The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond

BENOIT BAUDRY, INRIA
MARTIN MONPERRUS, Université de Lille

Early experiments with software diversity in the mid 1970s investigated N-version programming and recovery blocks to increase the reliability of embedded systems. Four decades later, the literature about software diversity has expanded in multiple directions: goals (fault tolerance, security, software engineering), means (managed or automated diversity), and analytical studies (quantification of diversity and its impact). Our article contributes to the field of software diversity as the first work that adopts an inclusive vision of the area, with an emphasis on the most recent advances in the field. This survey includes classical work about design and data diversity for fault tolerance, as well as the cybersecurity literature that investigates randomization at different system levels. It broadens this standard scope of diversity to include the study and exploitation of natural diversity and the management of diverse software products. Our survey includes the most recent works, with an emphasis from 2000 to the present. The targeted audience is researchers and practitioners in one of the surveyed fields who miss the big picture of software diversity. Assembling the multiple facets of this fascinating topic sheds a new light on the field.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability; D.4.6 [**Operating Systems**]: Security and Protection; D.3.4 [**Programming Languages**]: Processors; D.2.11 [**Software Engineering**]: Software Architectures

General Terms: Reliability, Security

Additional Key Words and Phrases: Software diversity, program transformation, design principles

ACM Reference Format:
Benoit Baudry and Martin Monperrus. 2015. The multiple facets of software diversity: Recent developments in year 2000 and beyond. ACM Comput. Surv. 48, 1, Article 16 (September 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2807593

16

1. INTRODUCTION

In nature, diversity refers to the fact that many species coexist (among many other definitions). In society, it sometimes refers to the idea of gathering people coming from different cultures and background. In all of these domains, diversity (a fact) is considered essential to the emergence of resilience, stability, or novelty (a property) [McCann 2000]. In software, we take the problem upside-down. We want properties, such as resilience, for which diversity may be the key. The main research question is thus formulated as such: how to create, maintain, exploit—that is, engineer—diversity in software?

This work is partially supported by the EU FP7-ICT-2011-9 No. 600654 DIVERSIFY project.
Authors' addresses: B. Baudry, Campus de Beaulieu, 35042 Rennes, France; email: benoit.baudry@inria.fr; M. Monperrus, Université de Lille 1, Centre de Recherche en Informatique, Signal et Automatique de Lille, 59655 Villeneuve d'Ascq CEDEX, France; email: martin.monperrus@univ-lille1.fr.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2015 ACM 0360-0300/2015/09-ART16 $15.00
DOI: http://dx.doi.org/10.1145/2807593

For instance, early experiments with software diversity in the mid 1970s (e.g., recovery blocks [Randell 1978]) advocate design and implementation diversity as a means for tolerating faults. Indeed, similarly to natural systems, software systems including diverse functions and elements are able to cope with many kinds of unanticipatable problems and failures. Currently, the concept of software diversity appears as a rich and polymorphic notion, with multiple applications. Yet the exploration of this concept is very fragmented over different communities, who do not necessarily know each other.

We aim at putting together the many pieces of the puzzle of software diversity. Previous surveys on classical work about diversity for fault tolerance [Deswarte et al. 1998] or for security [Just and Cornwell 2004] provide important milestones in this direction. However, their scope is very focused on a single type of software diversity, and they do not include the most recent works in the area. Our article contributes to the field of software diversity as the first work that adopts an inclusive vision of the area, with an emphasis on the most recent advances in the field.

Scope. This survey includes classical work about design and data diversity for fault tolerance, as well as the cybersecurity literature that investigates randomization at different system levels. Beyond that, we broaden this standard scope of diversity to include work about the study and exploitation of natural diversity and about the management of diverse software products in software architecture. Since the main barriers between communities are words, we had to cross terminological chasms several times: diversity, randomization, poly- and metamorphism, to only cite a few that are intrinsically related. This inclusive definition allows us to draw a more complete landscape of software diversity than previous surveys [Knight 2011; Schaefer et al. 2012; Just and Cornwell 2004; Deswarte et al. 1998], which we discuss in Section 2.1. For the first time, this survey gathers under the same umbrella works that are often considered very different while sharing a similar underlying concept: software diversity.

Novelty. The field of software diversity has been very active in the 1970s and 1980s for fault-tolerance purposes. There has been a revival in the late 1990s and early 2000s, this time with automatic diversity for security. Both periods have been covered by previous surveys [Deswarte et al. 1998; Just and Cornwell 2004]. The past decade's research on software diversity has also been extremely rich and dynamic. Yet this activity is only partially covered in recent surveys by Schaefer et al. [2012], Knight [2011], and Larsen et al. [2014], which have specific focuses. Our survey includes the most recent works in all areas of software diversity, with an emphasis from 2000 to the present.

Audience. The targeted audience of this article is researchers and practitioners in one of the surveyed fields who miss the big picture of software diversity. Our intention is to let them know and understand the related approaches, so far unknown to them because of the community boundaries. We believe that this shared awareness and understanding, with different technical backgrounds, will be the key enabling factor for the development of integrated and multitier software diversification techniques [Allier et al. 2015]. This will contribute to the construction of future resilient and secure software systems.

Structure. Given the breadth of this work's scope, there is no single decomposition criterion to structure our article. Software diversity has multiple facets: the goal of diversity, the diversification techniques, the scale of diversity, the application domain, when it is applied, and so forth. This diversity of software diversity is reflected in Table I. As shown in Figure 1, we decided to organize this survey mainly along two oppositions. First, we differentiate engineering work that aims at exploiting diversity (Sections 3 and 4) from papers that are more observational in nature, where software

Table I. The Diversity of Software Diversity (Not an Exhaustive Overview)
Over time and over research communities, many kinds of software diversity have been proposed or studied.

Software diversity for . . .	fault tolerance [Randell 1978; Avizienis and Kelly 1984], security [Forrest et al. 1997; Cox et al. 2006], reusability [Pohl et al. 2005], software testing [Chen et al. 2010], performance [Sidiroglou-Douskos et al. 2011], bypassing antivirus software [Borello et al. 2010] . . .
Software diversity at the scale of . . .	networks [O'Donnell and Sethu 2004], operating systems [Koopman and DeVale 1999], components [Gashi et al. 2004], data structures [Ammann and Knight 1988], statements [Schulte et al. 2013], . . .
Software diversity as . . .	a natural phenomenon [Mendez et al. 2013], a goal [Cohen 1993], a means [Collberg et al. 2012], a research object [Knight and Leveson 1986] . . .
Software diversity in . . .	market products [Han et al. 2009], operating systems [Koopman and DeVale 1999], developer expertise [Posnett et al. 2013], . . .
Software diversity when . . .	the specifications are written [Yoo and Seong 2002], the code is developed [Avizienis and Kelly 1984], the application is deployed [Franz 2010], executed [Ammann and Knight 1988] . . .

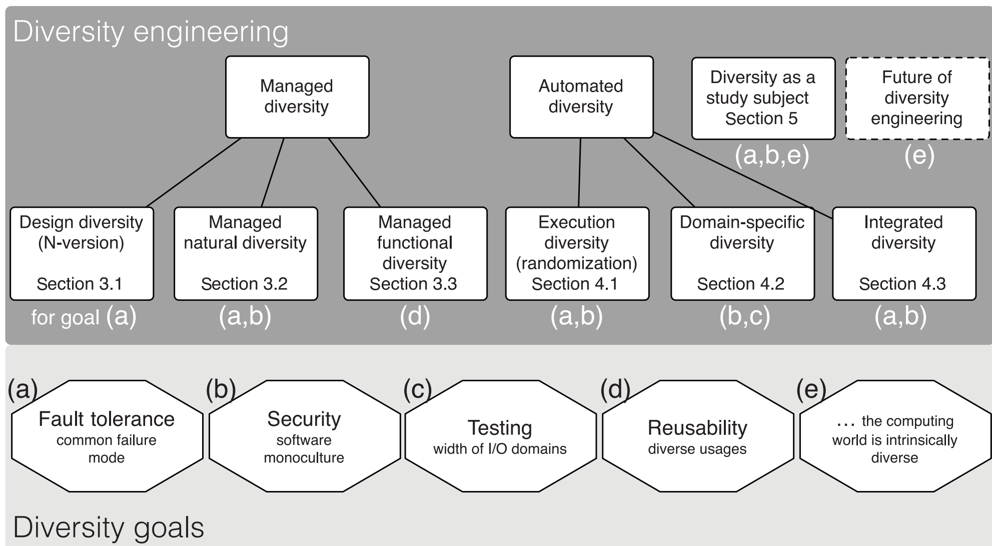

Fig. 1. The diverse dimensions of software diversity.

diversity is a study subject (Section 5.2). Then, we split the engineering papers on *managed diversity* approaches, which aim at manually controlling software diversity (Section 3), and the papers describing *automated diversity* techniques (Section 4). This structuring supports our main goal of bridging different research communities and enables us to discuss, in the same section, papers coming from very different fields. The article can be read linearly. However, each section is meant to be self-contained, and there is a diversity of reading pathways. We invite the reader to use Figure 1 for choosing one's own pathway.

2. SURVEY PROCESS

To prepare this survey, we first analyzed the existing surveys on the topic (Section 2.1). None of them covers the material that we cover. Second, we set up and conducted a systematic process described in Section 2.2.

2.1. Other Surveys on Software Diversity

The oldest survey we found is by Deswarte et al. [1998]. It clearly shows that software diversity has different scales: from the level of human users or operators to the level of hardware and execution. Our survey exactly goes along this line of exploring the diversity of diversities. In addition to classical and 1990s' software diversity, our survey discusses the rich work that has been done around software diversity during the past 15 years: instruction-set randomization, adaptive random testing, and many others.

Littlewood et al. [2001] focus on design diversity (N-version programming). In particular, they review their own work on the probabilistic reasoning that can be made on N-version systems. To this extent, as the abstract puts it, the survey is more a tutorial on design diversity than a broad perspective on software diversity.

The goal of the review paper of Just et al. [2004] is to list the techniques of synthetic diversity that can improve software survivability. "Synthetic diversity" is equivalent, in our view, to "artificial automated diversity." Here, we consider other goals than only security (e.g., quality of service; see Section 4.1.3) and consider other diversity engineering techniques (e.g., managed software diversity; see Section 3).

In a published survey, Knight [2011] discusses four kinds of diversity: classical design diversity (N-version and recovery block), data diversity (a research direction that he has both invented and led), artificial diversity (in the sense of instruction-set randomization for security and the like), and N-variant systems (compared to N-version, N-variant diversity uses artificial and automated diversity). In addition, he introduces the concept of "temporal diversity" as a diversity over time, such as by regularly changing the key for instruction-set randomization. We agree on all points that Knight considers as software diversity. However, we have a broader definition of software diversity: we discuss more kinds of managed software diversity (e.g., software product lines; see 3.3.3), more kinds of artificial diversity (e.g., runtime diversity; see Section 4.1.2), and papers for which diversity is the main study subject (see Section 5).

In 2012, Schaefer et al. coauthored *Software Diversity: State of the Art and Perspectives* [Schaefer et al. 2012]. Despite what the title suggests, this paper surveys only one kind of software diversity: software product lines. As we will discuss later, the techniques of software product lines enable one to manage a set of related features to build diverse products in a specific domain. We refer to this kind of diversity as managed software diversity. In our article, not only do we describe other kinds of managed software diversity such as design diversity, but we also discuss artificial diversity and natural diversity.

Larsen et al. [2014] recently authored a survey about automated software diversity for security and privacy. They discuss the different threat models that can be addressed via diversification. Then, they classify the surveyed approaches according to the nature of the object to be diversified and the temporal dimension of the diversification process. They conclude with an insightful discussion about compiler-based versus binary rewriting diversity synthesis.

2.2. Systematic Process

We followed a systematic process to select the papers discussed in this article. We started with 30 papers that we knew and were written by the most remarkable authors, such as Avizienis, Randell, Forrest, Cohen, Knight and Levenson, and Schaefer. They appear in top publications of these fields (ACM TISSEC, IEEE TSE, IEEE S&P, CCS, ICSE, PLDI, DSN, etc.) and are generally considered as seminal work in each area. Then, we increased this set through a systematic keyword-based search using Google Scholar, IEEE Xplore, and ACM DL. This set went through a second expansion phase when we followed the citation graph of the selected papers. This provided us with a

set of more than 300 papers. Then, we filtered out papers. First, we discarded the redundant papers discussing a similar problem or solution (e.g., we selected only a few papers about product lines or about multiversion execution). Second, we filtered out the papers that had no impact on the literature (that appeared in unknown conferences or had less than five citations after 20 years). Since our survey focuses on recent developments in the field of software diversity, we took special care to keep the most significant recent works (up to papers that appeared in 2014).

3. MANAGED SOFTWARE DIVERSITY

Managed software diversity relates to technical approaches aiming at encouraging or controlling software diversity. This kind of diversity is principally embodied in the work on multiversion software (early structuring of diversity), open software architecture (encouraging diversity), and software product lines (controlling diversity).

3.1. Design Diversity (N-Version)

Since the late 1970s, many different authors have devised engineering methods for software diversification to cope with accidental and deliberate faults. Here, an accidental fault is any form of bug—that is, an internal problem unintentionally introduced by a developer of the execution environment. N-version programming [Avizienis 1985] and recovery blocks [Randell 1978] were the two initial proposals to introduce diversity in computation to limit the impact of bugs. Those techniques are traditionally called *design diversity* techniques.

N-version design is defined as "the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification" [Avizienis and Kelly 1984; Avizienis 1985]. This consists of providing N development teams with the same requirements. Those teams then develop N independent versions, using different technologies, processes, verification techniques, and so forth. The N versions are then run in parallel, and a voting mechanism is executed on the N results. The increased diversity in design, programming languages, and humans is meant to reduce the number of faults by emergence of the best behavior, the emergence resulting from the vote on the output value.

Since the initial definition of the N-version paradigm, it has been refined along different dimensions: the process, the product, and the environment necessary for N-version development [Avizienis 1995]. For example, Kelly et al. [1991] distinguish between random diversity (let independent teams develop their version) from enforced diversity, in which there is an explicit effort to design diverse algorithms or data structures. More recently, Avizienis [2000] proposed to adapt the concept to software survivability.

Recovery blocks were developed at the same time as N-version design and proposed a way of structuring the code, using diverse alternative software solutions, for fault tolerance [Randell 1978]. The idea was to have recovery blocks in the program—that is, blocks equipped with error detection mechanisms and one or more spares that are executed in case of errors. These spares are diverse variant implementations of the function.

In the latest work about N-version development, both N-version design and recovery blocks were included in the same global framework [Avizienis 1995]. This framework has been used in multiple domains, including the design of multiple versions of firewalls [Liu and Gouda 2008]. Whereas the essential conceptual elements of design diversity have remained stable over time, most subsequent works have focused on experimenting and quantifying the effects of this approach on fault tolerance. The work related to the analysis of N-version programming is synthesized in Section 5.1.

3.2. Managed Natural Software Diversity

We call *natural diversity* the existence of different software solutions that provide similar functionalities and spontaneously emerge from software development processes. There exist several forms of natural software diversity. For example, the programs that can be customized through several parameters embed a natural mechanism for diversification (two instances of the same program tuned with different parameters can have different behaviors in terms of performance). Software market and competition are also strong vectors that drive the natural emergence for software diversity. For example, the gigantic business opportunities offered by the World Wide Web has driven the emergence of many competing Web browsers. Web browsers are diverse in their implementation, in their performance, and in some of their plugins, yet they are functionally very similar and can be used for one another in most cases. Other examples of such market software diversity include operating systems, firewalls, database management systems, virtual machines, routers, middleware, application servers, and so forth. In this section, we present a set of works that exploit this natural diversity for different purposes. We will come back to natural diversity later in Section 5.2, discussing authors who study natural diversity with no engineering goals at all.

Hiltunen et al. [2000] propose the Cactus mechanism for survivability—that is, a mechanism that monitors and controls a running application to tolerate unpredictable events such as bugs or attacks. The Cactus approach relies on fine-grain customization of the different components in the application, as well as runtime adaptation, to achieve survivability. They discuss how they can switch between different security and fault-tolerance solutions through customization and also discuss how this natural way of changing a system supports the emergence of natural diversity and thus increases resilience.

Caballero et al. [2008] exploit the existing diversity in router technology to design a network topology that has a diverse routing infrastructure. Their work introduces a novel metric to quantify the robustness of a network. Then, they use it to compare the robustness of different, more or less diverse, routing infrastructure. They explore the impact of different levels of diversity by converting the problem into a graph coloring problem. They show that a small amount of router technology and well-designed topology actually increases the global robustness of the infrastructure.

Totel et al. [2006] propose designing an intrusion detection mechanism by design diversity, leveraging the natural diversity of components-off-the-shelf (COTS). They exploit the fact that COTS for database management and Web servers have very few common mode failures [Wang et al. 2003; Gashi et al. 2004] and are thus very good candidates for N-version design based on natural diversity. The authors deploy an architecture with three diverse servers running on three different operating systems and feed it with the requests sent on their campus Web page in the past month (800,000 requests, out of which around 1% can be harmful). The results show that the COTS-based IDS only raises a small number of false positives. Along the same line, Garcia et al. [2014] conducted a study on the impact of operating system diversity with respect to security bugs of the NIST National Vulnerability Database (NVD). Their results show that diversity indeed contributes to building intrusion-tolerant systems.

Oberheide et al. [2008] exploit the diversity of antivirus and malware systems to propose what is called *N-version protection*. It is based on multiple and diverse detection engines running in parallel. Their prototype system intercepts suspicious files on a host machine and sends them to the cloud to check for viruses and malware against diverse antivirus systems. They evaluate their system over 7,220 malwares and show that it is able to detect 98% of the malware. It provides better results than a single antivirus in 35% of the cases. The idea has been further investigated by Bishop et al. [2011], who

explored the deep characteristics of the dataset of known malware to reduce global vulnerability.

O'Donnell and Sethu [2004] leverage the diversity of software packages in operating systems and investigate several algorithms to increase the global diversity in a network of machines. They model the diversification of distributed machines as a graph coloring problem and compare different algorithms according to their ability to set a network that is tolerant to attacks. The experiments are based on a simulation that uses the topology from email traffic at the authors' institution. They show that the introduction of diversity at multiple levels provides the best defense.

Carzaniga et al. [2010] find multiple different sequences of method calls in Javascript code, which happen to have the same behavior. They harness this redundancy to set up a runtime recovery mechanism for Web applications.

Gorbenko et al. [2011] propose an intrusion avoidance architecture based on multilevel software diversity and dynamic software reconfiguration in IaaS cloud layers. The approach leverages the natural diversity of COTS that are found in the cloud (operating system, Web server, database management system, and application server) in combination with dynamic reconfiguration strategies. The authors illustrate the approach with an experiment over several weeks, during which they switch between four diverse operating systems that have different open vulnerabilities. They discuss how this mechanism reduces exposure to vulnerabilities.

Summary. In this section, we have focused on techniques that exploit the natural diversity that can be found among COTS or even as redundancy in programs. All of these works identify different forms of natural diversity and demonstrate how it can be harnessed to address fault-tolerance or security issues.

3.3. Managed Functional Diversity

In software, it is known that many functions are the same yet different. For instance, passing a message to a distant machine or writing to a local file is conceptually the same: writing data to a location. However, the different implementations (e.g., for network or for file input/output) of this abstract function are radically different. One responsibility of software abstraction is to capture this conceptual identity and to abstract over the diversity of implementation details. The goals of this abstraction are reuse, modularity, maintainability, extensibility, and so forth.

For instance, Unix is well known because its concept of file captures all input/output operations, whether on the network, on a physical file on disk, or on the memory of a kernel module. We refer to this facet of abstraction as managing the functional diversity.

Many software abstractions have the clear goal of managing functional diversity. In the following, we will review classical object-oriented software, plugin-based software architectures, and techniques related to the software product line research field.

3.3.1. Class Diversity. The object-oriented software paradigm is rich in implications (on understandability, reuse, etc.). There is one point in this paradigm related to managing the diversity: polymorphism.

Polymorphism is the mechanism enabling us to have code that calls other pieces of code in a nonpredefined manner. The late binding between functions enables an object to call a diverse set of functions and even to call code that will be written in the future. To this extent, polymorphism is the key mechanism enabling the management of the function diversity (as embodied in classes). In other words, polymorphism (with abstract methods, interfaces, or other fancy object-oriented constructs) supports the construction of a program architecture that is ready for handling diversity.

As Meyer [1988] puts it:

> "We are at the heart of the object-oriented method's contribution to reusability: offering not just frozen components (such as found in subroutine libraries), but flexible solutions that provide the basic schemes and can be adapted to suit the needs of many diverse applications."

3.3.2. Diversity through Plugin-Based Software Architecture. Plugin-based software architectures offer the means to design open software systems. Plugins are software units that encapsulate a given functionality and some information about its dependencies. As far as we know, Wijnstra [2000] was one of the first authors to assess the suitability of plugins to handle the diversity of configurations and usages of a complex software system. He proposed using plugins together with a component framework to design an extensible system for medical imaging. In this context, he needed to have a core set of functionalities to deploy a diversity of products that fit different requirements or different environments.

More recently, very successful software projects such as Wordpress, Firefox, and Eclipse have adopted plugin-based architectures. This allows them to be open, thus leveraging the efforts of large open source communities, while keeping a core set of functionalities across all versions. But most importantly, this architecture supports a true explosion of functional software diversity. For example, there are 25,000 plugins available for Wordpress that can be combined by users in billions of functionally diverse configurations, each of them fitting a specific purpose or need. This was somehow predicted by Van Ommering [2002], who used a plugin-based architecture in which connections between plugins handle design-time or runtime diversity.

3.3.3. Software Product Lines. The techniques around software product lines can be considered as the means of controlling a diversity of software solutions capable of handling a diversity of requirements (user requirements or environmental constraints) [Pohl et al. 2005; Clements and Northrop 2002]. Software product line engineering is about the development of "a diversity of software products and software-intensive systems at lower costs, in shorter time, and with higher quality" [Pohl et al. 2005]. This consists of building an explicit variability model that captures all commonalities and variation points in requirements and software solutions. In other words, the variability model is an explicit definition of the space of diverse solutions that can be engineered in a particular domain. For example, this model can be expressed in the form of a feature model [Kang et al. 1990] or a decision model [Schmid et al. 2011].

In the context of software product lines, the main challenge for software diversity management consists of providing systematic ways to reuse existing parts of software systems to derive diverse solutions.

We synthesize the main works in software product lines; for an exhaustive survey, we refer the reader to Schaefer et al. [2012]. We start by looking at solutions that handle diversity in design, then we summarize solutions for diversity in implementation.

Software product lines mainly offer support for design diversity through architectural solutions [Clements and Northrop 2002]. An essential challenge is to handle both the logical variability (the set of features that architects manipulate) and the variability of concrete assets (diversity of software pieces that can actually be composed to implement a particular product). Initial solutions are based on annotations to relate both views [Atkinson 2002]. Hendrickson and van der Hoek [2007] propose a product line architecture modeling approach that unites the two, using change sets to cluster related architectural differences. Several approaches are founded on a compositional approach to derive products from architectural models. Ziadi et al. [2004] propose sound composition operations for UML 2.0 scenarios to automatically synthesize diverse statecharts inside a given product line, whereas Morin et al. [2008] compose software components

to derive software configurations at runtime. Other approaches rely on an orthogonal variability model associated with model transformations for product derivation, as is the case for the common variability language [Haugen et al. 2008] and the orthogonal variability model [Pohl et al. 2005], which are annotative variability modeling approaches, such as preprocessor variability. At the boundary between models and implementation, it is possible to capture the variants of a program with explicit design patterns, as suggested by Jézéquel [1998]. At the source code level, several mechanisms exist to manage a set of variants for a given program: feature-oriented programming [Prehofer 1997] proposes a flexible model for object composition; delta-oriented programming [Schaefer et al. 2010] instantiates the concept of delta modeling [Clarke et al. 2011] to specify a specific set of deltas for a program, as well as transformations that can systematically inject a set of selected deltas in a program to derive a variant; Figueiredo et al. [2008] have reported on the usage of aspect-oriented programming to handle variants in a product line and discuss the positive and negative effects on design stability; and preprocessing was one of the first language technologies used to handle program variants and has been extensively analyzed, such as in the recent work by Liebig et al. [2010].

3.3.4. Discussion. The main benefit of software construction paradigms with respect to diversity is reusability: a large range of diverse products can be made with a smaller number of software "bricks." This is our motivation for considering software construction and design paradigms in our survey.

However, the overall effect of those paradigms is to reduce software design diversity for a given set of product functions. Indeed, those reuse-oriented paradigms create a tension between reusability and monoculture [Allier et al. 2015]. Both relate to diversity (the second one in a dual manner). In practice, there is an engineering trade-off between the increase in diversity due to the very large number of possible combinations and the decrease in diversity due to massive reuse.

3.4. Summary

This section has focused on three areas of software engineering that *manage* software diversity. The first area presented was multiversion design, an approach to fault-tolerance that aims at managing the manual development of diverse program versions. The second area explored managing and exploiting software diversity that naturally emerges in software markets or open source communities to build fault- or attack-tolerant systems. The third area opened on a series of works dedicated to the management of functional diversity to fulfill the various usages of a given system. These three parts refer to different research communities, yet they all share a common approach: software diversity can be managed and harnessed to achieve specific software engineering objectives.

4. AUTOMATED SOFTWARE DIVERSITY

Automated software diversity consists of techniques for artificially and automatically synthesizing diversity in software. Instead of using the adjective *automated*, some authors call it *synthetic diversity* [Just and Cornwell 2004] or *artificial diversity* (e.g., Locasto et al. [2006]). However, *artificial* literally means "created or caused by people."[1] To this extent, N-version programming also produces artificial diversity, but the diverse program variants are produced manually. We prefer *automated diversity*, which emphasizes the absence of humans in the loop and is in clear opposition to managed software diversity. Beyond those details, we actually equate those three terms: artificial, synthetic, and automated diversity.

[1]Merriam-Webster: http://www.merriam-webster.com/dictionary/artificial.

Automated software diversity is valuable in different contexts, such as software security or fault tolerance. However, these different *goals* are not the only dimension in which we can characterize the various approaches to automated software diversity. First, the *scale* dimension characterizes the fact that software systems are engineered at several scales, from a set of interacting machines in a distributed system down to the optimization of a particular loop. Research has produced techniques for automated software diversity along all of those different scales. Second, the *genericity* dimension explores whether the diversification technique is domain specific or not. Third, the *integrated* dimension is about the assembly of multiple diversification techniques in a global approach.

In the following, we choose to present the literature on automated software diversity along three axes. We first present the wide range of randomization techniques (either static or dynamic) in Section 4.1. To achieve automated diversity, many authors have exploited specificities of the application domain or the technology used, and this is presented in Section 4.2. Finally, Section 3 discusses blending different kinds of software diversity together—what we call *integrated software diversity*.

4.1. Randomization

The mainstream software paradigms are built on determinism. All layers of the software stack tend to be deterministic, from programming language constructs, to compilers, to middleware, up to application-level code.

However, it is known that randomization can be useful, such as to improve security [Bhatkar et al. 2003]. A classical example of randomization is compiler-based randomization: a compiler may compile the same code with different memory layouts to decrease the risk of code injection.

What is the relation between randomization and diversity? A randomization technique creates—directly or indirectly—a set of unique executions for the very same program. As mentioned by Bhatkar et al. [2003], "the use of randomized program transformations [is] a way to introduce diversity into applications." The notion of diversity of execution is broad: it may mean diverse performances, diverse outputs, diverse memory locations, and so forth. We present an overview of diversifying randomization techniques in this survey. For a more detailed survey about randomization, we refer the reader to surveys dedicated to that topic, particularly the one of Keromytis and Prevelakis [2005].

There are different kinds of diversifying randomization. First, one can create different versions of the same program. For instance, one can randomize the data structures at the source or at the binary level. We call this kind of randomization *static*. Static randomization is discussed in Section 4.1.1.

Second, one can automatically integrate randomization points in the executable program. For instance, a malloc primitive (memory allocation) with random padding is a randomization point: each execution of malloc yields a different result. Contrary to static randomization, there is still one single version of the executable program, but their executions are diverse. We call this kind of randomization *dynamic randomization* (also called *runtime randomization* [Xu et al. 2003]) and discuss it in Section 4.1.2.

Third, some randomization techniques do not aim at providing a strict behavioral equivalence between the the original program and the randomized executions. They are discussed in Section 4.1.3.

Finally, as we will see later in Section 4.3, diversification techniques can be stacked. This also holds for randomization: one can stack static and dynamic randomization. In this case, there are diverse versions of the same program that embed randomization points which they themselves produce different executions.

4.1.1. Static Randomization. One of seminal papers on static randomization is by Forrest et al. [1997], who highlight two families of randomization: randomly adding or deleting nonfunctional code and reordering code. Those transformations are also described by Cohen [1993] in the context of operating system protection. Lin et al. [2009] randomize the data structure of C code. Following the line of thought of Forrest et al. [1997], they reorder fields of data structures (`struct` and `class` in C/C++ code) and insert garbage ones.

The concept of instruction-set randomization was invented in 2003 by two independent teams [Kc et al. 2003; Barrantes et al. 2003]. It consists of creating a unique mapping between artificial CPU instructions and real ones. This mapping is encoded in a key that must be known at runtime to actually execute the program. Eventually, the instruction set of a machine can be considered as unique, and it is very hard for an attacker ignoring the key to inject executable code. Instruction-set randomization can be done statically (a variant of the program using a generated instruction set is written somewhere) or dynamically (the artificial instruction set is synthesized at load time). In both cases, instruction-set randomization indeed creates a diversity of execution that is the essence of the countermeasure against code injection.

In some execution environments (e.g., x86 CPUs), there exists a "no operation" (NOP) instruction, which was invented for the sake of optimization to align instructions with respect to some alignment criteria (e.g., memory or cache). Merckxt [2006] and later Jackson [2012] explored how to use NOP to statically diversify programs. The intuition is simple: by construction NOP does nothing, and the insertion of any amount of it results in a semantically equivalent program. However, it breaks the predictability of program execution and to this extent mitigates certain exploits.

Banescu et al. [2015] exploit software diversity, along with white-box cryptography against changeware. Changeware software modifies resources of software applications (e.g., configuration files). Browser hijacking malware is one popular example that aims at changing Web browser settings such as the default search engine or the home page. They demonstrate the effectiveness of the solution against different kinds of attacks for changeware.

Obfuscation is a classical application domain of static randomization. Code obfuscation consists of modifying software for the sake of hindering reverse engineering and code tampering. Its main goal is to protect intellectual property and business secrets. A basic obfuscation technique simply transforms a program P into a program P' that is distributed. However, since obfuscation is automated, it is often possible to generate several different obfuscated versions of the same program (e.g., as proposed by Collberg et al. [2012]). To this extent, code obfuscation is one kind of software diversification, with one specific criterion in mind. For an overview on code obfuscation, we refer the reader to the now classical taxonomy of Collberg et al. [1997]. For an example of a concrete obfuscation engine for Java programs, we refer to Collberg et al. [1998, Fig. 1]. When obfuscation happens at runtime, it is a kind of execution diversity, which we discuss in Section 4.1.2.

4.1.2. Dynamic Randomization. Chew and Song [2002] target operating system randomization. More specifically, they randomize the interface between the operating system and the user-land applications: the system call numbers, the library entry points (memory addresses), and the stack placement. All of those techniques are dynamic, done at runtime using load time preprocessing and rewriting.

Dynamic randomization can address different kinds of problems. In particular, it mitigates a large range of memory error exploits. Bhatkar et al. [2003, 2005] have proposed some of the seminal research in this direction. Their approach is based on three kinds of randomization transformations: randomizing the base addresses of

applications and libraries' memory regions, random permutation of the order of variables and routines, and the random introduction of random gaps between objects.

Static randomization creates a diverse version of the same program at compilation time, and dynamic randomization creates diverse executions of the same program under the same input at runtime. What about just-in-time compilation randomization? This point has been studied by Homescu et al. [2013] at the University of California at Irvine. Their approach neither creates diverse versions of the same program nor introduces randomization points: the randomization happens in the just-in-time compiler directly. Their randomization is based on two diversification techniques: insertion of NOP instructions and constant blinding.

In the techniques that we have just discussed, the support for dynamic randomization is implemented within the execution environment. On the contrary, self-modifying programs embed their own randomization techniques [Mavrogiannopoulos et al. 2011]. This is done for the sake of security and is considered one of the strongest obfuscation mechanisms [Mavrogiannopoulos et al. 2011].

The data diversity of Ammann and Knight [1988] represents another family of randomization. The goal of data diversity is not security but fault tolerance. The technique aims at enabling the computation of a program in the presence of failures. The idea of data diversity is that when a failure occurs, the input data is changed so that the new input does not result in a failure. The output based on this artificial input, through a inverse transformation, remains acceptable in the domain under consideration. To this extent, this technique dynamically diversifies the input data.

The notion of environment diversity [Vaidyanathan and Trivedi 2005] refers to techniques that change the environment to overcome failures. For instance, changing the scheduler or its parameter is indeed a change in the environment. This is larger in scope than just changing some process data, such as standard randomization.

4.1.3. Unsound Randomization. Traditional randomization techniques are meant to produce programs or executions that are semantically equivalent to the original program or execution. However, researchers have explored the domain of "unsound" randomization techniques, either statically or dynamically.

Foster and Somayaji [2010] recombine binary object files of commodity applications. If an application is made of two binary files A and B, they show that is possible to run the application by artificially linking a version of A with a different yet close version of B. The technique enables them to tolerate bugs and even let new functions emerge but has no guarantee on the behavior of the recombination.

Schulte et al. [2013] describe a property of software that has not been reported previously. Software can be mutated, and at the same time it can preserve a certain level of correctness. Using an analogy from genomics, they call this property *software mutational robustness*. This property has a direct relation to diversification: one can mutate the code to get functionally equivalent variants of a program. Doing this in advance is called *proactive diversity*. The authors present a set of experiments showing that this proactive diversity is able to fix certain bugs.

In our previous work [Baudry et al. 2014], we experimented with different transformation strategies, on Java statements, to synthesize "sosie" programs. The sosies of a program P are variants of P (i.e., different source code), which pass the same test suite and exhibit a form of computation diversity. In other words, our technique synthesizes large quantities of variants, which provide the same functionality as the original through a different control or data flow, reducing the predictability of the program's computation.

Another kind of runtime diversity emerges from the technique of loop perforation; Sidiroglou-Douskos et al. [2011] have shown that in some domains, it is possible to skip

the execution of loop iterations. For instance, in a video decoding algorithm (codec), skipping some loop iterations has an effect on some pixels or contours but does not further degrade or crash the software application. On the other hand, skipping loop iterations is key with respect to performance. In other words, there is a trade-off between the performance and accuracy. This trade-off can be set offline (e.g., by arbitrarily skipping one every two loops) or dynamically based on the current load of the machine. In both cases, this kind of technique results in a semantic diversity of execution profiles and consequently is deeply related to automated diversity.

4.1.4. Summary. In this section, we have focused on techniques that automatically randomize some aspects of a program, either statically (Section 4.1.1) or dynamically (Section 4.1.2) . Diversity occurs in memory, in the operating system, in the bytecode, or in the source code, but in all cases it happens with no human intervention through random processes. The most audacious randomization techniques are sometimes considered unsound (Section 4.1.3), as they dare to change the execution semantics of the program being diversified.

4.2. Domain-Specific Diversity

The techniques presented so far are independent of any application domain. Yet domain knowledge can be essential to devise efficient diversification techniques. This section illustrates such situations.

For instance, a common vulnerability of Web applications is the possibility of injecting SQL code to access unauthorized data or corrupt existing data. Boyd and Keromytis [2004] proposed a technique to diversify the SQL query themselves. By simply prefixing all SQL keywords with an execution specific token, they create an unpredictable language that is hardly attackable from the outside and diverse for each database.

Feldt [1998] exploited the structure of the genetic programming problem domain for the sake of diversification. He uses a genetic programming system to create a pool of diverse airplane arrestment controllers. He then shows that the failure modes of the synthesized programs are diverse—in other words, that the approach is effective for the generation of a kind of failure diversity.

Oh et al. [2002] presented a program transformation aiming at detecting a particular hardware fault (stuck-at faults in data paths of functional units). The transformation consists of multiplying all numerical computations by a constant k in a semantics-preserving way. The authors show that this technique is effective with respect to their fault model. Obviously, it enables one to automatically obtain diverse implementations of the same program (for different values of k).

Computer viruses are programs whose main opponents are antivirus systems. Inventors of computer viruses obviously care about being reverse engineered. However, more importantly for them, the computer viruses must remain undetectable as long as possible. Diversification is one solution in this very specific domain: if the virus exists under many different forms, it is harder for antivirus systems to detect them all. From the perspective of the virus itself, it is even better to constantly change itself. This kind of diversification is performed through so-called metamorphic engines, where metamorphism refers to the concept of having different forms for the same identity. For a recent account on this kind of diversification, we refer the reader to Borello et al. [2010].

In the domain of sensor networks, Alarifi and Du [2006] propose an approach to diversifying sensor software to mitigate reverse-engineering effort. Their approach diversifies both the data (e.g., the keys used to communicate between nodes) and the code. As a result, each node in a sensor network is likely to be unique.

So far, we have discussed the diversification of software applications. Test cases are executable programs, but very specific ones. Although they are often written in general-purpose programming languages, their unique goal is to verify the correctness of an application. They do not provide services to users. Interestingly, this fundamental difference does not prevent diversity and diversification and thus is valuable in test cases as well. Adaptive random testing [Chen et al. 2010] is a random testing technique whose goal is generate input test data. It is adaptive in the sense that the generated test cases depend on the previously generated ones. The final goal is to evenly spread test cases throughout the input domain. To this extent, adaptive random testing aims at generating diverse test cases, and this is clear for the authors themselves, who subtitled their flagship paper *The Art of Test Case Diversity*. The VAT model of Feldt et al. [2008] is an example of adaptive random testing. They use an information distance for information theory to maximize the diversity of generated test cases.

These applications of automated software diversity illustrate the wide applicability of the concept of software diversity. Software diversity is not specific to a paradigm, a platform, a language, or an application domain, and it appears that it can be applied to any software object. The concept seems to be as general as that of computing.

4.3. Integrated Diversity

Integrated software diversity is about works that aim at automatically injecting different forms of diversity at the same time in the same program. In this line of thought, previous researchers have either emphasized the fact that the diversity is stacked (Section 4.3.1) or whether these different forms of diversity are managed with a specific diversity controller (Section 4.3.2).

4.3.1. Stacked Diversity. The different contributions discussed in this section all share the same intuition that each kind of artificial diversity has value in one perspective (a specific kind of attack or bug), and thus integrating several forms of diversity should increase the global ability of the software system with respect to security or fault tolerance.

Wang et al. [2001] propose a multilevel program transformation that aims at introducing diversity at multiple levels in the control flow so as to provide in-depth obfuscation. This work on program transformation takes place in the context of a software architecture for survivable systems as proposed by Knight et al. [2000]. The architecture of Wang et al. relies on probing mechanisms that integrate two forms of diversity: in time (the probe algorithms are replaced regularly) and in space (there are different probing algorithms running on the different nodes of the distributed system).

Bhatkar et al. [2003] aim at developing a technique for address obfuscation to thwart code injection attacks. This obfuscation approach relies on the combination of several randomization transformations: randomize base addresses of memory regions to make the address of objects unpredictable, permute the order of variables in the stack, and introduce random gaps in the memory layout. Considering that all of these transformations have a random component, they synthesize different outputs on different machines, thus increasing the diversity of attack surfaces that are visible to attackers.

In a report of the DARPA project Self-Regenerative System (SRS), Knight et al. [2007] summarize the main features of the Genesis Diversity Toolkit. This tool is one of the most recent approaches that integrates multiple forms of artificial diversity. The goal of the project was to generate 100 diverse versions of a program that were functionally equivalent but for which a maximum of 33 versions had the same deficiency. The tool supports the injection of five forms of diversity: address space randomization (ASR), stack space randomization (SSR), simple execution randomization (SER), strong instruction set randomization (SISR), and calling sequence diversity (CSD).

The GENESIS project, also coordinated by Knight's group, explored a complete program compilation chain that applies diversity transformations at different steps to break the monoculture [Williams et al. 2009]. Diversity transformations are applied at compile time, link time, load time, and runtime. The latter step is the main innovation of GENESIS and relies on the Strata virtual machine technology, which supports the injection of runtime software diversity. This application-level virtual machine realizes two forms of diversification: CSD and instruction set diversity.

Jacob et al. [2008] propose superdiversification as a technique that integrates several forms of diversification to synthesize individualized versions of programs. The approach, inspired by compilation superoptimization, consists of selecting sequences of bytecode and in synthesizing new sequences that are functionally equivalent. Given the very large number of potential candidate sequences, the authors discuss several strategies to reduce the search space, including learning occurrence frequencies of certain sequences.

Franz [2010] advocates for massive-scale diversity as a new paradigm for software security. The idea is that today some programs are distributed several million times, and all of these software clones run on millions of machines in the world. The essential issue is that even if takes a long time to an attacker to discover a way to exploit a vulnerability, this time is worth spending because the exploit can be reused to attack millions of machines. Franz envisions a new context in which each time a binary program is shipped, it is automatically diversified and individualized to prevent large-scale reuse of exploits. The approach relies on four paradigm shifts as enablers for his vision: online software distribution, ultra reliable compilers, cloud computing, and good enough performance.

In 2010, Moving Target Defense (MTD) was announced by the President's Cyberspace Policy Review as one of the three "game-changing" themes to cybersecurity. The software component of MTD integrates spatial and temporal software diversity to "limit the exposure of vulnerabilities and opportunities for attack" [Jajodia et al. 2011]. With such a statement, future solutions for MTD will heavily rely on the integration of various software diversity mechanisms to achieve their objectives.

Inspired by the work of Cohen [1993], who suggested multiple kinds of program transformations to diversify software, Collberg et al. [2012] compose multiple forms of diversity and code replacement in a distributed system to protect it from remote man-at-the-end attacks. The diversification transformations used in this work are adapted from obfuscation techniques: flatten the control flow, merge or split functions, non-functional code addition, parameter reordering, and variable encoding. These transformations for spatial diversity are combined with temporal diversity (when and how frequently diversity is injected), which rely on a diversity scheduler that regularly produces new variants.

Allier et al. [2015] recently proposed using software diversification in multiple components of Web applications. They combine different software diversification strategies, from the deployment of different vendor solutions to fine-grained code transformations, to provide different forms of protection. Their form of multitier software diversity is a kind of integrated diversity in application-level code.

4.3.2. Controllers of Automated Diversity. If mixed together and put at a certain scale of automation and size, all kinds of automated diversity need to be controlled. Popov et al. [2012] provide an in-depth analysis of diversity controllers, showing that diversity controlled with specific diversity management decisions is better than naive diversity maximization. On the engineering side, several researchers have discussed how to manage the diverse variants of the same program.

Cox et al. [2006] introduce the idea of N-variant systems, which consists of automatically generating variants of a given program and then running them in parallel to detect security issues. This is different from N-version programming because the variants are generated automatically and not written manually. The approach is integrated because it synthesizes variants using two different techniques: address space partitioning and instruction set tagging. Both techniques are complementary: address space partitioning protects against attacks that rely on absolute memory addresses, whereas instruction set tagging is effective against the injection of malicious instructions. In subsequent work, the same group proposed another transformation that aims at thwarting user ID corruption attacks [Nguyen-Tuong et al. 2008].

Salamat et al. [2008] and Jackson et al. [2011] find a nice name for this concept: *multivariant execution environment*. This type of environment provides support for running multiple diverse versions of the same program in parallel. The diverse versions are automatically synthesized at compile time with reverse stack execution [Salamat et al. 2009, 2011]. The execution differences allow some kind of analysis and reasoning on the program behavior. For instance, multivariant execution enables Salamat et al. [2008] to detect malicious code that is trying to manipulate the stack.

Locasto et al. [2006] introduce the idea of collaborative application communities. The same application (e.g., a Web server) is run on different nodes. In the presence of bugs (invalid memory accesses), each node tries a different runtime fix alternative. If the fix proves to be successful, a controller shares it among other nodes. This healing process contains both a diversification phase (at the level of nodes) and a convergence phase (at the level of the community).

4.3.3. Summary. Each form of software diversification targets a specific goal (e.g., against a specific attack vector). Many recent works have thus experimented with the integration of multiple forms of diversity in a system and have benefit from several forms of protection. We have discussed these works here, as well as the specific kinds of controllers that are required to integrate various diversification techniques.

4.4. Summary

This section has presented a broad range of contributions on automated software diversity. They come from different research communities, and some of them do not even use the word *diversity*. However, they all share the same idea that programs and program executions need not be identical. With respect to the rest of this article, they are fully automated, which is different from the natural diversity discussed in Sections 3.2 and 5.2 and the managed yet mostly manual diversity presented in Section 3.

5. DIVERSITY AS STUDY SUBJECT

In this section, we present different works that focus on analyzing and quantifying software diversity and its effects on different aspects of reliability (e.g., fault tolerance or intrusion avoidance). Contrary to the previous sections, the work presented here is not primarily an engineering contribution, and it is not a new technique to support, encourage, or create a new kind of software diversity. These approaches all have in common that they consider software diversity as their research subject per se. They simply aim at understanding the deep nature of software diversity from the causes to the implications.

First, Section 5.1 discusses the theoretical models of design diversity and its effects on fault tolerance. Then, Section 5.2 presents the literature on the analysis of the natural diversity that is found in COTS and source code.

5.1. Theoretical Modeling of Design Diversity

Failure independence is a critical assumption of the design diversity principle for fault-tolerant critical systems. After the introduction of N-version programming and recovery blocks in the late 1970s, a large number of studies investigated their theoretical foundations and the validity of their assumptions. We discuss the most important studies here.

Design diversity (N-version programming, recovery blocks) was one of the earliest proposals to leverage diversity and redundancy in software for the sake of fault tolerance. Fault tolerance is ensured under one essential assumption: the independence of failures among the diverse solutions. Because of the critical impact of this assumption, a large number of papers have investigated the validity of this assumption. Whereas Section 3.1 focused on the principles of design diversity, here we focus on the studies that have evaluated the impact of this approach through empirical studies and statistical modeling.

Knight and Leveson [1986] provide the first large-scale experiment that aimed at validating the independence assumption in N-version programming. They asked students to write a program from a single requirements document (for a simple antimissile system) and obtained 27 programs. Each program was tested against 1 million random test cases. The quality of the programs was very high (very few faults), but still there were errors found in more than one version (the same error in independently developed programs). A statistical analysis of the results revealed a significant lack of independence between certain errors in the multiple versions of this program. Consequently, the paper was the first major criticism of the effectiveness of design diversity.

Bishop et al. [1986] summarize the results of the PODS project, which aimed at evaluating N-version design on the reliability of software. Their experimental setup is based on the development of three versions of a controller for overpower protection. The requirements document is the same for the three teams, but then they use different methods and languages for the implementation. They concluded that running the three versions, with a voting mechanism, produces a system that is more reliable than the most reliable version and also that back-to-back testing on all three versions is an effective solution to find residual bugs.

Several pieces of work propose theoretical frameworks to analyze and quantify the effects of N-version design on reliability. Eckhardt and Lee [1985] develop a theoretical statistical model for evaluating the impact of diversity on fault tolerance. This model quantifies the effect of joint occurrences of errors on the reliability of the global system. Then, they use this model to explore the conditions under which N-version design can improve fault tolerance and the limits of coincidental errors on the effect of N-version design. Littlewood et al. refine the work of Eckhardt, first by considering the diversity of development methods [Littlewood and Miller 1989] and more recently by adding further hypotheses and studying two-channel systems [Littlewood and Rushby 2012]. They show that methodological diversity, analyzed as the diversity of development decisions, is very likely to produce behavioral diversity. Popov and Strigini [2001] propose another model to analyze the effects of design diversity in which they rely on data that are more related to physical attributes than previous proposals, making the model more actionable for reliability analysis and prediction. Mitra et al. [1999] define metrics to quantify diversity in N-version designs and highlight new results about the effectiveness of N versions on software reliability: diversity increases fault tolerance in the presence of common mode failures, as well as self-testing capacities, but the effects of diversity decrease over time. Nicola and Goyal [1990] propose a statistical model that captures the distribution of correlated failures in multiple versions, as well as a combinatorial formula to predict the reliability of a system running N versions. They analyze the effectiveness of N-version design and demonstrate the need for loose

correlations between failures in the N versions. Hatton [1997] evaluates N-version design slightly differently: he proposes a theoretical model to compare the development of a single highly reliable version of a software component versus the development of N versions of the component. He concludes that N-version design is good, especially considering our inability to make a really good version.

Kanoun [1999] performs a cost analysis of developing two diverse versions of the same program. She aims at providing feedback about the overhead of developing the second version, considering one version as the reference. She focuses on working hours records for cost estimates. She observes between 25% and 134% overhead depending on the development phase (the highest overhead is for the coding and unit tests, whereas the lowest is for functional specification). These results confirm other observations from controlled experiments, with actual data from industrial software development.

Partridge and Krzanowski [1997] start from the framework of Littlewood and Miller [1989] and extend it: they look at the impact of multiple versions beyond failure diversity, including other targets for diversity, such as specializing the performance of some versions for specific tasks. They evaluate the possibility of an optimal diversity level for reliable software. Partridge and Krzanowski provide an initial attempt to understand the role of software diversity at multiple levels and to systematically quantify diversity in complex systems.

Van der Meulen and Revilla [2008] analyze the impact of design diversity with thousands of programs that all implement the same set of requirements. Those programs come from the UVa Online Judge Web site, which proposes a set of programming challenges that can be automatically corrected. Hence, the programs were written by thousands of anonymous programmers attracted by the Web site concept. Van der Meulen and Revilla use the frameworks of Eckhardt and Lee [1985] and Littlewood and Miller [1989]. The authors classify different categories of faults that occur in different versions, then, through random selections of pairs of versions, they evaluate the reliability of the system (assuming that the system does not fail if one of the versions does not fail). They confirm that N-version design is more effective when different versions fail independently and that the diversity of programming language has a positive effect (programmers make different faults and different kinds of faults with different languages). Given the size of their dataset, the authors stress the statistical validity of their findings.

Salako and Strigini [2014] question the independent sampling assumption posed by the models of Eckhardt and Lee [1985] and Littlewood and Miller [1989]. They analyze the consequences of violating this assumption and evaluate the opportunity of using different versions of a program (not developed independently) to build fault-tolerant systems. Their results confirm the important influence of independence on diversity. Yet they also open the discussion about different forms of independence and different processes that can be applied to mitigate the influences between different versions.

A large number of theoretical and empirical studies have dissected the foundations of design diversity. We have summarized these works here and discussed how they have contributed to a fine-grain understanding of the conditions for effective design diversity.

5.2. Study of Natural Software Diversity

Natural software diversity is any form of software diversity that spontaneously emerges from software development. The emergence comes from many factors, such as the market competition and the diversity of developers, languages, or execution environments. In Section 3, we discussed how natural diversity can be used to establish reliable software systems (Section 3.2). In this section, we resume the conversation on natural diversity and discuss the literature that studies and describes this existing natural

diversity. The different studies presented here explore different kinds of software diversity: in software components, in source code, and in the social behaviors in open source communities.

Gashi et al. [2004] studied bug reports for four off-the-shelf SQL servers (Oracle 8.0.5, Microsoft SQL, PostgreSQL 7.0.0, and Interbase 6.0) to understand whether these solutions could be good candidates for fault tolerance (i.e., exhibit failure diversity). Their study consisted of selecting bugs for each of the servers, collecting the test cases that trigger the bug on a server, and running them on the other servers to check whether the other solutions present the same bug. Following this protocol, for a total of 181 bugs, they observed that only four were bugs in two versions simultaneously, and no bug was found in more than 2 versions. They emphasize that the diversity of solutions is major asset for forward error recovery, as it is possible to copy the state of a correct database in a failed one. They propose using this natural diversity to design an architecture for a fault-tolerant database management system [Gashi et al. 2007].

Barman et al. [2009] focus on host intrusion detection systems (HIDS) deployed on all machines of enterprise networks. The ability of an IDS to detect intrusions depends on different thresholds that should depend on each user, yet these thresholds are usually set to the same value on each machine, because of a lack of guidelines about how to configure them. The authors analyze the impact of this monoculture of HIDS, showing that it provides very poor results in terms of intrusion detection. These poor results are mainly because the behavior of users are so diverse that they HIDS should also have diverse configurations to be effective. Then, the authors experiment with increasing configuration diversity and observe a clear benefit to reduce the number of missed detections.

Koopman and DeVale [1999] evaluate the diversity of POSIX operating systems using a robustness metric based on failure rates. The authors compare 13 implementations of POSIX. They use the Ballista testing tool to generate large quantities of robustness test cases that they run on each version. This reveals between a 6% and 19% failure rate. Then, the authors perform a multiversion comparison to analyze the diversity of failures and thus the usability of these POSIX versions for N-version fault-tolerance. The results demonstrate that multiversions can be used to increase robustness, yet with the two most diverse solutions, there is still a 9.7% common mode failure exposure for system calls.

Han et al. [2009] analyze the diversity of COTS with respect to their diversity of vulnerabilities. They provide a systematic analysis of the ability of multiversion systems to prevent exploits. The study is based on 6,000 vulnerabilities published in 2007. The main result is that components available for Web servers are diverse with respect to their vulnerabilities and cannot be compromised by the same exploit. Consequently, all of these components can run on multiple operating systems to increase diversity. They conclude that the natural diversity of off-the-shelf software applications is beneficial to build attack-tolerant systems.

Some recent work studied the natural diversity or redundancy that emerges in large-scale source code. Gabel and Su [2010] analyze uniqueness in source code through the analysis of 6,000 programs covering 420 million lines of code. The authors focus on the level of granularity at which diversity emerges in source code. Their main finding is that for sequences up to 40 tokens, there is a lot of redundancy. Beyond this (of course fuzzy) threshold, the diversity and uniqueness of source code appears. Jiang and Su [2009] propose an approach for the identification of functionally equivalent source code snippets in large software projects. This approach consists of extracting code snippets of a given length, randomly generating input data for these snippets, and identifying the snippets that produce the same output values (which are considered functionally equivalent with respect to the set of random test inputs). They ran their analysis on

the Linux kernel 2.6.24 for several days and found a large number of functionally equivalent code fragments, most of which were syntactically different. Both studies explore the tension between redundancy and diversity that exists in software.

Mendez et al. [2013] analyze the diversity in source code at the level of usages of Java classes. They analyzed hundreds of thousands of Java classes, looking for type usages (i.e., sets of methods called on an object of a given type). They found 748 classes with more than 100 different usages of the API, with the most extreme case being the `String` of the Java library, for which they found 2,460 different usages. This reveals a very high degree of usage diversity in object-oriented software.

Diversity also emerges in social behaviors in open source software development. In this area, Posnett et al. [2013] analyze the focus of developers (whether they contribute too few or too many artifacts) and the ownership (to what extent an artifact is "owned" by one or several developers). Through an analogy with predator-prey relations, they set up entropy measures to quantify the diversity in focus and ownership. They observe high levels of diversity in open source projects and also demonstrate that these entropy metrics have good predictive properties: focused developers introduce fewer defects, whereas artifacts that receive contributions from several developers tend to have more defects. Vasilescu et al. [2013] study the development of the GNOME community and observe diversity both from the point of view of contributors (the diversity of activities of different project contributors) and from that of the project (the diversity of activities going on in different GNOME projects).

Software diversity spontaneously emerges through multiple phenomena. In this section, we have discussed the methods to study these different phenomena, as well as the experimental procedures that have been implemented to analyze the impact of this specific form of software diversity. These recent studies illustrate how the analysis of complex diversification processes must leverage techniques from multiple domains ranging from software analysis, data mining, statistics to threat models, and exploit replication.

5.3. Summary

This section has presented two main areas in the analysis and the theoretical modeling of software diversity and its impact. The first part provided an overview of three decades of works that analyzed N-version programming and proposed several statistical methods and foundational assumptions that underlie the effectiveness of this technique for fault-tolerant software systems. The second part discussed novel work analyzing the implication and the effectiveness of natural software diversity (as presented in Section 3.2) for building resilient systems.

6. CONCLUSION

In this article, we have provided a global picture of the software diversity landscape. We decided to broaden the standard scope of diversity to give a very inclusive vision of the field and hopefully a better understanding of the nature of software diversity. The survey gathered work from various scientific communities (security, software engineering, programming languages), which we organized around one dimension: the diversity engineering technique (managed, automated, natural).

Looking at all of these works from a temporal perspective, we realize that the interest for diversity has always existed in the past 40 years. The latest studies even discover phenomena of natural diversity emergence—that is, diversity is observed, but the processes that led to its presence are unknown. We believe that harnessing this natural diversity will be an essential step in the future of software diversification. This could be the intermediate step toward the amplification of natural diversity. Indeed, diversity in natural complex systems is never explicitly developed, but emerges as a side effect of

other phenomena. For example, biodiversity at different scales of ecosystems emerges as the result of sexual reproduction, mutation, dispersal, and frequency-dependent selection [De Aguiar et al. 2009; Melián et al. 2010]. To this extent, the main area of future work is to identify the software engineering principles and evolution rules that drive the emergence and the constant renewal of diversity in software systems. In other words, can we engineer open-ended software diversification?

ACKNOWLEDGMENTS

We would like to thank Paul Amman, Benoit Gauzens, and Sebastian Banescu for their valuable feedback on this article.

REFERENCES

Abdulrahman Alarifi and Wenliang Du. 2006. Diversify sensor nodes to improve resilience against node compromise. In *Proceedings of the 4th ACM Workshop on Security of Ad Hoc and Sensor Networks*. ACM, New York, NY, 101–112.

Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, and Maxime Tricoire. 2015. Multitier diversification in Web-based software applications. *IEEE Software* 32, 1, 83–90.

Paul E. Ammann and John C. Knight. 1988. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers* 37, 4, 418–425.

Colin Atkinson. 2002. *Component-Based Product Line Engineering with UML*. Pearson Education.

Algirdas Avizienis. 1985. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering* 11, 12, 1491–1501.

Algirdas Avizienis. 1995. The methodology of N-version programming. *Software Fault Tolerance* 3, 23–46.

Algirdas Avizienis. 2000. Design diversity and the immune system paradigm: Cornerstones for information system survivability. In *Proceedings of the 3rd Information Survivability Workshop (ISW'00)*.

Algirdas Avizienis and John P. J. Kelly. 1984. Fault tolerance by design diversity: Concepts and experiments. *Computer* 17, 8, 67–80.

Sebastian Banescu, Alexander Pretschner, Dominic Battré, Stéfano Cazzulani, Robert Shield, and Greg Thompson. 2015. Software-based protection against changeware. In *Proceedings of the Conference on Data and Application Security and Privacy (CODASPY'15)*. 231–242.

Dhiman Barman, Jaideep Chandrashekar, Nina Taft, Michalis Faloutsos, Ling Huang, and Frederic Giroire. 2009. Impact of it monoculture on behavioral end host intrusion detection. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*. ACM, New York, NY, 27–36.

Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. 2003. *Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks*. Technical Report TR-CS-2003-10. University of New Mexico. http://www.cs.unm.edu/∼moore/tr/03-02/rise.pdf.

Benoit Baudry, Simon Allier, and Martin Monperrus. 2014. Tailored source code transformations to synthesize computationally diverse program variants. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. 149–159. http://hal.archives-ouvertes.fr/docs/00/93/88/55/PDF/sosies.pdf.

Sandeep Bhatkar, Daniel C. DuVarney, and Ron Sekar. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the USENIX Security Symposium*.

Sandeep Bhatkar, Ron Sekar, and Daniel C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the USENIX Security Symposium*. 271–286.

Peter Bishop, Robin Bloomfield, Ilir Gashi, and Vladimir Stankovic. 2011. Diversity for security: A study with off-the-shelf antivirus engines. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'11)*. 11–19.

Peter Bishop, David G. Esp, Mel Barnes, Peter Humphreys, Gustav Dahll, and Jaakko Lahti. 1986. PODS—a project on diverse software. *IEEE Transactions on Software Engineering* 12, 9, 929–940.

Jean-Marie Borello, Eric Filiol, and Ludovic Mé. 2010. From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques. *Journal in Computer Virology* 6, 3, 277–287.

Stephen W. Boyd and Angelos D. Keromytis. 2004. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security Conference (ACNS'04)*. 292–302.

Juan Caballero, Theocharis Kampouris, Dawn Song, and Jia Wang. 2008. Would diversity really increase the robustness of the routing infrastructure against software defects? In *Proceedings of the 16th*

Annual Network and Distributed System Security Symposium (NDSS'08). http://software.imdea.org/~juanca/papers/coloring_ndss08.pdf.

Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. 2010. Automatic workarounds for Web applications. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, 237–246.

Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software* 83, 1, 60–66.

Monica Chew and Dawn Song. 2002. *Mitigating Buffer Overflows by Operating System Randomization*. Technical Report CS-02-197. Carnegie Mellon University.

Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. 2011. Abstract delta modeling. *ACM SIGPLAN Notices* 46, 2, 13–22. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW592.pdf.

Paul Clements and Linda Northrop. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Frederick B. Cohen. 1993. Operating system protection through program evolution. *Computers and Security* 12, 6, 565–584.

Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. 2012. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, New York, NY, 319–328.

Christian Collberg, Clark Thomborson, and Douglas Low. 1997. *A Taxonomy of Obfuscating Transformations*. Technical Report. Department of Computer Science, University of Auckland, New Zealand.

Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, 184–196.

Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the USENIX Security Symposium (USENIX-SS'06)*. http://dl.acm.org/citation.cfm?id=1267336.1267344

Marcus Aloizo Martinez De Aguiar, Michel Baranger, Elizabeth M. Baptestini, Les Kaufman, and Yaneer Bar-Yam. 2009. Global patterns of speciation and diversity. *Nature* 460, 7253, 384–387.

Yves Deswarte, Karama Kanoun, and Jean-Claude Laprie. 1998. Diversity against accidental and deliberate faults. In *Proceedings of the Conference on Computer Security, Dependability, and Assurance: From Needs to Solutions (CSDA'98)*. IEEE, Los Alamitos, CA, 171–181. http://dl.acm.org/citation.cfm?id=519453.793920

Dave E. Eckhardt and Larry D. Lee. 1985. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering* 11, 12, 1511–1517.

Robert Feldt. 1998. Generating diverse software versions with genetic programming: An experimental study. *IEE Proceedings on Software* 145, 6, 228–236.

Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. 2008. Searching for cognitively diverse tests: Towards universal test diversity metrics. In *Proceedings of the 1st Search-Based Software Testing Workshop (SBST'08)*. 178–186. http://www.cse.chalmers.se/~feldt/publications/feldt_2008_sbst_universal_test_diversity.html.

Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. 2008. Evolving software product lines with aspects. In *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering (ICSE'08)*. IEEE, Los Alamitos, CA, 261–270. http://www.facom.ufu.br/~figueiredo/publications/icse08ready.pdf.

Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HOTOS'97)*. IEEE, Los Alamitos, CA, 67. http://dl.acm.org/citation.cfm?id=822075.822408

Blair Foster and Anil Somayaji. 2010. Object-level recombination of commodity applications. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO'10)*. 957–964.

Michael Franz. 2010. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the Workshop on New Security Paradigms*. ACM, New York, NY, 7–16.

Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, 147–156.

Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. 2014. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience* 44, 6, 735–770.

Ilir Gashi, Peter Popov, and Lorenzo Strigini. 2004. Fault diversity among off-the-shelf SQL database servers. In *Proceedings of International Conference on Dependable Systems and Networks*. IEEE, Los Alamitos, CA, 389–398. core.kmi.open.ac.uk/download/pdf/2707859.pdf.

Ilir Gashi, Peter Popov, and Lorenzo Strigini. 2007. Fault tolerance via diversity for off-the-shelf products: A study with SQL database servers. *IEEE Transactions on Dependable and Secure Computing* 4, 4, 280–294. http://www.researchgate.net/publication/3449451_Fault_Tolerance_via_Diversity_for_Off-the-Shelf_Products_A_Study_with_SQL_Database_Servers/file/e0b4952794d1aaca43.pdf.

Anatoliy Gorbenko, Vyacheslav Kharchenko, Olga Tarasyuk, and Alexander Romanovsky. 2011. Using diversity in cloud-based deployment environment to avoid intrusions. In *Proceedings of the 3rd International Conference on Software Engineering for Resilient Systems*. 145–155.

Jin Han, Debin Gao, and Robert H. Deng. 2009. On the effectiveness of software diversity: A systematic study on real-world vulnerabilities. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 127–146.

Leslie Hatton. 1997. N-version design vs. one good version. *IEEE Software* 14, 6, 71–76.

Oystein Haugen, Birger Moller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. 2008. Adding standardized variability to domain specific languages. In *Proceedings of the 12th International Software Product Line Conference (SPLC'08)*. IEEE, Los Alamitos, CA, 139–148.

Scott A. Hendrickson and Andre van der Hoek. 2007. Modeling product line architectures through change sets and relationships. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 189–198.

Matti A. Hiltunen, Richard D. Schlichting, Carlos A. Ugarte, and Gary T. Wong. 2000. Survivability through customization and adaptability: The Cactus approach. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX'00)*, Vol. 1. 294–307. http://www.dependability.org/wg10.4/meeting38/12-Schli.pdf.

Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-guided automated software diversity. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*. 1–11.

Todd Jackson. 2012. *On the Design, Implications, and Effects of Implementing Software Diversity for Security*. Ph.D. Dissertation. University of California, Irvine.

Todd Jackson, Babak Salamat, Andrei Homescu, Karthikeyan Manivannan, Gregor Wagner, Andreas Gal, Stefan Brunthaler, Christian Wimmer, and Michael Franz. 2011. Compiler-generated software diversity. In *Moving Target Defense*. Springer, 77–98.

Matthias Jacob, Mariusz H. Jakubowski, Prasad Naldurg, Chit Wei Nick Saw, and Ramarathnam Venkatesan. 2008. The superdiversifier: Peephole individualization for software protection. In *Advances in Information and Computer Security*. Springer, 100–120.

Sushil Jajodia, Anup K. Ghosh, Vipin Swarup, Cliff Wang, and X. Sean Wang. 2011. *Moving Target Defense*. Springer.

Jean-Marc Jézéquel. 1998. Reifying configuration management for object-oriented software. In *Proceedings of the 1998 International Conference on Software Engineering.* 240–249. http://www.irisa.fr/triskell/publis/1998/Jezequel98b.pdf.

Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*. ACM, New York, NY, 81–92.

James E. Just and Mark Cornwell. 2004. Review and analysis of synthetic diversity for breaking monocultures. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode (WORM'04)*. ACM, New York, NY, 23–32. DOI:http://dx.doi.org/10.1145/1029618.1029623

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. DTIC.

Karama Kanoun. 1999. Cost of software design diversity an empirical evaluation. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'99)*. IEEE, Los Alamitos, CA, 242–247.

Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. ACM, New York, NY, 272–280.

John P. J. Kelly, Thomas I. McVittie, and Wayne I. Yamamoto. 1991. Implementing design diversity to achieve fault tolerance. *IEEE Software* 8, 4, 61–71.

Angelos Keromytis and Vassilis Prevelakis. 2005. *A Survey of Randomization Techniques against Common Mode Attacks*. Technical Report DU-CS-05-04. Department of Computer Science, Drexel University.

John C. Knight. 2011. Diversity. In *Dependable and Historic Computing*. Lecture Notes in Computer Science, Vol. 6875. Springer, 298–312.

John C. Knight, Jack W. Davidson, David Evans, Anh Nguyen-Tuong, and Chenxi Wang. 2007. *Genesis: A Framework for Achieving Software Component Diversity*. Technical Report. DTIC.

John C. Knight and Nancy G. Leveson. 1986. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering* 12, 1, 96–109.

John C. Knight, Chenxi Wang, Kevin J. Sullivan, and Matthew C. Elder. 2000. Survivability architectures: Issues and approaches. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Vol. 2. IEEE, Los Alamitos, CA, 1157.

Philip Koopman and John DeVale. 1999. Comparing the robustness of POSIX operating systems. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*. IEEE, Los Alamitos, CA, 30–37.

Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated software diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. 276–291.

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the ACM/IEEE 32nd International Conference on the Software Engineering*, Vol. 1. 105–114. http://www.cs.cmu.edu/~ckaestne/pdf/icse10.pdf.

Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. 2009. Polymorphing software by randomizing data structure layout. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 107–126.

Bev Littlewood and Douglas R. Miller. 1989. Conceptual modeling of coincident failures in multiversion software. *IEEE Transactions on Software Engineering* 15, 12, 1596–1614.

Bev Littlewood, Peter Popov, and Lorenzo Strigini. 2001. Modeling software design diversity: A review. *ACM Computing Surveys* 33, 2, 177–208.

Bev Littlewood and John Rushby. 2012. Reasoning about the reliability of diverse two-channel systems in which one channel is "possibly perfect." *IEEE Transactions on Software Engineering* 38, 5, 1178–1194. http://openaccess.city.ac.uk/1069/1/1oo2-revised-13apr11.pdf.

Alex X. Liu and Mohamed G. Gouda. 2008. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems* 19, 9, 1237–1251.

Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. 2006. Software self-healing using collaborative application communities. In *Proceedings of the 2006 Network and Distributed System Security Symposium (NDSS'06)*.

Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. 2011. A taxonomy of self-modifying code for obfuscation. *Computers and Security* 30, 8, 679–691. http://cosic.esat.kuleuven.be/publications/article-1529.pdf.

Kevin Shear McCann. 2000. The diversity–stability debate. *Nature* 405, 6783, 228–233.

Carlos J. Melián, David Alonso, Diego P. Vázquez, James Regetz, and Stefano Allesina. 2010. Frequency-dependent selection predicts patterns of radiations and biodiversity. *PLoS Computational Biology* 6, 8, 31000892.

Diego Mendez, Benoit Baudry, and Martin Monperrus. 2013. Empirical evidence of large-scale diversity in API usage of object-oriented software. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'13)*. http://www.monperrus.net/martin/diversity-api-usage.pdf.

Gert Merckxt. 2006. *Software Security through Targeted Diversification*. Master's thesis, Katholieke Universiteit Leuven.

Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Prentice Hall.

Subhasish Mitra, Nirmal R. Saxena, and Edward J. McCluskey. 1999. A design diversity metric and reliability analysis for redundant systems. In *Proceedings of the 1999 International Test Conference.* 662–671.

Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. 2008. An aspect-oriented and model-driven approach for managing dynamic variability. In *Model Driven Engineering Languages and Systems*. Springer, 782–796.

Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, and Jack W. Davidson. 2008. Security through redundant data diversity. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks with FTCS and DCC (DSN'08)*. IEEE, Los Alamitos, CA, 187–196.

Victor F. Nicola and Ambuj Goyal. 1990. Modeling of correlated failures and community error recovery in multiversion software. *IEEE Transactions on Software Engineering* 16, 3, 350–359.

Jon Oberheide, Evan Cooke, and Farnam Jahanian. 2008. CloudAV: N-version antivirus in the network cloud. In *Proceedings of the USENIX Security Symposium*. 91–106. http://www.eng.auburn.edu/~xqin/courses/comp7370/CloudAV-sec08.pdf.

Adam J. O'Donnell and Harish Sethu. 2004. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 121–131. http://pdf.aminer.org/000/084/043/on_achieving_software_diversity_for_improved_network_security_using_distributed.pdf.

Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. 2002. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers* 51, 2, 180–199.

Derek Partridge and Wojtek Krzanowski. 1997. Software diversity: Practical statistics for its measurement and exploitation. *Information and Software Technology* 39, 10, 707–717.

Klaus Pohl, Günter Böckle, and Franck Van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag, New York, NY.

Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. 2012. An empirical study of the effectiveness of "forcing" diversity based on a large population of diverse programs. In *Proceedings of the International Symposium on Software Reliability Engineering*. 41–50.

Peter Popov and Lorenzo Strigini. 2001. The reliability of diverse systems: A contribution using modelling of the fault creation process. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*. IEEE, Los Alamitos, CA, 5–14.

Daryl Posnett, Raissa D'Souza, Premkumar Devanbu, and Vladimir Filkov. 2013. Dual ecological measures of focus in software development. In *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE'13)*. IEEE, Los Alamitos, CA, 452–461.

Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*. 419–443. http://wwwbroy.informatik.tu-muenchen.de/publ/papers/ecoop_prehofer_1997_Publication.pdf.

Brian Randell. 1978. *System Structure for Software Fault Tolerance*. Springer.

Kizito Salako and Lorenzo Strigini. 2014. When does "diversity" in development reduce common failures? Insights from probabilistic modelling. *IEEE Transactions on Dependable and Secure Computing* PP, 99, 1.

Babak Salamat, Andreas Gal, and Michael Franz. 2008. Reverse stack execution in a multi-variant execution environment. In *Proceedings of the Workshop on Compiler and Architectural Techniques for Application Reliability and Security*.

Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European Conference on Computer Systems*. ACM, New York, NY, 33–46.

Babak Salamat, Todd Jackson, Gregor Wagner, Christian Wimmer, and Michael Franz. 2011. Runtime defense against code injection attacks using replicated execution. *IEEE Transactions on Dependable and Secure Computing* 8, 4, 588–601.

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. 2010. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond*. Springer, 77–91. http://rap.dsi.unifi.it/phpbibliography/files/main.pdf.

Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software diversity: State of the art and perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5, 477–495. DOI:http://dx.doi.org/10.1007/s10009-012-0253-y.

Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*. ACM, New York, NY, 119–126. http://klausschmid.net/wp-content/uploads/paper/SchmidRabiserGruenbacher11.pdf.

Eric Schulte, Zachary Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2013. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3, 281–312. DOI:http://dx.doi.org/10.1007/s10710-013-9195-8

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'11)*. ACM, New York, NY, 124–134. DOI:http://dx.doi.org/10.1145/2025113.2025133

Eric Totel, Frédéric Majorczyk, and Ludovic Me. 2006. COTS diversity based intrusion detection and application to Web servers. In *Recent Advances in Intrusion Detection*. Springer, 43–62.

Kalyanaraman Vaidyanathan and Kishor S. Trivedi. 2005. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing* 2, 2, 124–137.

Meine J. P. van der Meulen and Miguel A. Revilla. 2008. The effectiveness of software diversity in a large population of programs. *IEEE Transactions on Software Engineering* 34, 753–764.

Rob Van Ommering. 2002. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering*. ACM, New York, NY, 255–265.

Bogdan Vasilescu, Alexander Serebrenik, Mathieu Goeminne, and Tom Mens. 2013. On the variation and specialisation of workload-a case study of the Gnome ecosystem community. *Empirical Software Engineering* 19, 4, 955–1008.

Chenxi Wang, Jack Davidson, Jonathan Hill, and John C. Knight. 2001. Protection of software-based survivability mechanisms. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*. IEEE, Los Alamitos, CA, 193–202.

Rong Wang, Feiyi Wang, and Gregory T. Byrd. 2003. Design and implementation of Acceptance Monitor for building intrusion tolerant systems. *Software: Practice and Experience* 33, 14, 1399–1417.

Jan Gerben Wijnstra. 2000. Supporting diversity with component frameworks as architectural elements. In *Proceedings of the 2000 International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 51–60.

Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. 2009. Security through diversity: Leveraging virtual machine technology. *IEEE Security and Privacy* 7, 1, 26–33. https://www.cs.unm.edu/~forrest/classes/readings/Williams09.pdf.

Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2003. Transparent runtime randomization for security. In *Proceedings of the International Symposium on Reliable Distributed Systems*. IEEE, Los Alamitos, CA, 260–269.

Chang Sik Yoo and Poong Hyun Seong. 2002. Experimental analysis of specification language diversity impact on {NPP} software diversity. *Journal of Systems and Software* 62, 2, 111–122.

Tewfic Ziadi, Loic Helouet, and Jean-Marc Jezequel. 2004. Revisiting statechart synthesis with an algebraic approach. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 242–251.

Received March 2014; revised March 2015; accepted July 2015