Security through Diversity:
Are We There Yet?

Per Larsen, Stefan Brunthaler, and Michael Franz | University of California, Irvine

Because most software attacks rely on predictable behavior on the target platform, mass distribution of identical software facilitates mass exploitation. Countermeasures include moving-target defenses in general and biologically inspired artificial software diversity in particular. Massive-scale software diversity has become practical due to the Internet (enabling distribution of individualized software) and cloud computing (enabling the computational power to perform diversification).

Hardly a day passes without a report of a major software vulnerability, often accompanied by the uncomfortable information that the vulnerability is being exploited in the wild. Almost all these vulnerabilities result from human error as well as developers programming in languages that have neither strong type systems nor automatic memory management, but that have a performance edge. In a perfect world, software development would occur in such a way that no exploitable errors are created. For the time being, technology that reduces software vulnerability will only have a chance in the marketplace if it has a small performance impact.

The principle that a moving target is harder to hit applies in not only conventional warfare but also in cybersecurity. Moving-target defenses change a system's attack surface with respect to time, space, or both. For instance, software diversity makes the software running on each individual system unique—and different from that of the attacker. Diversity can have a potentially large impact on security with little impact on runtime performance. That's not to say that software diversity is free or trivially easy to deploy, but it can be engineered to minimize impact on both developers and users. In addition, diversification costs can be placed up front (prior to execution) so there's no ongoing drag on performance.

In this article, we survey the current state of software diversity research and look at two fundamental approaches to software diversity—compile-time diversification and binary rewriting. We also describe an evolved approach that builds on these approaches' strengths while minimizing their weaknesses.

Real-World Solutions: Speed and Compatibility

When academics invent new software protection techniques, they're free to assume that programs can be rebuilt from sources, rewritten in new languages, or executed on postulated hardware platforms. Unfortunately, the real world has legacy hardware platforms, large bodies of legacy software, and proprietary software that can't be produced from source code—either because the source code is lost or the original production environment (compiler, linker, and so forth) no longer exists. And above all, runtime performance is important in the real world—protection schemes that cause software to operate 10 times slower have no chance of deployment.

 Copublished by the IEEE Computer and Reliability Societies

Techniques adopted by industry are typically performance neutral because systems are power constrained; performance impact translates to shorter battery life on mobile devices and higher costs to operate datacenters.

Another important requirement for adoption is near universal compatibility with existing code. Protections that require extensive changes at the source level simply aren't economical. In contrast, the best solutions—for example, cryptography—are inexpensive and transparent to developers and users alike. $W \oplus X$ protection against code injection attacks is both cheap and transparent due to hardware support. Similarly, stack canaries that stop stack-smashing attacks impact program performance by less than 1 percent on average and 5 percent at worst.

A poor man's diversity of sorts is already deployed in the form of address space layout randomization (ASLR). Unfortunately, position-independent code increases register pressure on x86 processors executing 32-bit code and thus degrades performance. Furthermore, ASLR is highly susceptible to information leakage attacks; because all addresses in a segment are shifted by a constant amount, a single leaked code pointer lets attackers sidestep this defense.

The point in the development pipeline at which diversity is introduced matters for several reasons. Because software is predominantly distributed in binary form, diversification during compilation means that it occurs before software distribution and installation on end-user systems. So, software developers or distributors must pay for the computational resources necessary for diversification. Postponing diversification until the time at which the binaries are installed or updated on the end-user system distributes the diversification cost among users instead. However, post facto diversification via binary rewriting interferes with code signing because it changes the cryptographic hash. Signed code is used pervasively on mobile devices and increasingly on desktops. Finally, not all applications of diversity are possible with host-based solutions. Diversification makes tampering and piracy significantly harder[1] and protects software updates against reverse engineering[2]; these protections are ineffective if diversification is host based—users can simply disable the diversification engine running on their systems.

The question, then, is this: Can software diversity meet the above constraints?

Fundamental Approaches

We've hinted at the tension between compilers and binary rewriters as delivery vehicles for diversity. It's instructive to revisit arguments made in favor of binary rewriting approaches. Daniel Williams and his colleagues are concerned that "applying a transformation … at compile time, although simple to do, creates another problem: it produces multiple binaries, creating both manufacturing and distribution problems."[3] Sandeep Bhatkar and his colleagues[4] and Richard Wartell and his colleagues[5] similarly highlight binary rewriting's compatibility with current distribution mechanisms. Jason Hiser and his colleagues add to these concerns while summarizing their work: "In short, ILR [instruction layout randomization] operates on arbitrary executables, requires no compiler support, and no user interaction."[6] Wartell and his colleagues[5] and Vasilis Pappas and his colleagues[7] highlight binary rewriting's compiler-agnostic nature and ability to work without program sources and debugging information.

Although we mostly agree with these researchers and think binary rewriting is an invaluable tool, it's no silver bullet. Compilation of source code to machine code is an inherently lossy transformation. Specifically, the von Neumann computer architecture makes distinguishing code from data difficult. In addition, indirect and external control transfer targets aren't fully recoverable. As a result, binary rewriters require various correctness-preserving strategies to compensate for the information lost in translation.

Compile-time diversity approaches are all similar to one another in the sense that they add another pass to the translation sequence, the key difference being the transformation types implemented. Binary-rewriting approaches show much greater variety, particularly with respect to the time at which rewriting occurs. Note that both approaches support dynamic randomization (that is, diversification during execution) in addition to static diversification ahead of time.[6,8]

Pappas and his colleagues present a fully static approach that rewrites binaries in situ.[7] Because the diversified binary's code layout is isomorphic with respect to the original code layout, indirect control transfers aren't a problem, and the performance impact is modest. Unfortunately, code snippets remain untouched and thus unprotected when they can't be decompiled.

Hiser and his colleagues' ILR approach couples static analysis with dynamic binary rewriting in a process virtual machine.[6] The instructions' virtual addresses are randomized in an offline step, and a fall-through map reassembles and caches code fragments during execution, diversifying all the application code with 13 percent performance overhead. Wartell and his colleagues' rewriter defers the layout randomization until load time.[5] Again, an offline step rewrites the binary to "stir" it on execution. Because of the coarser diversification granularity, the runtime overhead drops to 5 percent. This approach requires twice the process memory normally used for code whereas its runtime overhead is on par with basic compiler-based diversity.[8]

Evolved Approaches

In our view, neither solution strictly dominates the other; a comprehensive diversification approach incorporates aspects of both.

Runtime Performance

The popularity of mobile devices, laptops, and cloud computing means that energy efficiency matters. Using more memory means higher power consumption; the same is true of less efficient code.

Diversifying a code fragment tends to make it less efficient. The impact on execution time depends on how frequently the fragment is executed. Because "hot" code fragments have more impact on the aggregate slowdown from diversification, we can vary the amount of diversification in proportion to execution frequency. Instead of leaving the most frequently executed code fragments untouched by diversification, we can employ instruction-level transformations that add a configurable amount of diversity to them. In addition, we can displace all program addresses without touching the "hottest" code because every change affects code at every subsequent address.

This is one area in which compiler-based diversity shows its strength. Every mature compiler includes the machinery to instrument and profile programs. Even if developers don't take the time to profile their programs, we can still vary the amount of diversity to lower performance impact. We can simply use a fallback mechanism that statically predicts execution frequencies on the basis of the code's structure.

Our recent work on a diversifying compiler, or *multicompiler*, is an instantiation of this idea.[9] It uses no-operation instructions (NOPs) to randomize the code layout based on observed execution profiles. NOP insertion and other fine-grained transformations are powerful counters to modern exploits that reuse tiny code fragments called *gadgets*. Unlike coarse-grained randomization, these transformations can displace gadgets and mutate their behavior. By varying the amount of diversity, the performance overhead drops from 5 to approximately 1 percent—in line with already deployed security techniques—while offering essentially the same security. Such profile-guided diversity typically leaves between 1 and 30 percent extra gadgets available to attackers relative to uniform diversification. To put these numbers into perspective, uniform diversification removes up to 99.95 percent of original gadgets, hence even a 30 percent increase isn't all that helpful to attackers. In fact, our experiments with two published ROP attack-generation tools showed that attacks created for the original binary didn't succeed in compromising a program diversified with or without profile guidance.

Coverage

Ahead-of-time compilers can't diversify all kinds of code. Two blind spots are particularly relevant: legacy code and dynamically generated code.

We can't recompile legacy code because the sources and tool chain are no longer available or because the code is closed source and isn't diversified by the vendor. Thus, we can't apply compile-time diversification. However, Kapil Anand and his colleagues' recent work on decompilation delivers the missing piece.[10] Instead of directly targeting a programming language, their system—called SecondWrite—targets the low-level virtual machine (LLVM) compiler's intermediate representation (IR). This has multiple advantages including the ability to leverage the extensive program analysis, optimization, and code-generation functionality already present in a mature compiler. In addition, a compiler IR traditionally is a lower-level representation than source code and thus closer to the binary program. We're currently integrating SecondWrite with our multicompiler; we expect that diversified binaries produced this way will perform on par with existing binary rewriters.

This unified approach has several advantages. When input programs are binaries, the output is no slower than using other rewriters. When programs are diversified from source, the resulting binary runs at close to full speed thanks to profile guidance. Improvements to our diversifying compiler apply directly to both input types.

The popularity of just-in-time (JIT) compilation also presents a challenge. Many newer languages have features that ahead-of-time compilers can't effectively optimize. Deferring generation of machine code until a code fragment is known to be hot often produces a better result. Unfortunately, with the tools we discussed, there's no way to diversify such dynamically generated code. In theory, all JIT compiler developers could implement their own diversifying transformations, and in fact, some already do. However, this is impractical for several reasons. First, the sheer number of internally different JIT compiler engines forces developers to repeatedly implement the same transformations. Second, it leaves legacy JITs without protection.

Our solution is to implement a black-box diversifier in the form of an external library called `librando`. Our library is essentially a dynamic binary-rewriting system.[11] It has the dual benefits of protecting all JITs on a given system without modification and placing all diversifying transformations within a common framework, resulting in fast and comprehensive deployment of new diversification techniques in response to emerging threats.

To support the widest range of JIT compilation engines, `librando` works in a fully transparent manner. Like Wartell and his colleagues, we keep the original code emitted by the JIT compiler in memory and mark it as

nonexecutable. Detecting when and where the JIT compiler generates code is relatively easy on modern systems. Because of the $W \oplus X$ page-protection mechanism, the JIT compiler must use special APIs to allocate executable memory; we intercept these calls and ignore the request to mark the pages as executable. When the control flow reaches the original code, a signal handler is triggered. Conceptually, `librando` handles this signal; disassembles the newly generated code; emits diversified code to a separate, executable location; and transfers control there.

Dynamic code-rewriting systems must be invisible to their host application. However, most important, fragments of diversified code reside at different virtual addresses from their unmodified equivalents. Using the usual machine instructions for function calls and returns means that the return addresses pushed onto the machine stack would point to diversified code. Because the JIT might parse the stack and even make arbitrary modifications to it, we keep unmodified return addresses on the stack and translate them to their diversified equivalents in the epilogue of diversified functions. This translation step means we execute more instructions per function than without diversification and we interfere with the processor's branch predictor.

With the current version of `librando`, performance overheads for highly dynamic code such as the V8 JavaScript benchmarks were a factor of 3.5 on average. For the Java Virtual Machine (HotSpot), the overhead factor was substantially lower—1.08 times on average. However, personal experience using `librando` daily indicates that the user experience isn't affected when browsing JavaScript-intensive webpages or running Java applications. Nevertheless, we're exploring ways to lower overhead. For instance, we expect that using an inline cache to store the diversified return address used the last time the function returned will dramatically decrease the performance impact on JavaScript code.

Combining the multicompiler, the SecondWrite IR-level decompiler, and `librando` for JIT compilers presents a unified, comprehensive approach to software diversity. Most if not all code can be protected using this method. The way the code is generated affects the resulting runtime overhead but not its security. This cements the position of diversity as a universally compatible and versatile defensive strategy, as opposed to the more narrowly focused solutions in use today.

Distribution

Modern software distribution systems are complex; to simplify the discussion, we fix a few of the variables. Proprietary software and signed binaries are more challenging than open source programs and unsigned binaries; we concentrate on the former cases. Similarly, software is increasingly sold and distributed via the Internet for both mobile and conventional computer systems, so we won't consider distribution via physical media. A concrete scenario that meets these criteria is software delivered via online marketplaces or "App Stores."

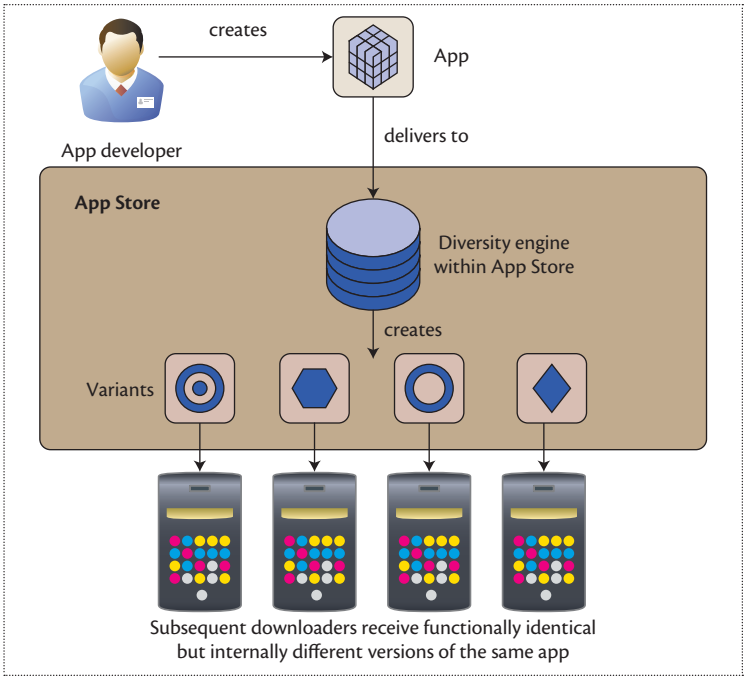

Figure 1. A diversification mechanism can be hidden entirely within an online software delivery system ("App Store") so that it's transparent to both code consumers and software developers.

In this scenario, we think the diversification engine should reside in the App Store (see Figure 1), not on end-user systems. This ensures that users don't disable the diversification engine. All users might agree to diversify code to protect against exploits launched by a third party, but not all will apply diversification to prevent piracy or tampering. In addition, host-based diversification of proprietary software is possible only with binary rewriting, which typically produces slower code relative to binaries diversified during vendor compilation. Finally, it's important to be compatible with code signing. Without it, third parties can inject malware into programs from a trusted source. Mandatory code signing also simplifies blacklisting of applications that contain malware and helps establish provenance.

On the other hand, compile-time diversification isn't necessarily practical because it's computationally expensive to recompile and link a new program variant for every user. However, diversifying a million binaries doesn't necessarily cost a million times more than diversifying it once. In essence, compilers consist of a series of code transformations arranged to form a pipeline. With this architecture, we can easily add a diversifying stage by adding another step to the pipeline. Code

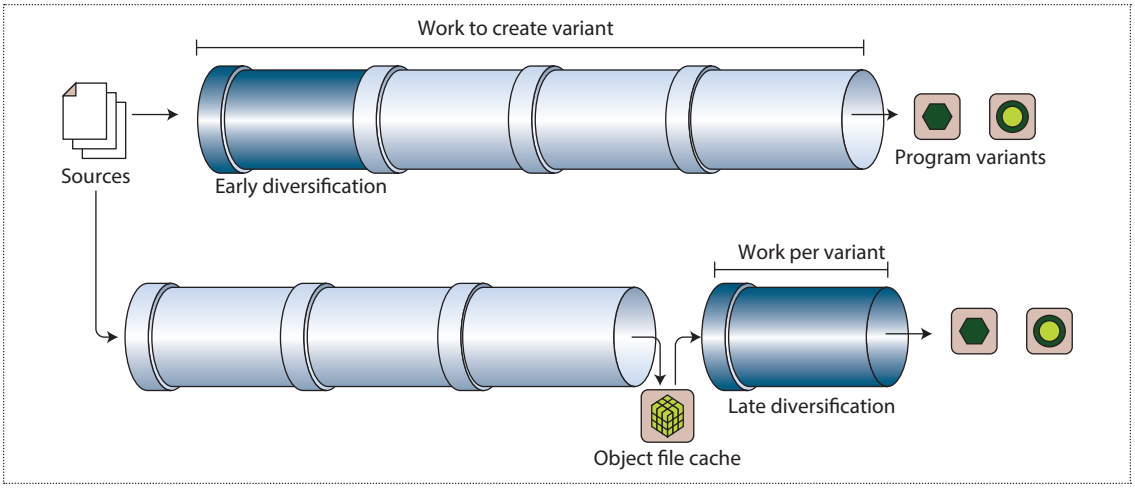

Figure 2. The effort required to create diversified program binaries from sources depends on two factors—the point in time at which diversification occurs in the compilation pipeline (later is better) and when the diversification-invariant work can be cached during compilation.

transformations that occur before the diversification stage are invariant with respect to generation of additional diversified binaries and needn't be recompiled. Figure 2 illustrates how the potential to cache the compilation work depends on the point in time at which diversification occurs. Simply put, later is better. Our multicompiler, built on top of the LLVM compilation framework, performs diversification in the code-generation step, which happens very late (see the lower half of Figure 2). We cache as much work as possible by writing out LLVM bitcode before the code-generation step and restarting the compilation from that point when creating additional variants. With bitcode caching, the time to create a variant is cut roughly in half.

Part of the reason we're not seeing even greater speedups is that code generation—including instruction selection, scheduling, and register allocation—is time consuming. Alternatively, we can perform transformations such as NOP insertion and instruction scheduling on the assembly code emitted by the compiler; this leads to additional savings because code generation becomes redundant.

Our approach analyzes assembly files and computes diversification opportunities ahead of time. Examples include the places NOPs can be inserted and valid instruction schedules. We use annotations to capture the analysis results. A fast template-processing pass reads the annotated assembly files and diversifies them before assembling the final object file. This reduces the time to create a diversified binary. For example, using Firefox version 17, the time decreases from 1 hour and 41 minutes to 25 minutes—a 75 percent improvement. We're exploring ways to speed up build processes, which are often I/O bound and contain repetitive or superfluous work, and see potential for further optimization. For instance, we've reduced the time to diversify several software packages by more than 92 percent: diversifying GNU `make` causes a reduction from 6.17 to 0.44 seconds and diversifying `vim` causes a reduction from 52.60 to 4.02 seconds by recording makefile actions and replaying only the essential ones.

We assume diversified binaries are distributed via an App Store hosted by an infrastructure-as-a-service (IaaS) provider such as Amazon, Google, or Microsoft. It's natural to assume that software developers will perform diversification in the cloud since this has several advantages over an in-house solution. The multiplexing of computing resources gives IaaS providers economies of scale. Because of intensive competition among the top providers, these savings are handed down to the customers. Furthermore, cloud computing requires no upfront investment, and resources are billed according to actual use. This means that computing resources to diversify binaries can be scaled up and down in response to the number of download requests.

In addition, we're optimizing the use of cloud computing resources when diversifying binaries. IaaS providers offer a range of hardware tiers with various processing, RAM, and storage capabilities. Excess capacity within each tier is sold at spot prices that vary according to demand. Permanent storage is similarly tiered according to availability and resilience. This means we can satisfy user demand for diversified binaries by maintaining a stockpile of prebuilt diversified binaries, so downloads start instantly like today's software downloads. If the stockpile is not running low, we choose to replenish it only when spot prices are favorable. If it needs quick restocking, we use

regular, on-demand instances that, in turn, cost more. In the unlikely case that this strategy can't keep up with demand at a reasonable cost, we can reduce the diversity level—and thus security—slightly. For instance, we can choose to distribute the same binary twice rather than once. When done correctly and when there are millions of users, attackers' chances of obtaining the same binary as their victims are small.

We believe that concerns about manufacturing and distribution problems will turn out to be nonissues in practice. Similarly, concerns that compiler-based diversity can't protect legacy binaries will go away once our multicompiler is coupled with a decompiler such as SecondWrite. Adding `librando` to the mix rounds out the types of code we can successfully diversify: legacy code, code that can be recompiled, and code yet to be compiled.

How Far Along Are We?

Recent years have brought tremendous progress in the shift from a software monoculture to a diverse software ecosystem. Researchers have several tools at their disposal to diversify existing software from either source or binary code, including black-box diversification approaches to diversify dynamically generated native machine code emitted by a JIT compiler.

However, not even the combined approach we outlined addresses all challenges in creating a diverse ecosystem. First, many major operating systems and mature software packages support automatic error reporting. In the presence of a fully diversified software stack, a client's error report doesn't directly identify the actual problem of the specific instance of a binary running on the client computer. Depending on the diversification techniques used, we would need to reliably "relocate" the reported error information in such a way that cybercriminals can't abuse this mechanism.

Second and in a similar vein, we need to address the software update mechanism, which depends on the software monoculture. Because a client's binary image will vary substantially among hosts, a single patch can't update all these hosts. A trivial solution to this problem is to hand out new versions of the software to each user. This has additional security benefits, as a host machine's software ecosystem would be in constant flux. On common desktop machines with fast Internet connections, this approach might not be a problem; however, mobile devices with expensive data plans would suffer from this update provisioning system. One solution is shipping diversified updates, customized for each binary running on end users' hosts.

To undo diversification and create customized updates, a diversified binary must know the random seed the diversification engine used to create it. Unfortunately, just adding this random seed to a binary poses a substantial security risk: if leaked, attackers can recreate the whole binary and use it to create a targeted code-reuse attack. The computer security community's solution to this problem is secret sharing.[12] Adapted to our problem, we would split the random seed in half: the binary receives one half, and we store the other half in the cloud. Compromising either one of these parties won't yield the complete seed necessary to recreate the binary versions, and therefore both parties are protected.

Besides securing both the cloud and its clients, this technique has other favorable properties. For example, the cloud could periodically remove entries from its database of half-keys. Consequently, a client requesting a patch would receive a fresh binary image. This is similar to a decoy, because attackers eavesdropping on the communication between clients and the cloud wouldn't be able to infer any meaningful information.

A third challenge relates to preservation of correctness after diversification. Current best practice dictates that a binary must pass a gamut of automated tests as well as alpha and beta testing by early adopters before it's released to the entire user base. Some software vendors might be reluctant to distribute diversified binaries that don't undergo the same level of testing. However, advances in correctness testing have made compilers some of the most mature and reliable computer programs in existence.[13] In our experience, diversification techniques such as NOP insertion and register allocation randomization are sufficiently simple that they chiefly rely on the correctness of the compiler's underlying code analysis and optimization framework. Moreover, to increase confidence in the diversification process, software vendors can scale up their automated testing procedures to run on several diversified binaries instead of testing just one undiversified version. The resulting increase in computational resources can be addressed by moving the testing procedures to a computing cloud.

Reach of Diversity

In this article, we focus on the protection of machine code in binaries against code-reuse attacks. However, artificial diversity can be applied at the source code level, too. Code randomization has been proposed as a counter to code injection attacks such as SQL injection and cross-site scripting.

We're confident that researchers will propose new applications of artificial diversity in response to new offensive techniques. However, we don't think of artificial software diversity as a universal panacea, but we suspect that defenses such as $W \oplus X$, stack canaries, and fault isolation will continue to provide supplemental coverage and defense-in-depth. Software diversity

works by randomizing implementation details; attacks that rely on defective program logic—regardless of its implementation—remain unaffected because diversity preserves program semantics.

In addition, researchers have demonstrated information leakage attacks that are designed specifically to circumvent software diversity. In the presence of arbitrary memory disclosures and scripting capabilities, attackers can analyze the target binary and generate code-reuse attacks on the target machine "just in time."[14] This effectively shifts the required defense from preventing code reuse to preventing memory disclosures, especially in the context of attacker-controlled scripting environments, such as Web browsers. Although this attack is possible, diversity raises the bar and forces attackers to use sophisticated and difficult attacks.

Software diversity targets properties fundamental to attacks on low-level code: knowledge of implementation details and the ability to replicate the victim environment. Diversifying program implementations not only stops a range of known attacks, it might also counter yet unseen attack types.

Further research is warranted to put software diversity into a unifying framework and to distinguish the range of attacks it prevents from those it doesn't. For example, we lack commonly agreed-on metrics and measurements to compare the security afforded by two diversifying transformations with one another and with competing techniques.

Considering how diversity can transition into practice is equally important. In that respect, we find cloud-based, compile-time diversification augmented by binary rewriting surprisingly attractive. It gets the broad strokes right: the performance impact is minimal, it can protect code regardless of how it was produced or whether it was signed, and it produces diversified binaries cost-effectively without enlarging the system's attack surface.

However, like its alternatives, cloud-based diversification isn't a silver bullet. Attackers might be able to latch onto implementation aspects that nobody thought to diversify. Luckily, once the investment in a diversified ecosystem is made, adding to the set of code randomization techniques is as easy and transparent as introducing new compiler optimizations and fixes via regular updates.

Work remains to be done on error reporting and patching of diversified software. To address these problems, the diversification process must be reproducible. We can drive the diversification using a pseudorandom sequence expanded from a seed value; the seed value then becomes the only secret in the system requiring protection. Rather than storing the whole seed in a single place, again, we can use secret-sharing techniques to store one part of the seed in the cloud and another on the client side.

With most of the major obstacles cleared, we expect the arrival of diversified software in the commercial marketplace within the next three years. There will be some initial resistance as people adapt to changes to established security practices, but in the end, we'll all be more secure. Instead of a single target binary replicated across millions of computers, we'll present adversaries with a moving target in the form of large numbers of binary variations and no proper way of matching attack vectors to target binaries. ■

Acknowledgments

This material is based on work partially supported by the Defense Advanced Research Projects Agency under contracts D11PC20024 and N660001-1-2-4014, by the National Science Foundation under grant CCF-1117162, and by a gift from Google. 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, its Contracting Agents, the National Science Foundation, or any other agency of the US government.

References

1. C.S. Collberg et al., "Distributed Application Tamper Detection via Continuous Software Updates," *Proc. 28th Ann. Computer Security Applications Conf.* (ACSAC 12), 2012, pp. 319–328.
2. B. Coppens, "Protecting Your Software Updates," *IEEE Security & Privacy*, vol. 11, no. 2, 2013, pp. 47–54.
3. D.W. Williams et al., "Security through Diversity: Leveraging Virtual Machine Technology," *IEEE Security & Privacy*, vol. 7, no. 1, 2009, pp. 26–33.
4. S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *Proc. 14th Usenix Security Symp.*, 2005, pp. 271–286.
5. R. Wartell et al., "Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code," *Proc. 19th ACM Conf. Computer and Communications Security* (CCS 12), 2012, pp. 157–168.
6. J. Hiser et al., "ILR: Where'd My Gadgets Go?," *Proc. 33rd IEEE Symp. Security and Privacy* (S&P 12), 2012, pp. 571–585.
7. V. Pappas, M. Polychronakis, and A.D. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization," *Proc. 33rd IEEE Symp. Security and Privacy* (S&P 12), 2012, pp. 601–615.
8. C. Giuffrida, A. Kuijsten, and A.S. Tanenbaum, "Enhanced Operating System Security through Efficient and Fine-Grained Address Space Randomization," *Proc. 21st Usenix Security Symp.*, 2012, pp. 475–490.

9. A. Homescu et al., "Profile-Guided Automatic Software Diversity," *Proc. 11th IEEE/ACM Int'l Symp. Code Generation and Optimization* (CGO 13), 2013, pp. 1–11.
10. K. Anand et al., "A Compiler-Level Intermediate Representation Based Binary Analysis and Rewriting System," *Proc. 8th EuroSys Conf.*, 2013, pp. 295–308.
11. A. Homescu et al., "Librando: Transparent Code Randomization for Just-in-Time Compilers," *Proc. 20th ACM Conf. Computer and Communications Security* (CCS 13), 2013, pp. 993–1004.
12. A. Shamir, "How to Share a Secret," *Comm. ACM*, vol. 22, no. 11, 1979, pp. 612–613.
13. M. Franz, "E unibus pluram: Massive-Scale Software Diversity as a Defense Mechanism," *Proc. 2010 Workshop on New Security Paradigms*, 2010, pp. 7–16.
14. K.Z. Snow et al., "Just-in-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization," *Proc. 34th IEEE Symp. Security and Privacy* (S&P 13), 2013, pp. 574–588.

Per Larsen is a postdoctoral scholar in the University of California, Irvine's Department of Computer Science. His research interests include security, compilers, code profiling, and optimization. Larsen received a PhD in computer science from the Technical University of Denmark. Contact him at perl@uci.edu.

Stefan Brunthaler is a postdoctoral scholar in the University of California, Irvine's Department of Computer Science. His research interests include compilation, interpretation, analysis, optimization, and verification. Brunthaler received a doctorate in technical sciences from the Vienna University of Technology. Contact him at s.brunthaler@uci.edu.

Michael Franz is a full processor in the University of California, Irvine's Department of Computer Science. His research interests include systems software, security, compilers, and virtual machines. Franz received a doctorate in technical sciences from ETH Zurich. He's a senior member of IEEE. Contact him at franz@uci.edu.