Large-Scale Automated Software Diversity—Program Evolution Redux

Andrei Homescu, Todd Jackson, Stephen Crane, Stefan Brunthaler, Per Larsen, and Michael Franz, *Senior Member, IEEE*

Abstract—The software monoculture favors attackers over defenders, since it makes all target environments appear similar. Code-reuse attacks, for example, rely on target hosts running identical software. Attackers use this assumption to their advantage by automating parts of creating an attack. This article presents large-scale automated software diversification as a means to shore up this vulnerability implied by our software monoculture. Besides describing an industrial-strength implementation of automated software diversity, we introduce methods to objectively measure the effectiveness of diversity in general, and its potential to eliminate code-reuse attacks in particular.

Index Terms—Biologically-inspired defenses, artificial software diversity, return-oriented programming, jump-oriented programming, code reuse attacks

1 Motivation

Time and again, attackers have demonstrated that they hold the high ground over defenders in the perennial struggle of cybersecurity. Current studies [1] estimate that defenders spend at least an order of magnitude more in defense against cyberattacks than attackers profit. Unfortunately, our primary defenses against attackers are mostly *passive*.

We employ monitoring techniques and heuristics to identify malicious behavior, yet, with all the advances in those fields, attackers always develop another 0-day attack to circumvent even the newest defense mechanisms. For example, the Google Chrome browser uses state-of-the-art security measures to defend against exploits, yet new attacks of increasing complexity continue to surface [2].

Given these trends, it is doubly illuminating to re-read Fred Cohen's seminal paper, "Operating System Protection Through Program Evolution" [3] published in 1993. This is an enlightening read and it touches upon many subjects that remain highly relevant even two decades later. It is striking that a full two decades after its first publication, Cohen's assessment of attacker economics and the potential of program evolution still hold. Throughout the paper he asks the research community to conduct further research into this area. Cohen concludes his paper with the following paragraph:

> Clearly, a great deal of further work is required for this field to mature. Specifically, more mathematical analysis of attacks and defenses, a better understanding of what we are trying to conceal and the degree to which evolution is effective at concealing it, and results on the tradeoffs of time and space of techniques are clearly called for.

Unfortunately the research community has not yet fully addressed these concerns. This article re-examines some of Cohen's assumptions, particularly in the light of recent developments [4], and provides updated information on both the practicality and the security implications of large-scale automated software diversity. We present our research on automated software diversification in general, and its effectiveness against code-reuse attacks in particular.

This article makes the following contributions:

- We describe our implementation of compiler-based, automated software diversity. To the best of our knowledge, this is the first industrial-strength implementation of such a system. In particular, we provide details on:
  - the implementation itself (Section 4.2),
  - achieving high-scalability for compiling in the cloud (Section 4.1),
- We present a general approach to measure the efficiency of diversification with respect to concrete security issues (Section 4.3). Specifically, we demonstrate how diversification affects code-reuse attacks, such as jump-oriented and return-oriented programming (ROP).
- We provide results of several detailed analyses of diversified software (Section 5). These results indicate that our approach:
  - has a negligible performance impact (5 percent geometric mean),
  - protects effectively against code-reuse attacks (affecting up to 99.99 percent of gadgets).

- *A. Homescu, S. Crane, S. Brunthaler, P. Larsen, and M. Franz are with the Information and Computer Sciences, University of California, Irvine, CA 92697. E-mail: {ahomescu, sjcrane, s.brunthaler, perl, Franz}@uci.edu.*
- *T. Jackson is with Google.*

Manuscript received 16 Dec. 2013; revised 22 Apr. 2015; accepted 3 May 2015. Date of publication 12 June 2015; date of current version 15 Mar. 2017. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2015.2433252

2 Background: Code-Reuse Attacks

Instead of injecting malicious code into a vulnerable binary, code-reuse attacks inject data (code addresses) into the program during execution to redirect control flow away from regular code paths toward paths chosen by the attacker. The attacker re-uses existing code in creative ways to execute arbitrary attack code.

Return-oriented programming is one of the most popular code-reuse attacks currently in use. This attack was introduced in 2007 by Shacham [5], who described a Turing-complete set of Intel binary code sequences that attackers can use to perform arbitrary computation. Historically, ROP generalized return-to-lib(c) attacks [6] and evolved Krahmer's "borrowed code chunks" technique [7]. Later flavors of ROP use jumps instead of returns for control flow ("Return-Oriented-Programming Without Returns" [8] and JOP—Jump-oriented Programming—[9]).

In a ROP attack, the attacker locates specific code sequences inside the binary (all ending in a return instruction), then places their addresses onto the program stack, using the return instruction itself to transfer control flow from one gadget to the next. Shacham described several such instruction sequences implementing various attack functionality (such as writing to memory) and defined them as *gadgets*. In addition, Shacham also found that any sufficiently large binary contains a Turing-complete set of gadgets [5], allowing the attacker to completely take over a program.

In practice, only a subset of this functionality is needed. ROP is frequently used merely to bypass or disable measures that prevent code injection (such as Data Execution Prevention—DEP), allowing the attacker to perform a classic code injection attack. A Security Intelligence Report released by Microsoft [10] investigates a large number of exploits targeting Microsoft products between 2012 and 2014, showing that the overwhelming majority of exploits that bypass DEP use ROP to do so. Since all current major operating systems implement some form of DEP, ROP is now practically required for any arbitrary code execution attack.

Crucial to all code-reuse attacks is the currently prevalent *software monoculture*: all binaries of a given software and version are identical. A large-scale attack using information from one binary can succeed only when the target binary is identical. Attackers discover the necessary gadget addresses from one binary and can craft a universal attack using these addresses. If, however, all binaries are sufficiently different, a gadget catalog inferred from one binary becomes useless, as gadget addresses in other binaries are different.

3 System Overview: Large-Scale Software Diversity

In 1993, Cohen [3] presented several diversification techniques to achieve security by obscurity, necessary for creating what he called the *ultimate defense*. This is of particular importance, as his motivation was as follows:

> 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. [...], even though it could eventually be successful.

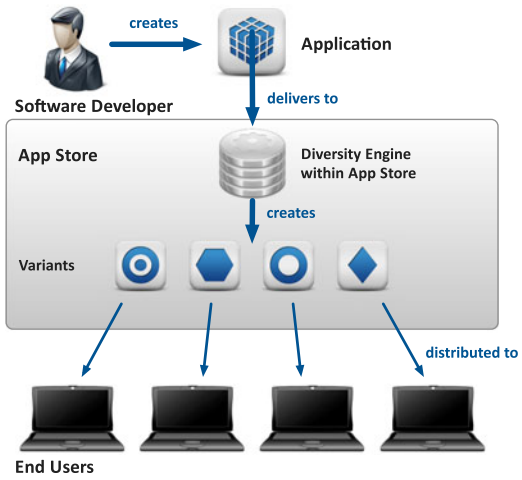


Fig. 1. By using an "App Store" architecture, the diversification process can be made transparent both to developers and end users.

Cohen described the ultimate attack as reverse engineering any defensive measures built into a program, given physical access to the underlying machine. If an attacker can change the machine's data and redirect control flow at will, they will do so in such a way that a program passes all security checks and proceeds as normal. His description of the ultimate attack and ultimate defense still hold two decades after publication, as does his recount of attacker economics. In fact, we share his assessment that diversity makes certain attacks economically unviable and think that we can eliminate code-reuse attacks altogether by using large-scale automated software diversity. Furthermore, Cohen asks other researchers to find answers to some of his questions—which we hope to answer conclusively in this article and to inspire new research in this direction.

In 1997, Forrest et al. [11] describe the first (to the best of our knowledge) practical implementation that uses software diversity as a security defense, implementing or discussing several of Cohen's ideas. Their work focuses on a different kind of attack (stack buffer overflows), but also introduces some other diversifying code transformations (basic block reordering, randomized instruction scheduling) that are now becoming increasingly important, due to the high popularity of code reuse attacks. Their experimental results show that software diversity can be practical and efficient, encouraging all later work on software diversity, including our own.

Transparency is crucial to our software diversification approach (see Fig. 1). In 2010, Franz [4] anticipated lasting changes in the software world that enable us to bring automated software diversity to the people *without* them actually noticing. Franz described a paradigm shift in software delivery: instead of using physical media in shrink-wrapped boxes, the current scenario uses App Stores, i.e., users download their binaries from a central repository on the web. This step would allow developers/distributors to provide different binaries to each end user. Cohen referred to this as program evolution "at the factory," which he concluded has the best security aspects but also takes more effort than simply distributing a

single copy to each user. Since software is moving away from distribution on physical media, this effort is now significantly reduced.

This leaves us with the problem of actually creating these unique binaries for all users. In general, there are two ways of approaching this problem: diversifying an existing binary, or creating a diversified binary from source code. Some implementations use the former approach, which certainly has benefits. For example, this is the only feasible approach when source code is not available. Unfortunately, however, there is also a clear downside to this: disassembling a binary with perfect accuracy is undecidable [3, p. 578]. The practical consequence of this is that not all parts of a binary are eligible for diversification, thereby leaving a door open to attackers. On the other hand, the latter approach—creating a diversified binary from source code—has the exact opposite features: it requires source code, and hence is decidable. In addition, source code provides a plethora of useful information for diversification, such as the layout of program data structures and stack frames, and accurate loop boundaries. Our solution follows this approach and uses a compiler to generate a diversified binary. Since we want to repeatedly diversify program binaries for each end-user we increase the necessary compilation-time with a factor proportional to the number of software downloads.

The recent trend of cloud computing provides the necessary compute-time at a reasonable cost. Unfortunately, compilers are not yet effective at scaling to warehouse-computing. An implementation of compile-time based software diversity must address this problem to effectively manage costs. If the approach requires constant, time-consuming re-compilation from scratch, deployment costs could grow by several orders of magnitude. At this scale, the costs of compilation might outweigh the benefits of a diverse ecosystem.

Compiling and distributing software through cloud-based "App Stores" ensures transparency in one direction: from the developers to the end-users. Usually, the software life-cycle depends on communication in the opposite direction, too. For example, automatic bug and crash reports from users are vital to the detection and resolution of software defects. Franz [4] proposes one possible solution: by adding a random number seed that drives compilation to the program itself, the diversifying compiler can re-create the program and "normalize" the bug reports for the developers. We could also ship customized patches to the end users in the same way. To this end, our system allows transparent operation while simultaneously delivering large-scale automated software diversity—liberating the world from the software monoculture.

4 Design and Implementation

After showing how diversified software can be distributed in practice, we provide the necessary details on how our compiler addresses the security goals laid out in the previous section. We then describe the effectiveness of diversification at thwarting code-reuse attacks.

We implemented our diversifying compiler by extending the LLVM compiler [12] infrastructure (version 2.9). Hence, we enabled automated diversification for all languages supported by the LLVM front-end. Furthermore, ElWazeer et al. [13] and Anand et al. [14] have shown how to decompile large bodies of legacy x86 programs to LLVM's internal representation. In consequence, these recent advances enable us to use our diversifying compiler on binaries that lack source code and debug information, where sufficiently accurate disassembly is possible (while perfect disassembly is undecidable, it is generally feasible to perform a partial disassembly, then complement it with heuristics and expensive run-time checks to compensate for the missing information).

4.1 Scalable Compilation in the Cloud

First, we address the practical question of compiler scalability. This is necessary as this drives our selection of compilation techniques, which we describe afterwards.

The overview section (Section 3) briefly states the importance of scalability for a diversifying compiler. To illustrate why scalability matters, consider a naive approach in which each download requires a full compilation and linking step to create a diversified binary. Since large software packages frequently require hours to compile and link, delivering a million diversified binaries would cost a five-digit dollar amount—even at today's affordable cloud computing rates.

We built a system to compile many variants in parallel on top of Amazon's Elastic Compute Cloud, EC2 for short. Our system implements a classic master-slave work distribution model. The master server keeps track of the diversified binaries using a database, while the slave instances start up automatically to compile new diversified binaries. The master keeps track of the number of downloads for each binary and allows the webserver front-end to serve the undistributed binaries first. In addition, the master also manages the quotas for each specific program and version, as well as a priority parameter to set the number of instances that any queue receives. Our system keeps a repository of undistributed binaries and automatically adds more as needed. We upload the completed binaries to Amazon's Simple Storage Service (S3) using the reduced redundancy tier. This allows us to manage costs efficiently, and does not cause a bottleneck, since we can always generate more program variants on demand. Currently, our implementation already builds and distributes diversified Firefox web browser binaries.

4.1.1 Cost-Effective Provisioning

Cost plays a big role in creating enough compiled versions to meet the demand of a large project. As an experiment, we chose to diversify the Firefox browser because it has a complex build system and is in high demand. For instance, there were over seven million downloads within 24 hours of the release of Firefox 4.

Amazon offers a variety of virtualized servers—"instances" in EC2 parlance—which vary in their computing resources and hourly cost. On-demand instances and reserved instances are billed at a fixed rate. Spot instances, in contrast, are priced based on demand through a bidding process. We report the cost to compile and diversify Firefox using six types of spot instances in Table 1 using current prices as of April 22, 2013. The choice of

TABLE 1
Comparison of Amazon EC2 Spot Instances
(as of April 22, 2013)

Amazon instance type	One build (Hours)	Cost per hour	Cost of 10^6 builds
Small	1.19	$0.007	$8,330.00
Medium	0.62	$0.013	$8,060.00
Large	0.37	$0.026	$9,620.00
X-Large	0.26	$0.052	$13,520.00
High-CPU Medium	0.39	$0.018	$7,020.00
High-CPU X-Large	0.18	$0.070	$12,600.00

instance matters; for example, using High-CPU Medium spot instances cost almost half of using X-Large instances. Since demand for Amazon spot instances and Firefox downloads varies, we (conservatively) provision storage resources for a million Firefox variants as a buffer. Again using current prices, choosing the S3 reduced redundancy tier to buffer 38 TB worth of binaries costs $103 per day.

4.1.2 Optimized Variant Generation

As it turns out, there is substantial optimization potential for compiling in the cloud. For example, our compiler-based diversification process requires no changes to the program source code. Therefore, the repeated parsing and optimization of source code is superfluous. Instead, we parse the source code only once and cache the program's intermediate representation rather than re-creating it for each invocation of the compiler.

A compiler generally consists of separate, decomposed phases. The compiler needs to linearize those phases to build an executable. Consequently, the phases themselves need to be composable. The following two observations allow us to devise a compilation strategy meeting our scalability requirements: first, caching the compilation steps preceding our diversifying transformations eliminates much of the redundant work required by the simple approach; second, performing the diversifying transformations as late as possible minimizes the work for creating each additional binary (see Fig. 2). As a result, we restrict ourselves to performing diversifying transformations only in the compiler's backend.

LLVM intermediate representation—bitcode—is fully documented and easy to serialize. The compiler produces bitcode after parsing the source code and performing some high-level optimizations. Serializing the bitcode eliminates a substantial part of the compute-time needed to diversify each binary. This reduces the time to compile Firefox to a diversified binary by 60 percent. (The compile times reported in Table 1 use this type of bitcode caching.)

Even within the backend, we diversify only in the later stages of compilation after lowering of the bitcode into a machine-specific representation. Therefore, we can reduce the compilation time even further by caching machine-specific intermediate code rather than bitcode. Unfortunately, however, LLVM does not currently support serialization of its machine-specific intermediate code. In consequence, we implemented our diversifying passes as

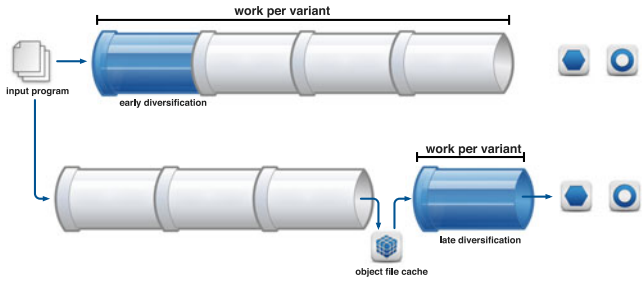


Fig. 2. Caching the redundant compilation steps as bitcode reduces the computational resources required to diversify binaries.

transformations on assembly code using the MAO framework [15]. Our preliminary results indicate time reduction of up to 75 percent relative to naive recompilation.

4.2 Diversifying Compiler Transformations

This section presents a detailed description of the transformations implemented in our compiler. Not only do these transformations displace intended code sequences, they also have the potential to break up unintended code sequences. Furthermore, none of these transformations interfere with program execution and therefore remain compatible with the widest possible range of programming techniques including JIT-compilers, interpreters and operating system kernels. Finally, this list of transformations is not exhaustive and/or exclusive; in fact, several additional transformations are possible, and we anticipate future researchers will discover new diversifying transformations.

There are two main criteria which guide the selection of transformations: entropy and granularity. Picking a high-entropy transformation ensures that the attacker must spend significant resources to undo the randomization by brute force. The low-entropy of ASLR on 32-bit systems has confirmed the importance of this factor: ASLR was brute-forced in 216 seconds [16] on a computer from 2004. However, high entropy is not sufficient. With ASLR, the base address of a randomized library gives the attacker access to all code sequences inside that library (we call this coarse-grained randomization). Therefore, randomization should be done on smaller units of code (finer-grained randomization), so each of those is randomized independently and the attacker is forced to separately discover the location of each code sequence of interest, greatly increasing the costs of the attack.

ASLR performs randomization at library granularity. There are several other levels at which to randomize code: function, loop, basic block and instruction level (Larsen et al. [17] give an overview of these levels and enumerate all current diversity implementations for each level). In this paper, we focus on transformations that operate at the finest granularity—instruction-level diversification. However, other researchers have used coarser-grained transformations (such as basic block reordering) successfully to randomize code and thwart attacks [18], [19]. We believe that these transformations are complementary, and diversification should be applied at every level for maximum security.

4.2.1 Insertion of NOP Instructions

As a first and simple code transformation, we insert NOP instructions between consecutive program instructions.

TABLE 2
NOP Insertion Candidate Instruction Sequences

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

There are two sources of randomness in this transformation: both where to insert and what to insert. We introduce a probability parameter—p_{NOP}—to decide whether to prepend each instruction with a NOP instruction.

Table 2 shows a list of possible NOP instructions. First of all, NOP instruction candidates must preserve the processor state at all times. Second, we chose these instructions carefully to minimize the likelihood of creating new gadgets. In the case of the two-byte instructions, the second byte decodes to an instruction that an attacker is unlikely to use.

Insertion of NOP instructions has desirable performance and security properties. Performance-wise, inserting NOP instructions has minimal impact on both space and time. The added instructions do not interfere with the program's working set besides requiring space in instruction memory. NOP instructions do require hardware resources to fetch and decode, but do not add computational effort to the program's original algorithms; some x86 processors even recognize and optimize the execution of certain NOPs. Security-wise, inserting NOP instructions causes displacement of subsequent program code thereby randomizing gadget locations. Finally, inserting NOP instructions breaks gadgets relying on misaligned instructions.

Fig. 3 illustrates this last effect. Before diversification, an unintentional add-with-carry (ADC) gadget hides in the intentional move and add instructions. Our NOP insertion pass could eliminate the gadget by inserting a NOP instruction between the intended instructions.

4.2.2 Instruction Scheduling

Compilers perform instruction scheduling by rearranging the order of instructions to decrease pipeline hazards. We diversify instruction scheduling by making a random choice instead. Furthermore, we can easily compute a *worst-case* instruction schedule, which helps us to evaluate the worst possible performance impact.

Concerning security, randomizing the instruction schedule has properties similar to the previous technique. If, and only if, the gadget contains all intentional instructions of a schedule, then rearranging will have no effect on the gadget,

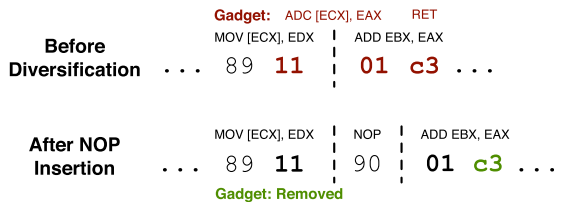


Fig. 3. Eliminating a gadget by inserting a NOP instruction.

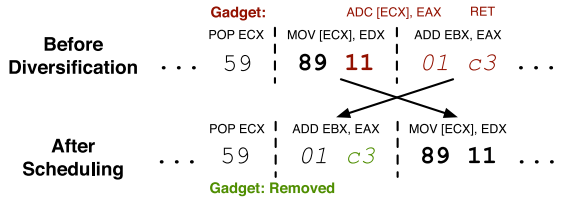


Fig. 4. Removing the ADC gadget by exchanging the MOV and ADD instructions.

since instruction scheduling preserves semantics. Due to the impracticality of using large gadgets, we believe this case is infrequent. In all other cases, including unintentional instructions, changing the order of instructions breaks gadgets.

For example in Fig. 4 we see how randomized instruction scheduling removes the unintended add-with-carry gadget by exchanging the intentionally emitted move and add instructions.

4.3 Measuring Diversification

In our background section (see Section 2), we have described how code-reuse attacks owe their existence to the software monoculture. Attackers can more or less conveniently create gadget catalogs from universally identical binaries and craft eminently successful attacks. Diversification by itself, however, does not remove all vulnerabilities. So, directed attacks where the attacker has access to the binary are still a possibility. Undirected attacks, on the other hand become uneconomical in the presence of diversity.

In general, quantifying diversification is an interesting problem. Traditionally, researchers rely on entropy for this purpose. We think it makes sense to break with this tradition for our specific application of automated software diversity, entropy alone is not sufficient in our scenario. For example, the entropy introduced by NOP insertion is limited not by machine architecture, i.e., word size, but rather by *practicality*: nothing prevents us from inserting millions of NOP instructions until hard disk size becomes a limiting factor. Contrary to intuition, having high entropy in this case does not imply high security. If we insert any number of NOP instructions in front of an intended gadget (resulting in what is frequently referred to as a *NOP sled*), and the attack uses the original address of the gadget, then the NOP sled would not alter attack functionality and the attack would still succeed. Consequently, we propose an additional way of quantifying diversification more precisely. It is worth noting that Pappas et al. [20] appear to use a similar quantitative estimate in their evaluation.

Our strategy to quantify successful diversification involves the concept of *surviving diversification*. Let us take a look at jump-oriented and return-oriented programming to explain this in sufficient detail. We already know that the attacker's gadget catalog is a list of gadget addresses for a specific binary. Subsequently, the attacker uses these addresses to craft an attack. This relies on the fact that all binaries have identical gadgets at identical addresses. A diversified binary, however, contains displaced or altered gadgets. However, a diversified binary is identical to the original from the point-of-view of an attacker if both the gadget functionality and addresses used in an attack are identical in both. Put differently, diversification did not

neutralize a specific gadget at a specific address, or as we say more colloquially, a gadget *survives* diversification.

We scan through the `.text` sections, looking for common instruction sequences—*candidate gadget matches*—ending in free branches such as returns, indirect calls, or jumps. A candidate gadget match is a pair of instruction sequences with identical offsets; one from the original `.text` section 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. We then compensate for the effects of our diversifying transformations in a multi-step normalization process:

1) We remove all potentially inserted NOP instructions from both instruction sequences.
2) We sort the resulting sequences to compensate for instruction scheduling.

Since the sorting step can produce the same result for two semantically different instruction sequences, our algorithm conservatively *overestimates* the actual number of gadgets surviving diversification. Notice that if we want a precise comparison between two instruction sequences, we can capture the semantics of the instruction sequences—for instance using expression trees [21]—and verify whether they have equivalent effects on the processor state. If the normalized instruction sequences are equivalent, the algorithm has identified a potentially surviving gadget.

Automated software diversification only prevents attacks reusing entire functions, such as return-into-lib(c), if the diversifying compiler displaces the functions.[1] We enumerate each function address, including tables for externally visible functions, and compute which identically named functions share the same location in two binaries.

Using this strategy, we determine how many functionally equivalent gadgets and function entry points remain at the same location in a pair of executables. These two properties are a requirement for a single code-reuse attack like ROP and return-into-lib(c) to work on multiple executables without modification. Because we use `.text` section offsets and not absolute addresses, we are able to perform our analysis in an environment where protections such as address space layout randomization (ASLR) [22] do not interfere with results.

5 Evaluation of Compilation Techniques

5.1 Setup

Performance evaluation. We evaluated the performance impact of our transformations using the C benchmarks in SPEC CPU2006 [23]. Our test system was a 2.50 GHz Intel Core 2 Quad Q9300 running Linux with kernel version 2.6.32-38. We ran each executable binary three times and reported the average of those. For each transformation we built 25 different diversified executables to create a better representation of the effect of the transformation, and reported averaged results.

Security evaluation. We evaluated the security impact of our diversifying compiler by examining how it affects gadgets in diversified executables. For this analysis, we considered the C++ tests from SPEC CINT2006 in addition to the C tests of SPEC CPU2006.

We extracted the `.text` sections from executables after diversification and compared them to the `.text` sections in unmodified original executables using our survivor algorithm.

To evaluate the security of our compiler-based automated software diversity technique, we assumed the following:

- The target environment is protected by all currently deployed defenses (such as address space randomization and non-executable data), forcing attackers to resort to code-reuse attacks.
- The attacker cannot create an exact replica of the target environment. Even though attackers may obtain their software from the same source as potential victims, we assume that each of them download and run different binary *variants*.
- Downloading of binaries and uploading of error reports preserves the integrity and confidentiality of the payload.
- Attackers are aware that program binaries are subject to diversification and have access to both program sources and the compiler producing the binaries.
- Attackers do not know the *randomization seed* of a victim's binary variant. Combining the diversifying compiler, the program sources and the seed would let attackers fully replicate the environments they target.

File size evaluation. To examine the effect that our transformations have on executable size, we compiled versions of the executables found in SPEC CINT2006 with several randomization parameters. To account for randomization within the diversifying transformations, we compiled 25 different versions for each combination of randomization parameters.

5.2 Individual Techniques

Performance. NOP insertion with $p_{\mathrm{NOP}} = 1$ (Nop(1.0) in Fig. 5a) showed the largest overhead when used alone. With $p_{\mathrm{NOP}} = 1$, the compiler inserts a NOP instruction for each instruction in the executable, representing the worst case scenario. We report overheads between 1.3 percent for 470.lbm and up to 40 percent for 482.sphinx3. An alternate setting for NOP insertion uses $p_{\mathrm{NOP}} = 0.5$ (Nop(0.5) in Fig. 5a). We measured slowdowns between 1 percent for 458.sjeng and up to 23 percent for 482.sphinx3. The geometric mean performance slowdown over all SPEC benchmarks is 5 percent.

Since the reported 40 percent performance degradation for 482.sphinx3—and 23 percent respectively for $p_{\mathrm{NOP}} = 0.5$—is almost twice of what we measure for the second worst decrease, we investigated why this benchmark is so sensitive to NOP insertion. We performed a similar experiment with a prototype diversifying GCC 4.6.2 and $p_{\mathrm{NOP}} = 1$ and found that it resulted in 20 percent overhead, which

1. Attackers can use the `.plt` section in return-into-lib(c) attacks when the target function is dynamically linked. Our compiler does not currently diversify the `.plt` section. We could do so easily, however, by randomly adding exported "dummy" functions as we compile. The dummy functions displace the addresses of regular function entries in the `.plt` without the need to modify the dynamic linker.

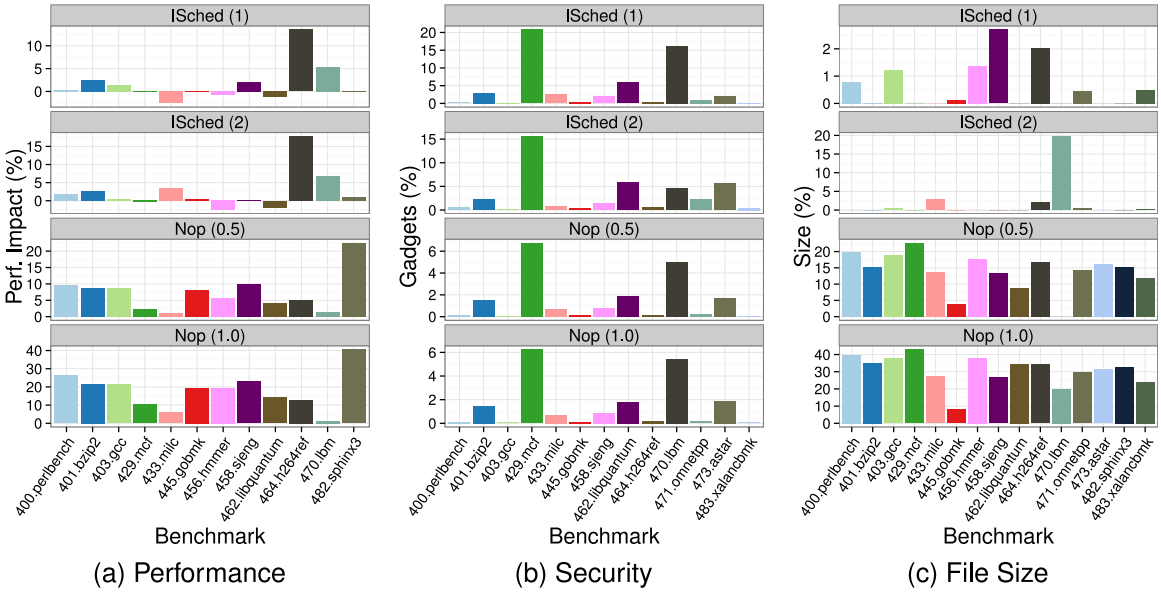


Fig. 5. Individual technique evaluation.

was closer to the results of the other test programs. In 32 bit mode, GCC uses x87 instructions by default whereas LLVM uses scalar SSE2 instructions. While the latter approach is generally most effective, the use of x87 instructions allows GCC to compile a performance critical loop body into a shorter instruction sequence. Since the number of NOPs inserted is proportional to the number of "regular" instructions, GCC's output had fewer NOPs inserted. In the diversified 482.sphinx3 binaries that we studied by hand, we identified a secondary effect contributing to the performance discrepancy: LLVM inserted the same NOP instruction twice in a row. This introduced (artificial) data-hazards (between the two consecutive NOPs) leading to pipeline stalls. We believe this further contributed to the performance disparity.

Besides NOP insertion, instruction scheduling shows the biggest performance impact. Randomized instruction scheduling (ISched(1) in Fig. 5a) and worst-case instruction scheduling (ISched(2)) both show significant effects on the 464.h264ref, with slowdowns of 9 and 20 percent respectively. This is not, however, surprising: the benchmark relies on the instruction scheduler to properly order the instructions of the many calculations in one tight loop. Changing this order results in a suboptimal schedule, causing many more pipeline stalls. On the other hand, we see that benchmarks 433.milc and 456.hmmer show performance improvements. Since these improvements are within the margin of experimental error, we did not investigate this matter further.

Security. Our security evaluation (Fig. 5b) shows the results of surviving gadgets, broken down by diversification technique and probability parameter. Instruction scheduling is the first technique we evaluate. Both randomized and worst-case instruction scheduling remove on average more than 95 percent of gadgets with respect to the undiversified, original binary. 429.mcf displays the worst result for instruction scheduling, having four times more surviving gadgets for randomized, and still three times more for worst-case parameters. We attribute this to the comparatively little importance instruction scheduling plays in this benchmark.

NOP insertion is most successful among all diversifying compiler transformations. With $p_{\mathrm{NOP}} = 1.0$, i.e., inserting a NOP instruction before every regularly emitted instruction, less than 4 percent of gadgets overall and less than 1 percent of gadgets in all but two benchmarks survive. Furthermore, our measurements indicate that diversifying with $p_{\mathrm{NOP}} = 0.5$ gives almost identical results. This is interesting in so far as we report an overall lower performance impact of only 5 percent geometric mean at this setting.

File Size: Unsurprisingly, NOP insertion has the biggest impact on file size At $p_{\mathrm{NOP}} = 0.5$, the range of file-size increase is between 3.9 percent for 445.gobmk and 22.6 percent for 429.mcf. Similarly, at $p_{\mathrm{NOP}} = 1.0$, we see the exact same benchmarks setting the lower limit at about 10 percent and the upper limit at a little bit over 40 percent. This is because most program instructions are longer than NOP instructions on architectures with variable-length instructions (such as x86).

Our results for instruction scheduling are inconclusive. Worst-case instruction scheduling has an outlier with a 15 percent file size increase for 470.lbm. Randomized instruction scheduling has only a negligible effect on file size, too.

5.3 Entropy

Surviving gadgets is one important security metric; entropy is another. When generating a diversified population of binaries, we must make sure that the binaries are sufficiently different that an attacker cannot find some common vulnerability in a large subset of the population. As experience with ASLR has shown, having enough entropy is crucial for a randomization-based defense.

When attacking a binary diversified with NOP insertion using a ROP attack, the attacker will go through each gadget in the attack and try to find or guess its new location. The main variable to look at when analyzing NOP insertion is the displacement of each instruction. This variable is independent for each instruction (and consequently each gadget), which

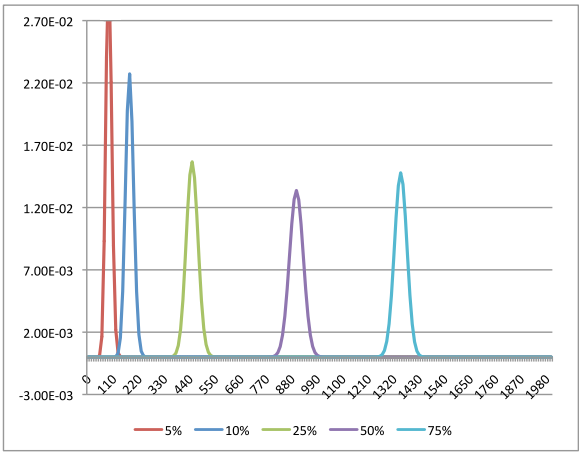


Fig. 6. Distribution of instruction displacement, after inserting NOPs in a program with $N = 1,]000$ instructions, for several p_{NOP} settings.

forces the attacker to spend the effort to find each gadget separately (which means the number of guesses the attacker needs grows exponentially with the number of gadgets; alternatively, the probability of success decreases exponentially).

The displacement of the Nth instruction follows a trinomial instruction, for the following reasons: before that N instruction, the algorithm makes one of the following choices N times: insert a 1-byte NOP with probability $0.2p_{\text{NOP}}$ (since one out of five NOPs has length 1, and they are chosen with equal probability), insert a 2-byte NOP with probability $0.8p_{\text{NOP}}$ (since four out of the five NOPs have length 2) we use or insert nothing with probability $1 - p_{\text{NOP}}$. The total length of the inserted NOPs adds up to the displacement. This displacement is a random discrete variable with values between 0 and $2N$. Fig. 6 shows the distribution of this variable, for $N = 1{,}000$ and several values of p_{NOP}. We observe that the distribution is narrower and taller for very small and very large values of p_{NOP}; once again, the most p_{NOP} setting with the most diversity is $p_{\text{NOP}} = 50\%$. For this value, a large majority of displacements fall between $0.8N$ and N (this should hold true for any value of N). This implies that for NOP insertion entropy grows with number of instructions, and consequently with position of the instruction/gadget; later gadgets have a much wider displacement interval than earlier ones. Past the first few kilobytes of the binary, the entropy of the accumulated displacement should be sufficient to thwart attacks (since there are at least several hundred possibilities of non-negligible likelihood).

Due to the myriad of different choices that the scheduler has during its operation, the entropy of instruction scheduling is much more difficult to measure. We leave this analysis for future work.

6 Evaluation of Diversification Efficiency

Section 5 detailed the results of diversification techniques and surviving gadgets, for individual techniques and their combinations. However, the previous evaluation falls short of diligently measuring diversification itself, since we only compared the surviving gadgets from the original to the diversified binaries. This section presents additional details on measuring diversity.

6.1 Frequently Surviving Gadgets

While it is interesting to analyze diversification techniques for their potential of removing gadgets with respect to the original binary, it is necessary to analyze surviving gadgets among the population of diversified binaries, too. The rationale for this is simple: if there are surviving gadgets common among the population of diversified binaries, they form an attack surface.

Therefore, we built 30 diversified versions of all C and C++ programs in SPEC CPU2006 with varying p_{NOP} values. For each binary we compared the surviving gadgets pairwise among all of those 30 binaries with the same p_{NOP} parameter, and recorded the offset and frequency of matches.

Figs. 7 and 8 display our analysis results for the 433.milc benchmark. We found 433.milc to be representative for other programs of SPEC CPU2006. First, our analysis showed that there is some low-level correlation in the middle of the binaries. This correlation indicates that while using $p_{\text{NOP}} = 0.01$ provides a significant amount of potential diversity, we realize little of it as we insert NOPs with only a small probability. Therefore, we see that gadgets frequently survive at lower offsets. Since NOP insertion displaces gadgets, we see that this correlation tapers off with increasing p_{NOP} probabilities.

Fig. 7 shows two interesting phenomena. First, there are frequently surviving gadgets at the beginning and at the end of the binaries (cf. Figs. 8a and 8b). Upon investigation regarding the surviving gadgets in the beginning and end, we found that this is due to the C runtime setup and finalization procedures. Clang relies on `crti.o` and `crtn.o` files to manage the C runtime and links those files verbatim

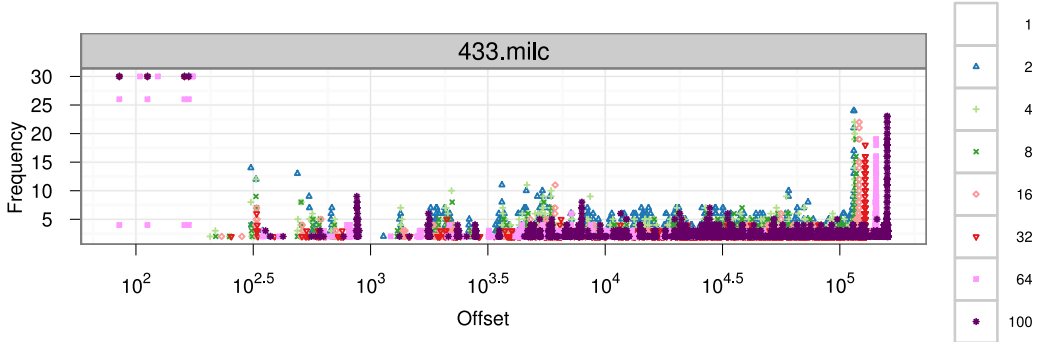
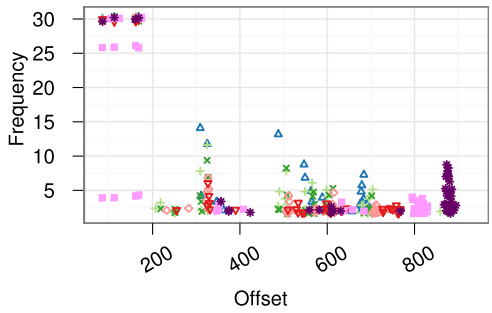
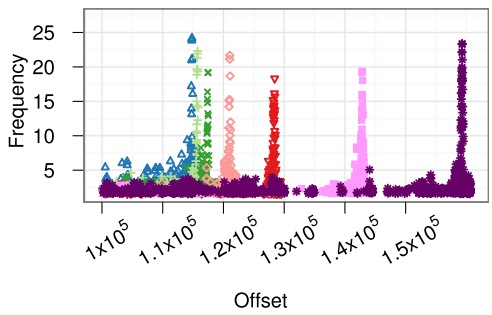


Fig. 7. Overall distribution of gadgets surviving diversification from the `.text` section. Colored shapes indicate p_{NOP} setting in percent.



(a) First 1024 bytes.



(b) Last 50,000 bytes.

Fig. 8. Frequencies of gadgets surviving diversification by location in 433.milc. Colored shapes indicate p_{NOP} setting in percent.

into the executable, *after* compilation. Our implementation of a diversifying compiler did not process those files at all.

Second, diversity at the extremes—i.e., for p_{NOP} at 0.01 and 1—is low. This is to be expected since p_{NOP} at 0.01 is close to zero which creates no diversity. The same holds for $p_{\text{NOP}} = 1$, where we always insert a NOP instruction before *every* instruction in the binary. This is consistent with our expectations from the theoretical analysis in Section 5.3 (the number of different versions follows a similar bimodal distribution to the ones in Fig. 6, centered around a maximum and tapering off for both very low and very high values of p_{NOP}).

6.2 Determining Optimal Parameters

In the previous section, we found that the probability parameter for NOP insertion shows interesting quirks. Inserting with a very low or high probability led to predictable insertions, so we want to insert with some moderate probability. Hence, we determined the optimal parameter for NOP insertion as follows.

We used the largest benchmark of SPEC CPU2006, 483.xalancbmk and built 50 samples for every p_{NOP} value from 0.02 to 1 in 2 percent increments, with a total of 2,500 samples. In addition, we changed compiler settings to account for function alignment (our compiler defaults to a 16-byte alignment, but we also evaluated 1-byte alignment for comparison). For each of the two alignment settings, we used the 2,500 samples per setting to build a smoothed regression curve that approximates the number of surviving gadgets relative to p_{NOP}.

Fig. 9 shows the results of our analysis. Our results indicate that the NOP insertion setting of $p_{\text{NOP}} \approx 0.26$ performs best and loses effectiveness at $p_{\text{NOP}} > 0.4$. Also, function alignment appears to consistently help effectiveness: a 16-byte alignment disrupts more gadgets. We believe that this

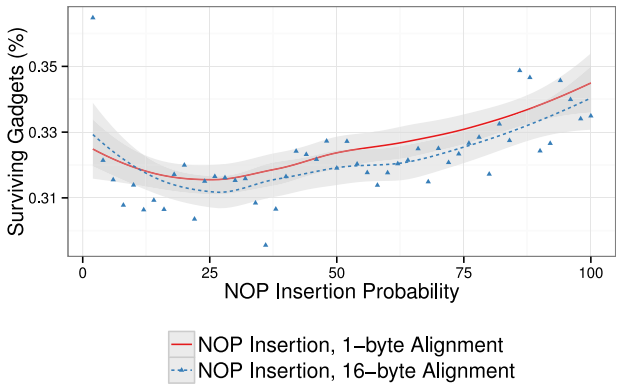


Fig. 9. Plot of NOP insertion effectiveness for different function alignment settings, including smoothed regression curves (the red/blue lines) and confidence intervals (the grey bars).

is due to function alignment causing more code motion—when a function is moved to another 16-byte offset, following gadgets are moved as well. Note that we obtained these results for 483.xalancbmk, results from other programs may vary slightly.

7 Related Work

Since our work addresses large-scale automated software diversity in general, and its application for protecting against code-reuse attacks in particular, we focus our discussion of related work accordingly.

However, we begin by noting that software diversity has several other applications. The earliest uses originate from the fault-tolerance community where multiple implementations by independent teams, possibly using different implementation languages, are used in the hope that they fail independently [24]. Software diversity is also closely related to software obfuscation and watermarking [25], [26] and substantial overlaps exist between the randomizing code transformations employed in these fields. Finally, continuously sending diversified updates to clients in a distributed system can overwhelm tampering motivated by social or financial gains [27].

7.1 Diversity-Based Defenses

Cohen's seminal paper [3] on operating system protection by leveraging program evolution anticipates much of the development in what we now call artificial software diversity. Consequently, it is safe to say that this work motivates subsequent research in automated software diversity in general. Cohen describes several program evolution techniques, which ours derive from with some minor variations and implementation details, for example, he describes adding garbage computation to another program using a source-to-source compiler, which we perform using only NOP instructions instead.

Forrest et al. [11] describe the first practical security-focused implementation of software diversity, demonstrating the practical feasability and performance of this approach. They implement and evaluate one diversification technique that defends against buffer overflow exploits, stack frame randomization, with very small performance and memory overhead. In addition, they discuss

many other options for compiler-level diversification, such as basic block reordering and randomized instruction scheduling (which we also investigate), which have become even more important with the increase in the frequency of code reuse attacks.

7.1.1 Currently Deployed Randomization-Based Techniques

Most mobile and desktop operating systems now employ *address space layout randomization* [22], [28], [29] which provides probabilistic protection against certain kinds of attacks. It does so by randomizing the positions of an application's memory regions (typically including the heap, stack and dynamically loaded libraries).

The diversification techniques presented in this paper are conceptually similar to a fine-grain version of ASLR. With ASLR already deployed, it is relevant to ask if and how our diversification techniques are an improvement. First, we note that the address space for 32-bit processes is so small that the Linux implementation [22] of ASLR only offers 16 bits of entropy, opening the door to brute force attacks [16]. Next, ASLR only applies to applications and libraries compiled to allow loading at arbitrary offsets. Unfortunately, the performance overhead of position-independent code (about 10 percent on average in 32-bit mode [30]) is high enough that executables themselves are often not position-independent. Finally and most importantly, ASLR shifts all objects inside a single memory region around by the same amount. Attackers can then infer the locations of *all* objects as soon as a *single* object inside the region leaks. With fine-grain diversity, such inferences are much more difficult.

7.1.2 Binary-Based Diversification Techniques

In 2012, Pappas et al. [20], Hiser et al. [31] and Wartell et al. [19] presented their approaches to introduce artificial software diversity. In contrast to our approach (which diversifies at compile time), the other approaches add diversity directly to binaries. The advantage of operating on binaries is that there is no need for source code. Unfortunately, however, there is also a major downside: disassembling is undecidable in general.[2]

This has two implications that manifest themselves in the different approaches. Pappas et al. [20] approach this problem by restricting a) the set of instructions eligible for diversification, b) performing only single instruction substitutions on those instructions, and c) relying on unsafe assumptions such as basic block boundaries being correctly discovered [20, p. 6]. The first requirement ensures that all possible jump targets remain untouched, whereas the second rule ensures that no jumps need relocation. This implies that all resulting binaries share a common topology. This topological identity minimizes the available attack surface to those blind spots. In numbers this means that whereas our approach eliminates up to 99.99 percent of all gadgets, this approach eliminates up to 80 percent of all gadgets.

2. We refer the interested reader to Cohen [3], who uses a simple and elegant reduction to the halting problem to prove this.

Hiser et al. [31] solved the disassembly problem by using heuristics-driven disassembling and a virtual machine that has all relevant information to correct mistakes at runtime.

Since the heuristics only minimize the problem of undecidable disassembling, Hiser et al. used a virtual machine—named Strata—to execute diversified binaries. The Strata VM needs to add computation to the existing program, in the form of verifying jump targets and indirect branches. Using this approach, they successfully eliminated over 99.96 percent of all gadgets in their evaluation programs. One problem with using a virtual machine for diversification is that this approach is not viable for certain programs, such as programs using self-modifying code—including JIT compilers and program obfuscators. Finally, the virtual machine itself becomes part of the attack surface.

Wartell et al. [19] took yet another approach to work around the undecidability of disassembly: "binary stirring". Since code and data cannot be fully separated, stirring duplicates program code such that one copy is treated as code and another one as non-executable data. The executable copy is diversified at load time by disassembling the code and instrumenting computed jumps and memory loads, while potential jump targets in the data copy are redirected to the proper code targets. The advantage of such load time diversification is that all clients download the same binary, while still breaking the software monoculture. Similar to Pappas et al., the approach is unsound in that it depends critically on being able to correctly identify features of the binary code such as all possible jump targets. Consequently, the rewritten programs may crash when the heuristics fail. From a security perspective, this approach is comparable to our own, successfully eliminating over 99.99 percent of gadgets.

Many compile- and load-time diversification techniques leave diversification of dynamically-generated code (e.g., code generated by just-in-time compilers—JITs) as future work. Homescu et al. [32] described an implementation of run-time diversity targeting JITs (and dynamic code generators in general)—librando. They applied the binary stirring approach to code emitted at run-time, mainly targeting JIT compilers for dynamically-typed languages (like JavaScript), in addition to Java compilers. librando is a complementary solution to static diversification techniques, covering the latter's weakness to dynamic code generation.

We expect that all techniques protect against ROP and JOP, with minor disadvantages for Pappas et al. [20] due to its blind spots. Hiser et al. [31], Wartell et al. [19], and our system protect against certain forms of return-into-lib(c) attacks, too. Our evaluation shows that using full-system scale diversification is an effective tool for protecting against return-into-lib(c) attacks. At this scale, hosting the whole operating system in a virtual machine is probably not viable.

7.1.3 Compiler-Based Diversification Techniques

In 2008, Jacob et al. [33] introduced the idea of a "superdiversifier," a compiler that performs superoptimization [34], [35] for the purposes of increasing computer security. They focused on the potential for security, but do not evaluate the performance/security tradeoff in a real-world environment.

We previously described our experiments with NOP insertion [36]. Our earlier experiments focused solely on NOPs and

showed the significant security benefits of these techniques on real-world applications (such as Apache and Chromium). Besides the evaluation of multiple diversification strategies, we also addressed scalability concerns and measured the security impact in context of return-into-lib(c) attacks in this work. Larsen et al. [37] addressed the schism between compilation and binary rewriting by combining our compiler-based approach with a static binary rewriting framework. This enables diversification of source code as well as legacy and proprietary binaries in a single framework.

Giuffrida et al. [18] used a diversifying compilation scheme to protect operating systems from kernel level exploits by transforming both the code and data layout of processes. Their code transformations included function shuffling and basic-block reordering inside functions. Their approach collects meta-data during compilation to optionally allow live re-randomization of kernel components while the remainder of the operating system keeps running. Similar to our system, the authors made use of the LLVM compiler and reported similar performance overheads from ahead-of-time diversification of SPEC 2006 (4.8 percent).

Homescu et al. [38] extend our present approach to compile-time diversity and show that the cost of software diversification becomes insignificantly small when combined with profile-guidance. The latter is a well-known compiler technique to focus optimization efforts on frequently executed code. Unlike optimization, diversity affects performance negatively, so Homescu et al. focus diversification efforts on *in*frequently executed code. Profile-guided diversity was only evaluated for NOP insertion but we believe it applies to all the techniques we presented here.

7.1.4 Hiding Randomized Code

Code randomization defenses rely on the memory secrecy assumption, i.e., that the attacker cannot read code after randomization. However, attacks such as JIT-ROP [39] violate this assumption by reading code during program execution, after it has been randomized. JIT-ROP targets programs that allow the attacker to execute arbitrary sandboxed code written in some high-level language (such as JavaScript or Python). Browsers such as Firefox and Chrome that execute malicious JavaScript are prime examples of applications exposed to JIT-ROP. Using a JavaScript memory leak, the attacker reads program code at run-time and relocates the gadget payload to match the diversified code layout. This attack successfully defeats software diversity that perform one-time randomization, such as during compilation or program load time. For this reason, code hiding has become a required component of any practical implementation of software diversity.

Several possible defenses have been proposed: runtime re-randomization [18], control-flow randomization [40], and execute-only memory. Execute-only memory implementations for x86 include XnR [41], HideM [42], and Readactor [43]. Execute-only memory, an idea that traces back to Multics, prevents read acccesses to code pages and therefore JIT-ROP. The most comprehensive approach—Readactor—also prevents code pointers in readable memory from indirectly leaking information about the code layout.

7.2 Non-Diversified Defenses

Szekeres et al. [44] survey non-diversifying protections against exploitable memory errors inherent to systems programming languages.

Onarlioglu et al. present a set of techniques that "de-generalize the [ROP] threat to a traditional return-into-lib(c) attack." [45] Their technique allows for comprehensive protection against (jump-) and return-oriented programming attacks at the expense of adding run-time checks to the secured programs. While this technique also relies on the insertion of NOP instructions, the purposes differ: We insert NOPs to diversify whereas Onarlioglu et al. use NOP-sleds to enforce aligned execution of instructions critical to their protection mechanism. They report an average overhead of about 3 percent on the presented programs (however, they used different benchmarks, so their results are not directly comparable to ours).

The main alternative to diversity against code reuse attacks is to restrict program control flow to intended code paths (or to a secured region of program code), an approach known as control-flow integrity (CFI) [46], [47]; control flow locking [48] and software-fault isolation (SFI) [49], [50], [51] operate in a similar way. In contrast to our approach of reducing predictability of a program, these implementations keep the program predictable but impose significant restrictions in its behavior. For example, CFI restricts control flow paths exclusively to allowed intended targets, preventing the attacks from exploiting unintended paths. While these techniques are very powerful against code reuse attacks, they still leave the program open to vulnerabilities in the defenses themselves. If the attacker discovers a vulnerability that bypasses SFI/CFI protection, they can use this vulnerability on all programs protected by these techniques. In fact, recent research [52], [53] shows that certain implementations of CFI can be easily bypassed by a ROP attack, using only gadgets that branch to targets allowed by CFI.[3] In contrast to the CFI-based approaches, discovering a vulnerability in a diversified program would allow the attacker to target only a subset of the diversified population. Additionally, our diversity-based approaches tackle a strictly greater set of attacks, such as return-into-lib(c), and micro-architectural attacks [54].

Another approach to defending against ROP attacks is detection. Implementations of this approach (such as DROP [55]) attempts to detect when the processor executes many return instructions in a short period of time, which is a common marker of ROP attacks. Later implementations take advantage of hardware support for this approach, in the form of a history of the most recently taken branches provided by the processor (called the Last Branch Record—LBR). Two anti-ROP defenses (kBouncer [56] and ROPecker [57]) use this information to detect ROP attacks, by checking for gadgets pointed by the entries in the LBR. However, novel attacks [58], [59] defeat this approach by overwriting the entire LBR contents with values that pass the security checks.

3. CFI presents a trade-off between security and performance: its strictest versions also have a significant performance overhead (Abadi et al. [47] report an average overhead of 21 percent); faster implementations of CFI relax some of the restrictions on control flow to gain some performance (so-called coarse-grained CFI), which renders them vulnerable to anti-CFI ROP attacks [52], [53].

8 Conclusion

The software monoculture benefits attackers by simultaneously widening the applicability of any single attack vector as well as enabling attackers to re-create a target environment while developing an exploit. We investigated the effectiveness of introducing artificial software diversity at compile-time. The code diversification methods we have developed focus on thwarting return-oriented and jump-oriented programming. These attacks have proven to be surprisingly difficult to protect against, and many existing defenses either restrict program generality, hinder performance, or both.

We evaluated the performance implications of using diversified software and reported only a minor run-time overhead of 5 percent (geometric mean with $p_{\mathrm{NOP}} = 0.5$). This result applies to CPU-intensive standard benchmarks and we anticipate that real-world applications will show significantly less overhead. In addition to our performance results, we analyzed the security implications of diversified software, using surviving gadgets as a metric. Our results indicate that a combination of our techniques shifts or removes up to 99.99 percent of potentially usable gadgets. The probability of a required set of gadgets chosen by the attacker to be present in predictable locations in any diversified binary therefore becomes vanishingly small.

Summing up, we believe that our techniques demonstrate the power of introducing artificial software diversity at compile-time, and we think that our results conclusively make the case for further investigation.

Acknowledgments

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] R. Anderson, C. Barton, R. Böhme, R. Clayton, M. van Eeten, M. Levi, T. Moore, and S. Savage, "Measuring the cost of cybercrime," in *Proc. 11th Annu. Workshop Econ. Inf. Security*, 2012.

[2] J. L. Obes, and J. Schuh. (2012). A tale of two Pwnies, Part 1, The Chromium Blog [Online]. Available: http://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html

[3] F. Cohen, "Operating system protection through program evolution," *Comput. Security*, vol. 12, no. 6, pp. 565–584, Oct. 1993.

[4] M. Franz, "E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism," in *Proc. Workshop New Security Paradigms*, 2010, pp. 7–16.

[5] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Security*, 2007, pp. 552–561.

[6] Nergal. (2001). The advanced return-into-lib(c) exploits: PaX case study. *Phrack Mag.* [Online]. *11(58).* Available: http://www.phrack.org/issues.html?issue=58&id=4

[7] S. Krahmer. (2005). x86-64 buffer overflow exploits and the borrowed code chunks exploitation techniques [Online]. Available: http://www.suse.de/~krahmer/no-nx.pdf

[8] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proc. 17th ACM Conf. Comput. Commun. Security*, 2010, pp. 559–72.

[9] T. Bletsch, X. Jiang, V. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proc. 6th ACM Symp. Inf., Comput. Commun. Security*, 2011, pp. 30–40.

[10] Microsoft Corp., "Microsoft security intelligence report," vol. 16, Jul.–Dec. 2013.

[11] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proc. Workshop Hot Topics Operating Syst.*, 1997, pp. 67–72.

[12] C. Lattner and V. Adve. (2004). LLVM: A compilation framework for lifelong program analysis & transformation, in *Proc. 2nd IEEE/ACM Int. Symp. Code Generation Optimization*, pp. 75–87 [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[13] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," in *Proc. 34th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2013, pp. 51–60.

[14] K. Anand, M. Smithson, K. ElWazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proc. 8th EuroSys Conf.*, Prague, Czech Republic, 2013 pp. 295–308.

[15] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani, "Mao—An extensible micro-architectural optimizer," in *Proc. 9th Annu. IEEE/ACM Int. Symp. Code Generation Optimization*, 2011, pp. 1–10.

[16] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proc. 11th ACM Conf. Comput. Commun. Security*, 2004, pp. 298–307.

[17] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proc. 35th IEEE Symp. Security Privacy*, 2011, pp. 1–10.

[18] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proc. 21st USENIX Security Symp.*, 2012, pp. 475–490.

[19] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. 19th ACM Conf. Comput. Commun. Security*, 2012, pp. 157–168.

[20] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proc. 33rd IEEE Symp. Security Privacy*, 2012, pp. 601–615.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *Proc. 20th USENIX Security Symp.*, 2011, p. 25.

[22] *Homepage of The PaX Team*, PaX. (2009) [Online]. Available: http://pax.grsecurity.net.

[23] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[24] L. Chen and A. Avizienis, "N-version programminc: A fault-tolerance approach to rellablllty of software operatlon," in *Proc. 25th Int. Symp. Fault-Tolerant Comput. Highlights Twenty-Five Years'*, Jun. 1995, p. 113.

[25] C. S. Collberg, C. D. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proc. 25th ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, San Diego, CA, USA, Jan. 19–21, 1998 pp. 184–196.

[26] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Reading, MA, USA: Addison-Wesley, 2009.

[27] C. S. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proc. 28th Annu. Comput. Security Appl. Conf.*, Orlando, FL, USA, Dec. 3–7, 2012, 2012, pp. 319–328.

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

[29] S. Bhatkar, R. Sekar, and D. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proc. 14th USENIX Security Symp.*, 2005, pp. 271–286.

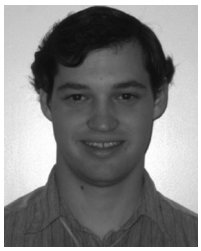
[30] M. Payer. (2012). Too much PIE is bad for performance, ETH Zurich, Tech. Rep. [Online]. Available: http://nebelwelt.net/research/publications/tr-pie12/

[31] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proc.e 33rd IEEE Symp. Security Privacy*, 2012, pp. 571–585.

[32] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: Transparent code randomization for just-in-time compilers," in *Proc. 20th ACM Conf. Comput. Commun. Security*, 2013, pp. 993–1004.
[33] M. Jacob, M. Jakubowski, P. Naldurg, C. Saw, and R. Venkatesan, "The superdiversifier: Peephole individualization for software protection," in *Proc. 3rd Int. Workshop Adv. Inf. Comput. Security*, 2008, pp. 100–120.
[34] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," in *Proc. 12th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2006, pp. 394–403.
[35] H. Massalin, "Superoptimizer: A look at the smallest program," in *Proc. 2nd Int. Conf. Archit. Support Program. Languages Operating Syst.*, 1987, pp. 122–126.
[36] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Diversifying the software stack using randomized NOP insertion," in *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*, series Advances in Information Security, S. Jajodia, A. K. Ghosh, V. Subramanian, V. Swarup, C. Wang, and X. S. Wang, Eds. New York, NY, USA: Springer, 2012, vol. 100, pp. 151–174.
[37] P. Larsen, S. Brunthaler, and M. Franz, "Security through diversity: Are we there yet?" *IEEE Security Privacy*, vol. 12, no. 2, pp. 28–35, Mar./Apr. 2014.
[38] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proc. Int. Symp. Code Generation Optimization*, 2013, pp. 1–11.
[39] K. Z. Snow, F. Monrose, L. V. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. 34th IEEE Symp. Security Privacy*, 2013, pp. 574–588.
[40] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2015.
[41] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proc. 21st ACM Conf. Comput. Commun. Security*, 2014, pp. 1342–1353.
[42] J. Gionta, W. Enck, and P. Ning, "HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proc. 5th ACM Conf. Data Appl. Security Privacy*, 2015, pp. 1342–1353.
[43] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proc. 36th IEEE Symp. Security Privacy*, 2015.
[44] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. 34th IEEE Symp. Security Privacy*, 2013, pp. 48–62.
[45] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, " G-Free: Defeating return-oriented programming through gadget-less binaries," in *Proc. 26th Annu. Comput. Security Appl. Conf.*, 2010, pp. 49–58.
[46] M. Abadi, M. Budiu, Ú. Erlingsson and J. Ligatti, "Control-flow integrity," in *Proc. 12th ACM Conf. Comput. Commun. Security*, 2005, pp. 340–353.
[47] M. Abadi, M. Budiu, Ú. Erlingsson and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Security*, vol. 13, pp. 4:1–4:40, 2009.
[48] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. 27th Annu. Comput. Security Appl. Conf.*, 2011, pp. 353–362.
[49] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. 14th ACM Symp. Operating Syst. Principles*, 1993, pp. 203–216.
[50] S. McCamant and G. Morrisett. (2006). Evaluating SFI for a CISC architecture in *Proc. 15th USENIX Security Symp.*, pp. 209–224 [Online]. Available: http://portal.acm.org/citation.cfm?id=1267336.1267351
[51] 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 *Proc. 30th IEEE Symp. Security Privacy*, 2009, pp. 79–93.
[52] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proc. 35th IEEE Symp. Security Privacy*, 2014, pp. 575–589.
[53] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 401–416.
[54] C. Percival, "Cache missing for fun and profit," in *Proc. Tech. BSD Conf.*, 2005.
[55] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *Proc. 5th Int. Conf. Inf. Syst. Security*, 2009, pp. 163–177.
[56] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *Proc. 22nd USENIX Security Symp.*, 2013, pp. 447–462.
[57] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2014.
[58] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 385–399.
[59] E. Goktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 417–432.

Andrei Homescu is currently working toward the PhD degree at the University of California Irvine, having previously received the Computer Engineering diploma from Politehnica University of Bucharest. His research interests include compilers, virtual machines, just-in-time compilers, and systems security.

Todd Jackson received the PhD degree in computer science degree from the University of California, Irvine, and is currently employed by Google.



Stephen Crane received the BS degree in computer science from California State Polytechnic University, Pomona, and is currently working toward PhD degree at the University of California, Irvine. His research interests include systems security and cryptography.

Stefan Brunthaler received the Dr Techn degree from the Vienna University of Technology. He is a postdoctoral scholar in the Department of Computer Science, University of California, Irvine. His research interests include compilation, interpretation, analysis, optimization, and verification.

Per Larsen received the PhD degree from the Technical University of Denmark. He is a postdoctoral scholar in the Department of Computer Science, University of California, Irvine. His research interests include security, compilers, code profiling, and optimization.

Michael Franz received the Doctor of Technical Sciences and Diplom-Ingenieur degrees from ETH Zurich. He is a professor of computer science in the Donald Bren School of Information and Computer Sciences (and, by courtesy, professor of EECS in the Henry Samueli School of Engineering) at the University of California, Irvine. He is a distinguished member of the ACM and a senior member of the IEEE.