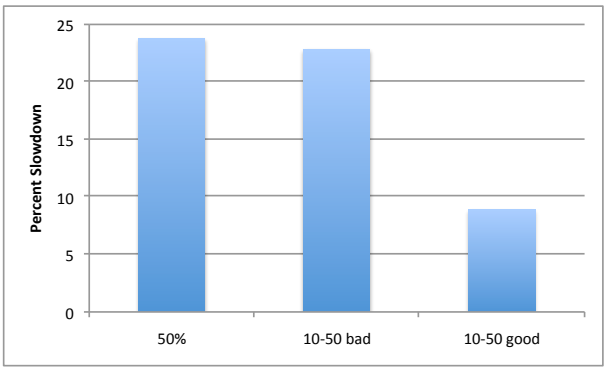


Figure 6: Dynamic profiling with good and bad training input

Figure 6 illustrates this point. Here a deliberately bad set of training data was used as the input for the `perlbench` benchmark. The figure shows that with the bad training input, the compiled binary, created using dynamic profiling, had a 22.9% slowdown, versus a 23.8% slowdown from the unprofiled run; almost no improvement. The dynamically profiled version with the standard Spec2006 input, at 8.9%, showed less than half of the slowdown of either of these. Static profiling results, which are not subject to the same downsides as dynamic profiling, compare very favorably to this kind of dynamic result.

The security comparison of the two profiling methods shows that profiled NOP insertion, in general, is very effective at preventing the transfer of gadgets from original benchmark code to compiled binaries, stopping 95% in all cases. But it also shows that there is very little difference between the static and dynamic diversification techniques with respect to how they protect against gadget sharing. For the insertion range of 10–50%, the mean values showed no difference between static and dynamic results, and for the 0–30% range the dynamic versions stopped only 1% more gadgets than did the static versions. The size of the file being compiled was a much bigger factor in what percentage of gadgets survived the diversification process.

The overall difference in the performance of the two profiling methods can be explained by understanding how the different methods predict the location of hot code. The static method tends to over-predict which code will be executed the most frequently since, without knowledge of the program input, it may consider any repetition equally likely to be hot. Since fewer NOP instructions are inserted into hot sections of code, this over-prediction means that binaries built using static profiling will have fewer NOPs inserted than the binaries created with dynamic profiling. The smaller amount of NOP insertion also means that there is slightly less overall diversification and that small files are therefore susceptible to having a higher percentage of gadgets surviving. Dynamic profiling, on the other hand, uses a training run to predict which code will be expected to be hot in a typical run. If the training run input is similar to the input used in normal circumstances, then the profile can be very accurate. The NOP insertion rate in the hottest areas can be lowered, while still maintaining a higher rate of insertion in the rest of the code. In this way, the diversification level is maintained at a high level while also reducing the execution time. The weakness of dynamic profiling, however, is that it does depend on the training run being representative of a typical full run. If the training run uses a different set of control structures than a normal run, then the wrong portion of the code will be identified as being hot. In that case, optimizations will be made in locations that improve performance very little, while omitting optimizations in areas with high need, and therefore performance will suffer.

A comparison based on file size shows that that binaries created using static profiling tend to be smaller than those created using dynamic profiling. This result supports the concept that static profiling is predicting a larger amount of hot code than the dynamic version. Identifying more code as hot means that those sections will have less new garbage code inserted and the file size will not grow as much.

One unexpected finding was that although static files are smaller, indicating that they have less NOPs inserted, they tend to run slightly slower than the well-trained dynamic files. We believe that this indicates that the static heuristics do over-predict the amount of hot code in a file, reducing the amount of NOP insertion and thus the file size, but that they do not identify most of these sections as belonging to the hottest category. So, for example, in a 10-50 range, the hottest portions of the code might be getting a 20% insertion rate instead the 10% rate that they will get in a well-trained dynamic run. Thus a higher percentage of the code is seen as hot, but less is seen as extremely hot, compared to the well-trained dynamic version.

6. RELATED WORK

Most compilers currently include some type of profiling mechanism. The profiling information is typically used to identify the hot sections of code to enable optimizations to increase program performance or to reduce file size. There are other applications, however, that produce optimizations from the less frequently executed cold code.

Code compression is one example where profiling has been used to identify cold code instead of hot code. Debray and Evans [10] found that compressing cold code sections and then decompressing them when needed resulted in a significant decrease in file size.

Khudia et al. [16] presented another use for cold code detection. Their work involved code duplication for the purpose of detecting transient faults in processors. Original instructions were duplicated and then original and duplicate versions were both executed. The results of the two executions were compared and faults were detected when the results were not the same. They used edge profiling to reduce the number of duplicated instructions. Their method involved adding extra instructions to the code and found that adding it only to the cold code sections produced the best results.

The implemention of software diversity is an effective method to defend against code-reuse attacks, but it is not the only one. There are also other methods that seek to prevent or detect such attacks. Coppens et al. [8] introduced feedback-driven diversification. Their feedback-driven compiler tool flow iteratively transforms code to thwart “exploit Wednesday” attacks.

DROP [6] is an example of detection software that runs at the same time as the program that it is examining. It dynamically instruments the running program to detect when a return is executed. If too many returns are taken within a certain amount of time, it is determined that a ROP attack

is in progress and DROP takes steps to counter the attack.

"Return-less kernels" [20] reorder program instructions and re-allocate registers so that free branch instructions never appear inside a binary. Since ROP attacks rely on these branches, their elimination effectively stops those attacks.

Other methods that use a combined compile-time and runtime approach include G-Free [25], control-flow locking [4], control-flow integrity [1], and software fault isolation (SFI) [21, 32]. The compiler adds code to the program, instrumenting all of the branch instructions and restricting the control flow of the program to only those edges in the original CFG. These methods have proven to be effective against ROP attacks. However, because of their focus on just ROP, some are unable to defend against other types of code-reuse attacks. As such, whoever implements these methods must modify and update them to deal with newer code-reuse attacks, if they can be so modified.

Some of the methods mentioned here incur much larger overheads than our profile-guided NOP insertion. However, many also rely on code insertion, so it is feasible that these other methods could also benefit from profile-guided optimization.

Larsen et al. [18] provided a much more complete overview of the overall field of code diversification in their SoK paper.

7. CONCLUSION

Diversifying binaries through NOP insertion successfully combats code-reuse attacks, but causes slowdowns that may be unacceptable. Static and dynamic profiling methods both enable optimizations that deliver very good performance results in speeding up the execution of diversified files while maintaining the high level of security needed to deter code-reuse attacks. Dynamic profiling is slightly preferred when representative training input is available, but if the training set is bad, there is no training data at all, or file size is a major factor, then dynamic profiling suffers in comparison to static profiling. Static profiling is a good alternative to dynamic profiling in the well-trained scenario, and is significantly better than dynamic in other situations.

The advantages of static profiling that we demonstrate in this paper are not limited to just profile-driven NOP insertion. Profile-driven optimization is sometimes seen as impractical [31], but this is typically because of the difficulties involved with setting up the training run for dynamic profiling or the inability of dynamic profiling to deal with programs with phase shifts. Static profiling can dependably be used instead of dynamic in these cases and in other randomizing transformations, without the up-front costs, while still providing a high quality profile for optimizations.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful suggestions on the previous version of this paper. We also thank Andrei Homescu and Stephen Crane for their assistance and feedback.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, by the National Science Foundation (NSF) under grant No. CCF-1117162, and by a gift from Google.

Any opinions, findings, 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), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

8. REFERENCES

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

[2] Ball, T., and Larus, J. R. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*, 4 (1994), 1319–1360.

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

[4] Bletsch, T., Jiang, X., and Freeh, V. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), ACM, pp. 353–362.

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

[6] Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. DROP: Detecting return-oriented programming malicious code. In *Information Systems Security*. Springer, 2009, pp. 163–177.

[7] Cohen, F. B. Operating system protection through program evolution. *Computers & Security 12*, 6 (1993), 565–584.

[8] Coppens, B., De Sutter, B., and Maebe, J. Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO) 9*, 4 (2013), 24.

[9] Crane, S., Larsen, P., Brunthaler, S., and Franz, M. Booby trapping software. In *Proceedings of the 2013 workshop on New security paradigms workshop* (2013), ACM, pp. 95–106.

[10] Debray, S., and Evans, W. Profile-guided code compression. In *ACM SIGPLAN Notices* (2002), vol. 37, ACM, pp. 95–105.

[11] Evans, C. Exploiting 64-bit linux like a boss, 2013.

[12] Franz, M. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms* (2010), ACM, pp. 7–16.

[13] Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *USENIX Security Symposium* (2012), pp. 475–490.

[14] Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., and Franz, M. Profile-guided automated software diversity. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on* (2013), IEEE, pp. 1–11.

[15] Jackson, T., Homescu, A., Crane, S., Larsen, P., Brunthaler, S., and Franz, M. Diversifying the software stack using randomized NOP insertion. In

Moving Target Defense II. Springer, 2013, pp. 151–173.

[16] Khudia, D. S., Wright, G., and Mahlke, S. Efficient soft error protection for commodity embedded microprocessors using profile information. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 99–108.

[17] Knuth, D. E. An empirical study of fortran programs. *Software: Practice and Experience 1*, 2 (1971), 105–133.

[18] Larsen, P., Homescu, A., Brunthaler, S., and Franz, M. SoK: Automated software diversity.

[19] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on* (2004), IEEE, pp. 75–86.

[20] Li, J., Wang, Z., Jiang, X., Grace, M., and Bahram, S. Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems* (2010), ACM, pp. 195–208.

[21] McCamant, S., and Morrisett, G. Evaluating SFI for a CISC architecture. In *Usenix Security* (2006), p. 15.

[22] Microsoft. Microsoft security intelligence report, vol.16. Tech. rep., 2014.

[23] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e* (2001).

[24] Neustifter, A. Efficient profiling in the llvm compiler infrastructure. Master's thesis, Faculty of Informatics, Vienna University of Technology, 2010.

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

[26] Serna, F. J. The info leak era on software exploitation. *Black Hat USA* (2012).

[27] Shacham, H. 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* (2007), ACM, pp. 552–561.

[28] Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security* (2004), ACM, pp. 298–307.

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

[30] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., and Walter, T. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security* (2009), ACM, pp. 1–8.

[31] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium. USENIX Association* (2014).

[32] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 79–93.