Diversity as an Enabler for Cyber Resilience

Joel Coffman and Andrew S. Gearhart

ABSTRACT

Dependable systems must continue to provide correct service in the presence of errors, regardless of whether the errors stem from fatigue of hardware components, design mistakes, software defects, or the malicious activity of adversaries. Fault tolerance aims to detect random errors and recover from them automatically. Unfortunately, many critical systems do not consider or properly account for adversaries' malicious actions that are designed to compromise the dependability or security of the system. This article argues for the role of software diversity in the construction of highly dependable and resilient computer systems. Specifically, our focus is the potential benefit of diversity as a defense against cyberattacks. This approach has received some attention in the academic community, but a robust science of software diversity has yet to emerge. Toward this end, and to enable characterization of software diversity strategies, we present several novel methods of differentiating diversified sets of programs and highlight areas of future investigation.

INTRODUCTION

Safety-critical systems pervade the modern world. Unfortunately, there are numerous examples of their failures.[1–4] Failures are particularly relevant in the context of cyberattacks where adversaries intentionally compromise the dependability or security of the system. Although isolation and one-off designs have historically mitigated such threats to safety-critical systems, increased connectivity—epitomized by the Internet of Things (IoT)—demands increased vigilance. For example, the Stuxnet worm demonstrated that even isolated networks are not immune to cyberattack.[5]

The proliferation of mass-market software targeting a single instruction set architecture (ISA) creates risks: a vulnerability in an application represents a common avenue of attack against all instances of that application.[6,7] This existing hardware and software "monoculture"[7] stands in marked contrast to biological development. Scientists now understand the importance of diversity to a species'—or even to an entire ecosystem's—health and survival against natural catastrophes and even targeted eradication.[8,9] Applying biological diversity principles to software may well reap significant dividends—namely, improvements in software resilience to cyberattacks.

In cybersecurity, failure independence through diversity means that a successful attack impacts a single instance of the software package instead of all instances of it. Thus, instead of a one-time cost for an initial compromise that applies to many systems, the attacker incurs a cost for each targeted system. Diversity applies in any situation that requires multiple instances of a software capability. This includes an application running on multiple networked machines, mass-produced embedded software running on a suite of IoT devices, or code running on a stand-alone system. The use of diversity in real-world computer systems has historical precedence; design diversity has long been recognized as a method to improve dependability.[10] As we will discuss, however, software diversity is not a cybersecurity panacea because of the challenge of reliably measuring its effectiveness.[11]

We begin this article with an overview of dependability and resilience and then introduce software diversity within the context of dependable and secure computing. Next we discuss several ways to characterize software diversity strategies: via clustering analysis, return-oriented programming (ROP) gadgets, and with a test suite of vulnerable programs. Then we enumerate challenges to using diversity in critical environments. Finally, we conclude and highlight opportunities for future work.

BACKGROUND

A system is dependable if it avoids unacceptable service failures.[12] A failure is an externally observable event that represents deviation from the system's required behavior (e.g., a hard drive crash). Critical systems must limit failures and their consequences. Similarly, resilient systems are also dependable in that they are able to maintain capability in the face of adversity, from both random and malicious threats.

Dependability and Security

Dependability is a multifaceted concept related to, but not synonymous with, security. The following list summarizes attributes of both concepts.

- **Availability:** Maintenance of service to authorized users, or "doing the right thing within the specified response time"[13]
- **Confidentiality:** "The absence of unauthorized disclosure of information"[12]
- **Integrity:** The absence of improper modification to the system or of improper modification (or destruction) of data
- **Maintainability:** The ability of the system to undergo modifications and repairs
- **Reliability:** The system's continuity of correct service
- **Safety:** The absence of catastrophic consequences to user(s) or to the environment

Figure 1 illustrates the relationship among these attributes and the overarching concepts of dependability and security. This characterization subsumes the concept of

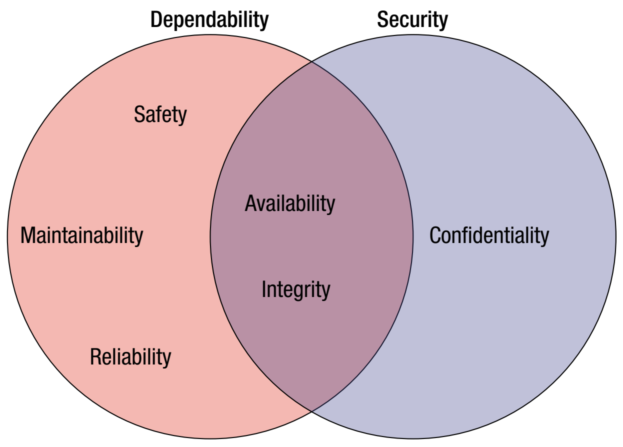

Figure 1. The relationships among dependability and security and their constituent attributes.[12]

software assurance, the "level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted anytime during its lifecycle and that the software functions in the intended manner."[14]

Failures are the major threat to dependability and security. The immediate precursor of a failure is an error, which is "a deviation from accuracy or correctness in state."[15] An error stems from a fault. A fault is "a physical defect or flaw within a hardware or software component."[15] Figure 2 depicts the relationship among faults, errors, and failures.

A system is fault tolerant if it continues to provide correct service in the presence of faults (and ensuing errors).[16] Fault tolerance is often combined with redundancy to guard against physical failures. Unfortunately, fault tolerance via redundancy has limitations: in one well-known example, the loss of the Ariane 5 rocket, an unhandled software exception caused the backup processor to fail, and the primary processor failed immediately afterward as a result of the same error.[17] Consequently, resilience may require diversity.

DIVERSITY

The research literature is replete with software diversity techniques, and successful application of diversity requires combining and synthesizing these approaches. While diversity may also improve the absolute security of individual variants, that is not its sole (or often primary) objective. Instead, diversity should improve the security of the population much as immunization protects a population from disease. Consequently, an

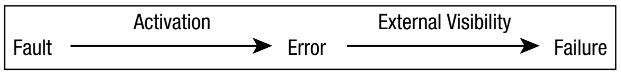
$$\begin{tikzcd}\text{Fault} \arrow[r, "\text{Activation}"] & \text{Error} \arrow[r, "\text{External Visibility}"] & \text{Failure}\end{tikzcd}$$

Figure 2. The relationship between faults, errors, and failures.

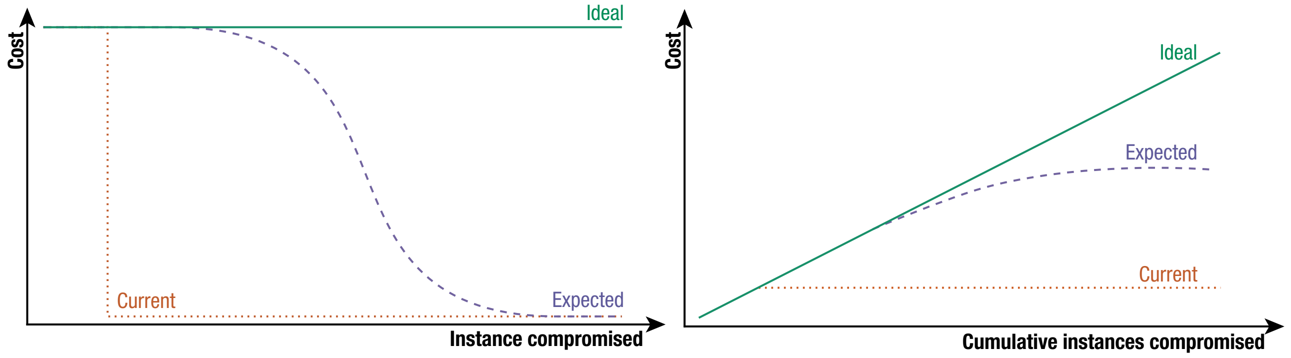

Figure 3. The impact of software diversity to attackers: on the left is the cost to compromise each instance, and on the right is the cumulative cost to compromise multiple instances. In an ideal scenario, each instance of the software application is as expensive to exploit as previously exploited versions. The attacker gains no knowledge from each successful attack that can be leveraged to improve future attacks. With software diversity, the cost of an attack is initially constant but gradually decreases as the attacker identifies commonalities among the variants. Eventually the attacker adapts to the diversity scheme and software diversity yields no additional benefit.

attacker's effort is expected to increase in a nonconstant, nontrivial manner with the number of infected software packages (Fig. 3). Because a generic vulnerability cannot be used to gain access to a system, targeted attacks must implant a specific variant for future exploitation, exfiltrate existing variants for analysis, or perform exploits "online" with a greatly reduced probability of success. These cases represent detectable behavior by the attacker: network exfiltration, application crashes, and anomalous system log entries all serve as indicators of an adversary's presence.

Software diversity strategies can be applied at any step of the software development life cycle, from design to execution to update. The choice of when to diversify may have significant impact on the system's cost and maintainability, although issues with maintainability may be mitigated by automated techniques at compile and execution time. Larsen et al., in a survey of software diversity research, provide additional information about when to diversify (see section 4, When to Diversify, in Ref. 18).

Design diversity is an "approach in which the hardware and software elements that are to be used for multiple computations are not copies, but are independently designed to meet a system's requirements."[19] One of the more widely known techniques is *n*-version programming. In this paradigm, multiple implementations of software are independently produced from a common specification. Care must be taken to ensure that faults occur independently, as the number of coincident failures has been shown to be higher than expected from an assumption of independence.[20] An n-variant system[21] is the evolution of this concept where failure modes are proven independent.

Design diversity is often too expensive to adopt outside of critical systems. This realization has led researchers to consider automated techniques to achieve these goals. A diversifying compiler[22] operates identically to an ordinary compiler but is not constrained to produce the same output when presented identical input. For example, a diversifying compiler need not eliminate dead code or, more precisely, probabilistically decides when to remove dead code. Although one concern is that such variability in the output executable will not guarantee runtime or size "optimality," compilers, in general, can rarely guarantee optimality and rely on several heuristics or execution traces (i.e., profiles) to reduce the space or time required for a computation. The resulting space-time trade-offs presented by these compiler techniques may allow for significant diversity, ideally with significant cost to attackers. Because these transformations are handled by the compiler, they do not affect the maintainability of the source code, and they can be proved correct (i.e., they do not change a program's input/output behavior).

Further, with the proliferation of modular compiler infrastructures, reconfigurable hardware (e.g., field-programmable gate arrays), and virtualization, the tools are now available to produce computers that run on a wide range of ISAs. If each machine used a different ISA, it would be expensive for an adversary to target a single machine within a critical network. To inject code directly into the application or mount an effective code reuse attack, the attacker must first determine the unique ISA for that machine. ISA randomization presents a unique ISA to the attacker by encrypting the base instruction set of the target machine.[23,24] Unlike compiler-based diversity techniques, ISA randomization may be applied to existing binaries and does not require access to the original source code.

Data diversity[25] is orthogonal to the aforementioned approaches. This technique is closely tied to fault tolerance mechanisms and complements other diversity strategies. Data diversity captures the intuition that two input values that are similar should produce similar outputs. For example, the intersection of two lines should be similar to the intersection of those lines when one is shifted by a fixed amount. This general technique holds promise for detecting and thwarting cyberattacks. Even a small perturbation of a malicious input controlled by an adversary may render it useless, contributing to fail-safe design—for example, the difference between crashing an application and gaining control of the system.

ANALYSIS OF COMPILER-BASED SOFTWARE DIVERSITY STRATEGIES

Despite the proliferation of diversity techniques, many security evaluations of diversity strategies are qualitative and based on logical argument.[18] There is no accepted methodology that researchers use to evaluate diversity techniques. This article discusses three novel ways to quantitatively compare diversity strategies, two of which have a clear link to adversary cost (see the sections on ROP gadgets and automated validation). Our analysis focuses on compiler-generated diversity and, in particular, the diversity techniques implemented by the open-source *Multicompiler*[26] and *Obfuscator-LLVM*.[27] A brief description of these techniques follows:

- **Garbage code insertion:** This strategy probabilistically inserts 0 or 1 garbage instructions prior to the current instruction. The garbage code varies from the x86 NOP instruction to instructions that preserve the processor state (e.g., mov esp, esp).
- **Instruction substitution:** In many cases, arithmetic operations can be expressed differently—for example, b + c = b − (−c) = −(−b + (−c)). When possible, an equivalent instruction sequence is substituted for binary and Boolean operations on integers.
- **Schedule randomization:** Conceptually, dependencies among instructions form a directed acyclic graph, and this transformation selects an arbitrary instruction from those that are eligible to appear next.
- **Bogus control flow:** This transformation modifies the control flow graph of a function by adding a basic block with an opaque predicate[28] prior to the current basic block.
- **Control flow flattening:** This transformation obscures the call graph by replacing direct jumps between basic blocks with indirect jumps through a “jump table.”[29]
- **Function shuffling:** This transformation permutes the order of functions in the object code produced by the compiler for each compilation unit.

In our early experiments (see the sections on clustering and ROP gadgets), we selected a single data set for evaluation: the GNU core utilities (http://www.gnu.org/s/coreutils). This data set includes over 100 C-language programs that compile rapidly, making them amenable to the generation of many variants. The experiments described in the section on automated validation evaluate diversity strategies via a suite of vulnerable programs developed for the Defense Advanced Research Projects Agency (DARPA) Cyber Grand Challenge program, allowing us to directly evaluate the impact of diversity on exploits.

Clustering

At the beginning of our diversity research, we asked a basic question: can diversity strategies be quantitatively differentiated? Our initial approach to this question involved using common methods from text clustering. Such an approach has the advantage of being agnostic to particular diversity strategies. In this section, we analyze several methods of generating feature vectors from diversified core utilities applications and then comparing the resulting clusters.

To cluster a data set, elements must first be represented as vectors in a high-dimensional space. These are called feature vectors. Given a compiled application (i.e., a binary), we define feature vector creation as three steps: disassembly, normalization, and conversion. After disassembly, instructions (of the form operation operand,…,operand) are normalized by stemming operations (e.g., addpd becomes addp), and operands are mapped to three generic types (similar to Sæbjørnsen et al.[30]). These types indicate that the original operand was a register, constant, or a memory reference. After normalization, instructions are converted to vectors via three approaches: (*i*) raw instruction frequencies; (*ii*) term-frequency, inverse document frequency scaling (unless otherwise noted, transformations and models utilize *scikit-learn*, http://scikit-learn.org/, implementations with default parameters); and (*iii*) a doc2vec model.[31] Parameters for the gensim doc2vec model follow those described by Lau and Baldwin.[32]

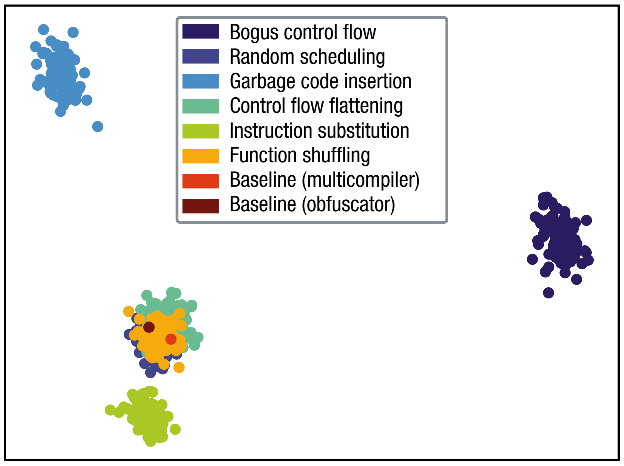

Figure 4. doc2vec feature vectors for *dir* with labels from ground truth.

In Fig. 4, doc2vec feature vectors have been reduced to two dimensions via principal component analysis. This allows for visualization and shows a clear distinction for the application *dir* between several strategies, with random scheduling, control flow flattening, and function shuffling appearing quite similar to the baseline undiversified binaries. (Note that there are two baselines, one for each of the two diversifying compilers used to generate variants.) This is not surprising for random scheduling and function shuffling (which only reorder instructions), and the similarity of these strategies to the baselines was consistent across applications. The similarity of control flow flattening to the baselines is curious and appeared in a subset of other applications. Further investigation of the doc2vec features is required to understand this particular strategy's proximity to the baselines.

Beyond qualitative observation of clusters, we used an agglomerative clustering algorithm to produce guesses as to features generated via the same diversity strategy. These approximate labels were then compared to ground truth. For example, the clustering algorithm may guess that two features are generated by the same diversity strategy, when in fact they are the product of different strategies. Cluster guesses were then compared to ground truth via the adjusted Rand index (ARI), with doc2vec demonstrating a clear advantage in producing clusters that correctly differentiate diversity strategies. [Provided with cluster labels generated by a clustering algorithm and also ground-truth labels, the ARI counts differences between sample pairs assigned in either (*i*) the same or (*ii*) different clusters. These counts are compared between the algorithm-derived and ground-truth labels, and the score is adjusted for random chance. This provides a similarity measure between two clusterings and is a score between –1.0 and 1.0. A score of 1.0 is a match.[33]] Figure 5 shows quartile information for these ARI experiments, highlighting outliers (the small circles in the figure). These results illustrate the potential of semantic embeddings such as doc2vec to differentiate diversity strategies.

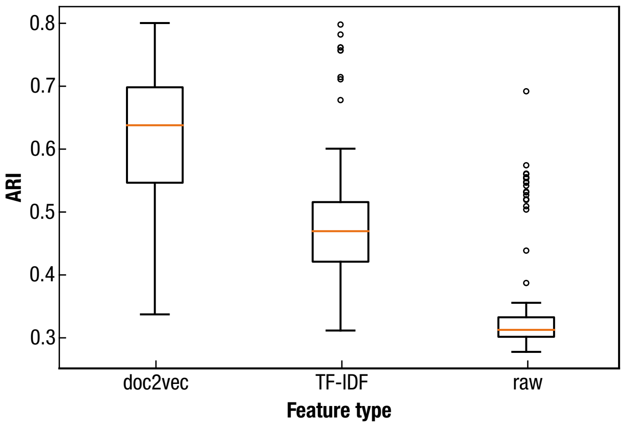

Figure 5. ARI comparison of feature types. TF-IDF, term-frequency, inverse document frequency.

ROP Gadgets

Our clustering approach in the preceding section is applicable across a range of diversity strategies and was able to differentiate variants generated by several strategies. It is unclear, however, how to link these results with attacker costs. Practically, any measure of the effectiveness of diversity for security should be driven by a measurement of the reusability of a single exploit across a population of variants. To this end, we proposed counting the number of ROP gadgets, an important building block used by modern exploits, shared across a diversified population. Specifically, a ROP gadget is a sequence of bytes in a program that can be interpreted as valid, unprivileged, nonbranching instructions terminating with a return instruction. Such gadgets are the building blocks of a ROP attack, a class of code reuse attacks wherein an attacker executes a series of ROP gadgets in a "ROP chain" to accomplish his or her objective.

With the hypothesis that an attacker's effort increases as the common set of executable code snippets (ROP gadgets) decreases across application variants, we explore how different diversification techniques affect the set of ROP gadgets available to an attacker. Successful reuse of an exploit against diversified application variants requires all variants to share the same code snippets used in the ROP attack.

To explore this common set of ROP gadgets, we compiled 100 diversified variants of the core utilities applications. We then used Floyd's sampling algorithm[34] to select 4000 unique combinations of variants for experimentation. This approach provides a uniform number of samples (i.e., an even random sample) for our analysis. We assume that a gadget is only useful to an attacker when the gadget has the same functionality at the same address. Otherwise, the attack must be modified for use against other binaries.

Figure 6 displays the percentage of surviving ROP gadgets across our random sample of variant combinations with sizes from 2 to 16 binaries. For each diversity technique, Fig. 6 graphs the mean of the median values of surviving gadgets for each binary in the GNU core utilities. Of the strategies evaluated, we found control flow flattening to be most effective. All diversification techniques except function shuffling eventually achieve a survival rate of less than 5% for larger sets of binaries. Function shuffling and schedule randomization improve rapidly from two to four binaries, starting at a 26% and 9% survival rate but eventually converging to an 8% and a 3% survival rate.

Many attacks against common applications require only a handful of unique gadgets in the payload (approximately 10–20).[35] Our analysis suggests that these diversity techniques are on the threshold of preventing these attacks outright, leaving attackers with the challenge of adapting their payload to use these surviving gadgets, which may not always be feasible.

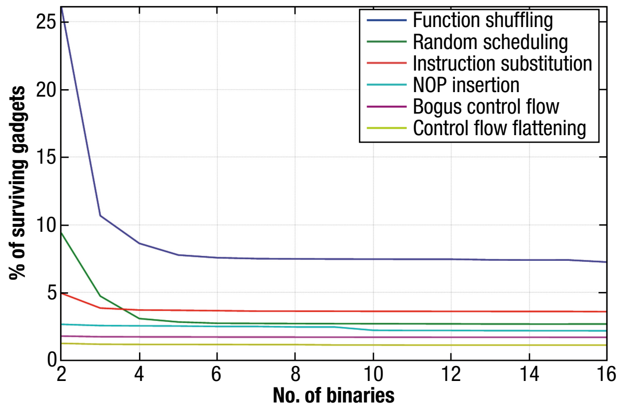

Figure 6. Median gadget survival across the GNU core utilities.

We created a visualization to understand the relationship between the locations of gadgets common to multiple variants (Fig. 7). Usually surviving gadgets reside at the beginning of the binary where the diversity technique (function shuffling) has had little opportunity to influence the output. Occasionally techniques like function shuffling will, by chance, align the same functions late in the binary, creating a zone of surviving gadgets deep in the binary. While it is logical to consider the combination of various diversity techniques, preliminary analysis indicated that the more effective techniques dominate, causing combinations to perform similarly to high-performing individual techniques.

Interestingly, the tail of our effectiveness distributions shows little change as the population increases from 6 to 16 variants (Fig. 6). Other researchers' results (e.g., Ref. 35) also suggest that some gadgets simply cannot be diversified away; our results support this finding.

Despite a small set of residual gadgets, attackers are still left with what might be a significant challenge. An intriguing question remains: is it possible to identify *a priori* the gadgets that survive diversification? If the core surviving gadgets are identifiable in some fashion, attackers will naturally adjust their techniques to target the gadgets that are present in all variants. Although we did identify the likely location of surviving gadgets that appear at the same location in memory (Fig. 7), additional research is necessary to identify other relationships among these surviving gadgets that attackers could exploit.

Automated Validation via a Set of Vulnerable Binaries

The analysis diversity strategies via clusters and common ROP gadgets are at best indirect methods to measure attacker effort. Clustering relies on distances between sets of features in an abstract space, something difficult to link back to attackers. Further, our ROP analysis assumes that fewer shared gadgets is correlated with attacker effort. While the above approaches are useful to build intuition, we would like to compare strategies for their direct impact on the behavior of exploits.

In 2015 and 2016, DARPA's Cyber Grand Challenge program developed a set of challenge binaries (CBs) with known vulnerabilities and working exploits. These CBs present an opportunity to study diversity, and our basic idea was to evaluate the effectiveness of diversity techniques by measuring how well they stop exploits across a population of variants. Some portion of the diversified variants should be immune to the exploit if diversity is an effective defense. Two types of exploits were examined:

- **Type 1:** These exploits cause the program to fault at an address negotiated with the testing system, with one of the general-purpose registers set to a second negotiated value. In the real world, these attacks allow an attacker to gain control of the program, potentially compromising the entire system.

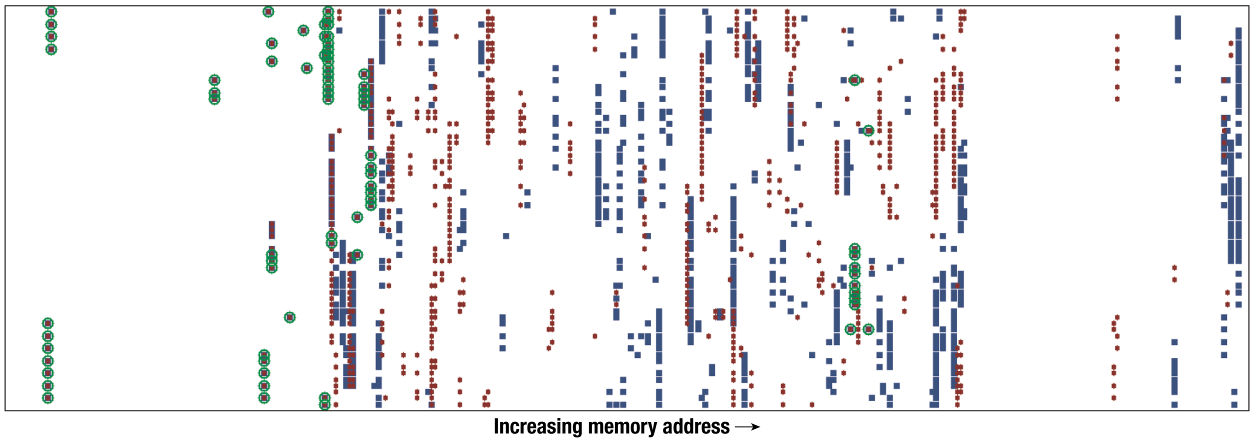

Figure 7. Gadget locations in two variants (red, blue) of the *dirname* application, with surviving gadgets circled in green. Function shuffling was used to diversify the two variants. This visualization shows the executable memory segments of the binaries normalized to the start of the first gadget (with some padding for visual clarity) and aligned on 64-byte addresses. Each column represents 64 bytes of memory (e.g., the first column shows memory addresses 0–63).

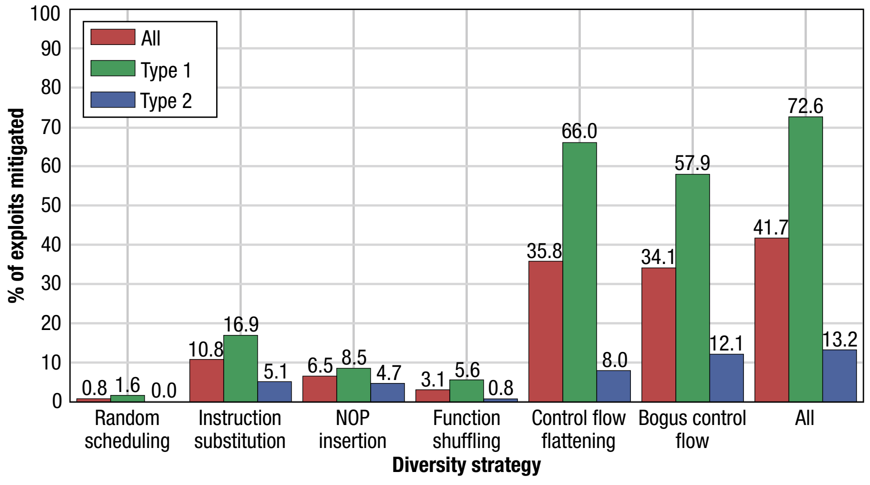

Figure 8. Percentage of exploits mitigated across all variants.

- **Type 2:** These exploits, after negotiation with the testing framework, can read a certain number of bytes from anywhere within a specific region of memory. In a real program, this could lead to the exposure of sensitive data such as administrator passwords or restricted information, similar to the Heartbleed vulnerability (http://heartbleed.com).

To perform our experiments, we built each CB in eight ways, once with the standard compiler, once for each of the six diversity techniques, and once with all the diversity techniques enabled at the same time. We repeated this process to produce a population of 100 diversified binaries for each (CB, diversity strategy) pair. We then ran exploits against these binaries to evaluate the effectiveness of the diversity at mitigating the attacks.

Figure 8 displays the percentage of exploits mitigated across all the variants. It represents the success rate taken over every variant for each exploit. Therefore, it indicates the average coverage for each strategy. For example, bogus control flow has a 34.1% effectiveness. This means that exploits were successfully mitigated (i.e., broken) in approximately 4,194 of the 12,300 tests (123 exploits * 100 variants). The graph illustrates a disparity between type 1 and type 2 exploits. Whereas some diversification strategies demonstrated considerable success in mitigating type 1 exploits, such as control flow flattening at 66.0%, all the techniques examined were largely ineffective for type 2 exploits. At best, diversification was able to prevent 13.2% of type 2 exploits, when all strategies were combined. At worst, diversification had no effect at all, as in the case of random scheduling.

How well a diversification strategy performs is of critical import to how broadly it is adopted. In particular, adoption of new security techniques often requires the execution time penalty to be less than 5–10%.[36] This facet of software diversity has been well studied in general, with results often reported in the aforementioned range (see Table III in Ref. 18). Figure 9 shows our performance results. Four of the strategies (NOP insertion, random scheduling, function shuffling, and instruction substitution) have a marginal overhead. The impact of bogus control flow is more pronounced (≈40% overhead) but may be tolerable given its impact to exploits and aim to hinder reverse engineering. In comparison, control flow flattening increases execution time by ≈360%; this overhead prohibits the use of this diversification strategy in most real-world settings except when its anti-tamper properties are critical. Combining all the techniques further decreases performance—a mean increase in execution time of ≈793%. Although the significant overhead of these final three diversification strategies may seem high, our results are in line with prior research that found “a performance penalty by a factor of less than 10,”[27] particularly in computationally intensive code.

These experiments indicate significant variation in the effectiveness of software diversity strategies at mitigating exploits. Intuitively, there is a link between mitigation success and two dimensions of diversification: scope and extent. We note that random scheduling, instruction substitution, and NOP insertion all operate on instruction-level scope, each making decisions on a per-instruction basis. Function shuffling, on the other hand, operates at a program scope. These strategies have little success at exploit mitigation, so scope does not appear to be a sufficient quality of the strategy. Fur-

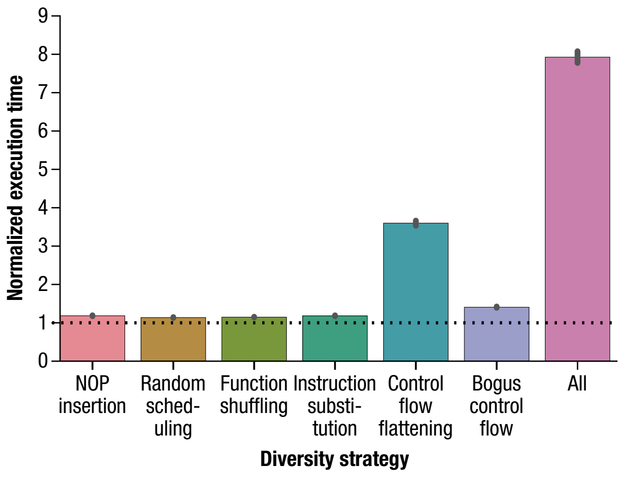

Figure 9. Performance overhead of diversity techniques. Times are normalized to the mean of five runs of the baseline (i.e., undiversified) program. Because of extreme outliers, this graph includes only the 99th percentile of the results of each diversity strategy.

ther investigation is required to quantify the extent of a diversity strategy.

While these results are intriguing, they are not fully representative of those that would be encountered in practice. The purpose of the Cyber Grand Challenge exploits is only to demonstrate the presence of a vulnerability, rather than the development of a complete attack. Real-world attacks are more extensive and often occur in stages, providing greater opportunity for mitigation through diversification. We believe that our results likely underestimate the impact of diversity in real-world scenarios.

CHALLENGES

From a practical perspective, several outstanding challenges must be resolved before a secure, scalable, and broadly applicable software diversity system can be created. Fundamentally, there are two key open questions:

1. How can we measure and characterize the impact of a particular diversity strategy on security and dependability?
2. Does diversity inherently conflict with existing policies, particularly the accreditation and certification requirements for critical systems?

This section discusses these questions and highlights particular areas that require further investigation.

Impact of Diversity

As discussed earlier, the current software monoculture contributes to the reuse of exploits across instances of an application. Diversity breaks the assumption of software consistency with the goal of improving cyber resilience but may inadvertently reduce resilience in other areas because of the increased difficulty of managing diversified variants and preventing the use of existing security tools such as those that whitelist known-good software. (Whitelisting tools store cryptographic signatures of binaries that are considered "safe" for a particular enterprise. If the signature generated by a binary differs from the safe signature, the unknown binary is flagged as potentially malicious. Multiple variants of an application would require such a system to keep track of signatures for each variant independently, potentially significantly increasing the required storage and runtime for effective whitelisting to be performed, or adopt alternative methods, e.g., code signing, to designate trusted binaries.) Further, a diversity strategy may result in a small subset of variants that are actually easier to exploit even if the entire population demonstrates improved average cyber resilience. Depending on the mission scenario, this situation may be unacceptable.

As noted by Larsen et al.,[18] there is little research on metrics that measure the diversity produced by various automated strategies. Logical arguments about security are common, as is measuring entropy,[37,38] but no research to date has shown a relationship between these approaches and adversaries' efforts. To be truly useful for cybersecurity, such metrics should be demonstrably correlated with attacker effort. This is a difficult problem that is not easily handled via red team studies.[39] Recent papers that compare diversity strategies via symbolic execution[40,41] and the results presented in this article are a step in this direction. Future work may use formal methods (e.g., abstract interpretation) or machine learning to define additional means of differentiating the populations of variants generated by diversity strategies.

Diversity techniques may serve as excellent obfuscators and improve cyber resilience on average. It is crucial to understand the relationship between diversity and the targeted classes of vulnerability. Approaches that may mitigate this risk include combining multiple diversity techniques and creating a larger number of variants than is needed, and then down-sampling based on a set of criteria. For example, a diversity technique could generate 1000 variants and use symbolic execution to choose the 100 variants that are most "secure" (or that adhere to a set of performance requirements). This strategy raises

DIVERSITY FOR SOFTWARE TESTING

As an additional interesting note, we observed issues with several CBs during the testing process. When CBs were built without diversification, provided code patches successfully prevented all exploits for these CBs (as expected). When they were built with diversification, however, three of the CBs always terminated with a segmentation fault. Initially, we suspected a flaw in the implementation of the diversity technique. On further inspection, we determined that diversification was not responsible for these failures. Rather, the process of diversification revealed the following latent faults:

- One program depended on an uninitialized variable (which was assumed to be initialized to zero), leading to a buffer overflow.
- One program lacked proper bounds checking, leading to a segmentation fault.
- One program failed because of a compiler fault in LLVM 3.4, in which an unaligned load led to a segmentation fault following a register spill onto the stack.

The exposure of these faults highlights another potential benefit of diversification. Whereas a reliance on undefined behavior can go unnoticed under normal compilation, deviation across variants in diversified binaries breaks such assumptions, revealing potential vulnerabilities. This ability to expose code faults suggests that automated diversity may be used as a low-cost "booster" for software testing.

questions related to the yield of a diversity technique, another direction of potential future research.

Finally, diversity strategies to protect populations of binaries and to improve resilience (e.g., *n*-version programming) assume the failure independence of variants under similar inputs. This assumption should be rigorously confirmed on any diversity system. With *n*-version programming, the problem is subtle. Not only must variants behave differently under malicious inputs, but they must also exhibit these differences in a manner that is rapidly detectable and verifiable. This is a significant challenge that requires careful specification of diversity strategies, as well as creative representation of program state within the complete system.

Policy and Certification

Beyond the challenges associated with scaling and validating a system that includes diversity, the presence of unique software variants may conflict with policies regarding the certification of mission- and safety-critical software. For example, the nondeterministic nature of some automated diversity techniques presents difficulties when attempting to confirm alignment with a specification. An example would be confirming that a critical software function conforms to timing requirements after diversification. (Although this article focuses on software diversity for resilience, such challenges are also pertinent to diversified hardware devices.) Most academic research focuses on preservation of a program's semantics (i.e., correctness)[42] rather than its temporal properties. However, the latter must be addressed before applying diversity to safety-critical embedded devices that have tight timing bounds and resource constraints.

The certification process for critical software often includes a number of defined stages involving specification checks, independent code reviews, static analysis, and live testing. This process may involve many participants and significant expense, and individual stages may not scale efficiently with the number of software variants. Live testing is particularly problematic in this context because diversity's power relies on subtle differences among variants. The incorporation of software diversity into critical systems requires understanding the time and financial cost of current certification steps, potential scaling of the process with software variants, and any changes required to incorporate automated diversity. An example paradigm shift that may be required is eliminating policies that implicitly depend on the compiled form of software (e.g., live testing with executable binaries) because such results would not apply to multiple variants. As an alternative, certification might test one variant but also require greater use of formal methods to prove the correctness of the source code and also the correctness of the generation of variants (e.g., a certified compiler[43]).

CONCLUSIONS

Geer[44] summarizes the typical reactions to diversity in the context of cybersecurity: (*i*) embrace monoculture, since it allows you to get consistent risk management exactly because everything is alike; or (*ii*) run from monoculture in the name of survivability.

Automated software diversity promises to improve system resilience to both random faults and cyber threats. With recent advances in compiler design, binary rewriting, and cloud computing, the practical development and application of diversity strategies is becoming feasible. However, much research is still required to develop a science of system resilience that accounts for the effect of diversity on both random faults and determined attackers. Further, current security tools and processes for critical software certification may need to be replaced or modified to account for deliberately breaking the software monoculture.

Diversity is not a cybersecurity panacea. It does not obviate the need for traditional techniques to ensure a system's dependability and security, and we recommend that such techniques be applied throughout the life cycle of acquisition programs. Nevertheless, diversity complements such approaches, adding defense in depth and resilience against many threats. Automated approaches to software diversity—such as diversifying compilers and ISA randomization—hold the most promise as a cost-effective form of cyber defense. These same techniques hold promise for uncovering dormant faults, as evidenced by our results that demonstrate the ability of diversified variants to uncover compiler bugs.

REFERENCES

[1]Long, T., "Sept. 26, 1983: The Man Who Saved the World by Doing . . . Nothing," *Wired*, 26 Sept 2007.

[2]Leveson, N. G., and Turner, C. S., "An Investigation of the Therac-25 Accidents," *Computer* **26**(7), 18–41 (1993).

[3]Burke, D., *All Circuits Are Busy Now: The 1990 AT&T Long Distance Network Collapse*, Report CSC440-01, California Polytechnic State University, San Luis Obispo, CA (1995).

[4]*Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, Report IMTEC-92-26, U.S. Government Accountability Office, Washington, DC (1992).

[5]Chen, T., and Abu-Nimeh, S., "Lessons from Stuxnet," *Computer* **44**(4), 91–93 (2011).

[6]Forrest, S., Somayaji, A., and Ackley, D. H., "Building Diverse Computer Systems," in *Proc. 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, MA, pp. 67–72 (May 1997).

[7]Geer, D., Schneier, B., Metzger, P., Bace, R., and Gutmann, P., *Cyber-Insecurity: The Cost of Monopoly*, Computer & Communications Industry Association, Washington DC (2003).

[8]Taylor, C., and Alves-Foss, J., "Diversity as a Computer Defense Mechanism," in *Proc. 2005 Workshop on New Security Paradigms*, Lake Arrowhead, CA, pp. 11–14 (2005).

[9]Seeley, T. D., and Tarpy, D. R., "Queen Promiscuity Lowers Disease within Honeybee Colonies," *Proc. Biol. Sci.* **274**(1606), 67–72 (2007).

[10]Lardner, D., "Babbage's Calculating Engine," *Edinburgh Rev.* **59**, 263–327 (1834).

[11]McHugh, J., "Software Diversity: Use of Diversity as a Defense Mechanism," in *Proc. 2005 Workshop on New Security Paradigms*, Lake Arrowhead, CA, pp. 19–20 (2005).

[12]Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C., "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secure Comput.* **1**(1), 11–33 (2004).

[13]Gray, J., *Why Do Computers Stop and What Can Be Done About It?*, Report 85.7, Tandem Computers, Cupertino, CA (1985).

[14]Committee on National Security Systems, *National Information Assurance (IA) Glossary*, CNSS Instruction No. 4009 (26 Apr 2010).

[15]Rogers, P., "Software Fault Tolerance," *Ada User J.* **30**(2), 125–127 (2009).

[16]Avižienis, A., "Design of Fault-Tolerant Computers," in *Proc. 1967 Fall Joint Computer Conf.*, Anaheim, CA, pp. 733–743 (1967).

[17]Lions, J.-L., *ARIANE 5: Flight 501 Failure*, Report by the Inquiry Board, European Space Agency, Paris (July 1996).

[18]Larsen, P., Homescu, A., Brunthaler, S., and Franz, M., "SoK: Automated Software Diversity," in *Proc. 2014 IEEE Symp. on Security and Privacy*, San Jose, CA, pp. 276–291 (2014).

[19]Avižienis, A., and Kelly, J. P. J., "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer* **17**(8), 67–80 (1984).

[20]Knight, J. C., and Leveson, N. G., "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Trans. Softw. Eng.* **SE-12**(1), 96–109 (1986).

[21]Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., et al., "N-Variant Systems: A Secretless Framework for Security through Diversity," in *Proc. 15th USENIX Security Symp.*, Vancouver, pp. 105–120 (2006).

[22]Franz, M., "E Unibus Pluram: Massive-Scale Software Diversity as a Defense Mechanism," in *Proc. 2010 Workshop on New Security Paradigms*, Concord, MA, pp. 7–16 (2010).

[23]Barrantes, E. G., Ackley, D. H., Forrest, S., Palmer, T. S., Stefanovic, D., and Zovi, D. D., "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *Proc. 10th ACM Conf. on Computer and Communications Security*, Singapore, pp. 281–289 (2003).

[24]Kc, G. S., Keromytis, A. D., and Prevelakis, V., "Countering Code-Injection Attacks with Instruction-Set Randomization," in *Proc. 10th ACM Conf. on Computer and Communications Security*, Singapore, pp. 272–280 (2003).

[25]Ammann, P. E, and Knight, J. C., "Data Diversity: An Approach to Software Fault Tolerance," *IEEE Trans. Comput.* **37**(4), 418–425 (1988).

[26]Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., and Franz, M., "Profile-Guided Automated Software Diversity," in *Proc. 2013 IEEE/ACM International Symp. on Code Generation and Optimization*, Shenzhen, China, pp. 1–11 (2013).

[27]Junod, P., Rinaldini, J., Wehrli, J., and Michielin, J., "Obfuscator-LLVM: Software Protection for the Masses," in *Proc. 1st International Workshop on Software Protection*, Florence, Italy, pp. 3–9 (2015).

[28]Collberg, C., Thomborson, C., and Low, D., "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *Proc. 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, San Diego, CA, pp. 184–196 (1998).

[29]László, T., and Kiss, Á., "Obfuscating C++ Programs via Control Flow Flattening," *Ann. Univ. Sci. Budapest. Sect. Comput.* **30**, 3–19 (2009).

[30]Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D., and Su, Z., "Detecting Code Clones in Binary Executables," in *Proc. 18th International Symp. on Software Testing and Analysis*, Chicago, IL, pp. 117–128 (2009).

[31]Le, Q., and Mikolov, T., "Distributed Representations of Sentences and Documents," in *Proc. 31st International Conf. on Machine Learning*, Beijing, China, pp. 1188–1196 (2014).

[32]Lau, J. H., and Baldwin, T., "An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation," in *Proc. 1st Workshop on Representation Learning for NLP*, Berlin, Germany, pp. 78–86 (2016).

[33]Hubert, L., and Arabie, P., "Comparing Partitions," *J. Classification* **2**(1), 193–218 (1985).

[34]Bentley, J., and Floyd, B., "Programming Pearls: A Sample of Brilliance," *Commun. ACM* **30**(9), 754–757 (1987).

[35]Pappas, V., Polychronakis, M., and Keromytis, A. D., "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization," in *Proc. 33rd IEEE Symp. on Security and Privacy*, San Francisco, CA, pp. 601–615 (2012).

[36]Szekeres, L., Payer, M., Wei, T., and Song, D., "SoK: Eternal War in Memory," in *Proc. 31st IEEE Symp. on Security and Privacy*, San Francisco, CA, pp. 48–62 (2013).

[37]Pendleton, M., Garcia-Lebron, R., Cho, J.-H., and Xu, S., "A Survey on Systems Security Metrics," *ACM Comput. Surv.* **49**(4), 62:1–62:35 (2016).

[38]Herlands, W., Hobson, T., and Donovan, P. J., "Effective Entropy: Security-Centric Metric for Memory Randomization Techniques," in *Proc. 7th Workshop on Cyber Security Experimentation and Test*, San Diego (2014).

[39]Williams, D., Hu, W., Davidson, J. W., Hiser, J. D., Knight, J. C., and Nguyen-Tuong, A., "Security Through Diversity: Leveraging Virtual Machine Technology," *IEEE Secur. Priv.* **7**(1), 26–33 (2009).

[40]Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., and Pretschner, A., "Code Obfuscation Against Symbolic Execution Attacks," in *Proc. 32nd Annual Conf. on Computer Security Applications*, Los Angeles, CA, pp. 189–200 (2016).

[41]Banescu, S., Collberg, C., and Pretschner, A., "Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning," in *Proc. 26th USENIX Security Symp.*, Vancouver, Canada, pp. 661–678 (2017).

[42]Sridhar, M., Wartell, R., and Hamlen, K. W., "Hippocratic Binary Instrumentation: First Do No Harm," *Sci. Comput. Program.* **93**(B), 110–124 (2014).

[43]Leroy, X., "Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant," in *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, Charleston, SC, pp. 42–54 (2006).

[44]Geer, D., "Monoculture on the Back of the Envelope," *login* **30**(6), 6–8 (2005).

Joel Coffman, Department of Computer and Cyber Sciences, United States Air Force Academy, Colorado Springs, CO

Joel Coffman is an assistant professor in the Department of Computer and Cyber Sciences at the United States Air Force Academy. He received a B.S. in computer science from Furman University and an M.S. and a Ph.D. in computer science from the University of Virginia. From 2012 to 2018, Joel was a member of the Senior Professional Staff in APL's Asymmetric Operations Sector. In this role, he served as an assistant program manager for the Engineering for Professionals Computer Science, Cybersecurity, and Information Systems Engineering programs and contributed to a variety of sponsored and internally funded research and development projects, including being the co-principal investigator for Proteus, the nexus for the work described in this article. Joel's research interests include the study of automated software diversity, cloud computing security, and keyword search in databases. His e-mail addresses are joel.coffman@jhu.edu and joel.coffman@usafa.edu.

Andrew S. Gearhart, Asymmetric Operations Sector, Johns Hopkins University Applied Physics Laboratory, Laurel, MD

Andrew S. Gearhart is a member of the Senior Professional Staff in APL's Research and Exploratory Development Department. He received a B.S. in computer science and mathematics and a B.A. in psychology from the University of Delaware, and a Ph.D. in computer science from the University of California, Berkeley. He serves as a lecturer for the Engineering for Professionals Computer Science program and served as the co-principal investigator for Proteus, a multiyear investigation of software diversity for cyber defense. Andrew's research interests include the study of software diversity, machine learning, and the application of data analytics to health care applications. His e-mail address is andrew.gearhart@jhuapl.edu.