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

Benoit Baudry, Martin Monperrus

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

Benoit Baudry *[1] and Martin Monperrus†[2]

[1]Inria, France
[2]University of Lille, France

Abstract

Early experiments with software diversity in the mid 1970's 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 paper contributes to the field of software diversity as the first paper 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 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.

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 these domains, diversity (a fact) is considered essential for the emergence of resilience, stability or novelty (a property) [76]. In software, we take the problem upside-down. We want properties, e.g. resilience, for which diversity may be the key. The main research question is thus formulated as: how to create, maintain, exploit – i.e. engineer – diversity in software?

For instance, early experiments with software diversity in the mid 1970's (e.g. recovery blocks [94]) 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 [31] or for security [56] provide important milestones in this direction. Yet, 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 paper contributes to the field of software diversity, as the first paper 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

*benoit.baudry@inria.fr
†martin.monperrus@univ-lille1.fr

Table 1: The diversity of software diversity (not exhaustive overview). Over time and over research communities, many kinds of software diversity have been proposed or studied.

Software diversity for ...	Fault tolerance [8, 94], security [29, 36], reusability [89], software testing [21], performance [104], bypassing antivirus software [17] ...
Software diversity at the scale of ...	Networks [86], operating systems [66], components [41], data structures [3], statements [103], ...
Software diversity as ...	a natural phenomenon [78], a goal [25], a means [26], a research object [64] ...
Software diversity in ...	market products [44], operating systems [66], developer expertise [92], ...
Software diversity when ...	the specifications are written [115], the code is developed [8], the application is deployed [38], executed [3] ...

communities are words, we had to cross terminological chasms several times: diversity, randomization, poly- and meta-morphism, 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 [31, 56, 62, 100], 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 they share a similar underlying concept: software diversity.

Novelty The field of software diversity has been very active in the 70's and 80's for fault-tolerance purposes. There has been a revival in the late 90's, early 2000's, this time with automatic diversity for security. Both periods have been covered by previous surveys [31, 56]. The last 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. [100], Knight [62] and Larsen et al. [67], which have specific focuses. Our survey includes the most recent works in all areas of software diversity, with an emphasis from 2000 to present.

Audience The targeted audience of this paper 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 multi-tier software diversification techniques [2]. 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 paper. Software diversity has multiple facets: the goal of diversity, the diversification techniques, the scale of diversity, the application domain, when it is applied ... This diversity of software diversity is reflected in Table 1. As shown in Figure 1, we decide 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 diversity is a study subject (Section 5.2). Then, we split the engineering papers on *managed diversity* approaches, that 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 paper 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 her own one.

2 Survey Process

To prepare this survey, we first analyzed the existing surveys on the topic (see Section 2.1). None of them covers the material 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. in 1998 [31]. 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 90ies' software

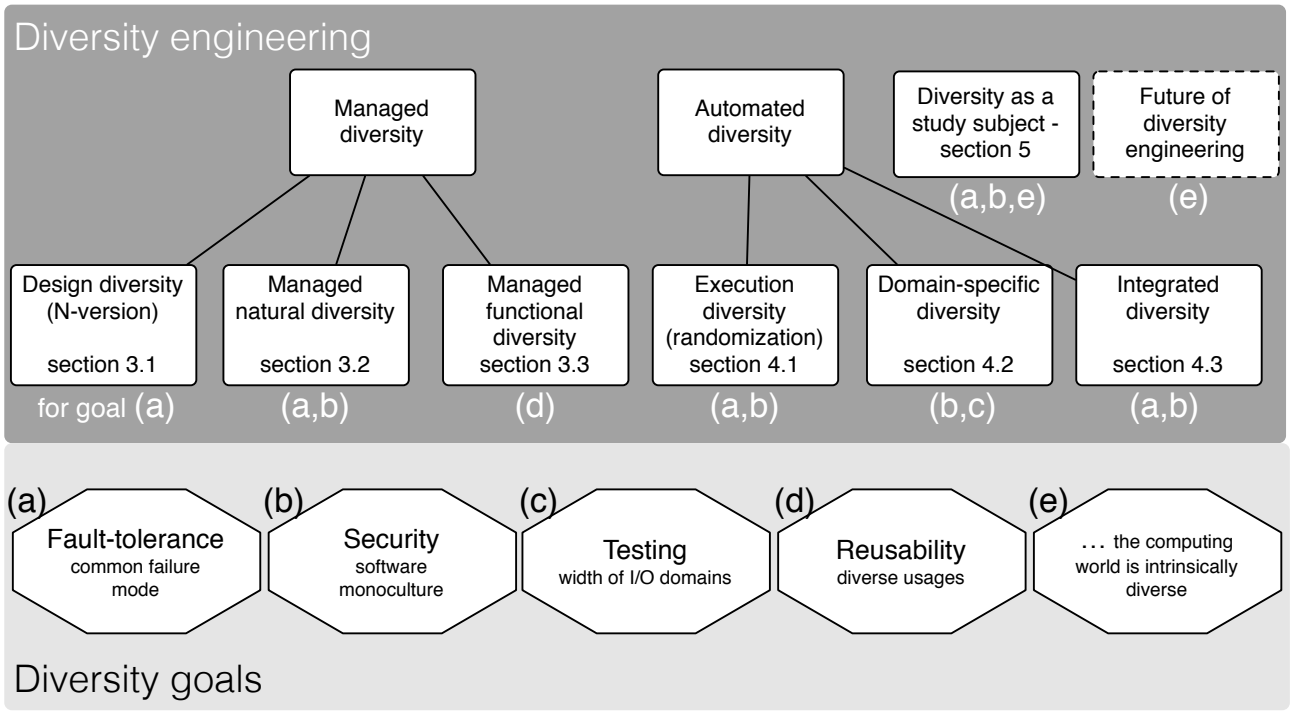

Figure 1: The diverse dimensions of software diversity

diversity, our survey discusses the rich work that has been done around software diversity during the last fifteen years: instruction-set randomization, adaptive random testing, and many others.

In 2001, Littlewood et al. [70] focus on design diversity (N-version programming). They review in particular 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 Just et al.'s review paper [56] is to list the techniques of synthetic diversity that can improve software survivability. "Synthetic diversity" is equivalent, in our views, to "artificial automated diversity". In our paper, we consider other goals than only security (such as quality of service, see Section 4.1.3), and consider other diversity engineering techniques (e.g., managed software diversity, see Section 3).

John Knight published a survey in 2011 [62]. He discusses four kinds of diversity: classical design diversity (N-version and recovery block), data diversity (a research direction he has both invented and lead), 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, for instance 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 (such as software product lines, see 3.3.3), more kinds of artificial diversity (such as runtime diversity, see Section 4.1.2), and papers for which diversity is the main study subject (see Section 5).

Schaefer and colleagues co-authored in 2012 "Software diversity: state of the art and perspectives" [100]. 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 paper, not only do we describe other kinds of managed software diversity such as design diversity, but we also discuss artificial diversity and natural diversity as well.

Larsen et al. [67] 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 vs. binary rewriting diversity synthesis.

2.2 Systematic Process

We followed a systematic process to select the papers discussed in this paper. We started with 30 papers that we knew and are written by the most remarkable authors: Avizienis, Randell, Forrest, Cohen, Knight and Levenson, Schaefer, etc.. 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 that discuss a similar problem or solution (e.g., we selected only a few papers about product lines or about multi-version execution). Second, we filtered out the papers that had no impact on the literature (that appear in unknown conferences or that had less than 5 citations after 20 years). Since our survey focuses on recent developments in the field of software diversity, we took a 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 multi-version 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 1970's 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, *i.e.*, an internal problem unintentionally introduced by a developer of the execution environment. N-version programming [5] and recovery blocks [94] 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" [5,8]. This consists in providing N development teams with the same requirements. Those teams then develop N independent versions, using different technologies, processes, verification techniques, etc. 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 [6]. For example Kelly [60] distinguishes 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 proposed to adapt the concept to software survivability [7].

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 [94]. The idea is to have recovery blocks in the program, *i.e.*, 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 [6]. This framework has then been used in multiple domains, including the design of multiple versions of firewalls [73]. While 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 which spontaneously emerge from software development processes. There exists 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, 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, etc. In this section we present a set of works which exploit this natural diversity for different purposes. We will come back to natural diversity later in Section 5.2, for discussing authors who study natural diversity with no engineering goals at all.

Hiltunen et al. [48] propose the Cactus mechanism for survivability, i.e., a mechanism that monitors and controls a running application in order 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 they also discuss how this natural way of changing a system supports the emergence of natural diversity and thus increases resilience.

Caballero et al. [19] 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. [105] propose to design 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 [41, 111] 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 last month (800000 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. [40] conducted a study on the impact of operating system diversity w.r.t. to security bugs of the NIST National Vulnerability Database (NVD). Their results show that diversity indeed contribute to building intrusion-tolerant systems.

Oberheide et al. [85] 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 send them in the cloud to check for viruses and malware against diverse antivirus systems. They evaluate their system over 7220 malware 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 explored by Bishop et al. [15], who explored the deep characteristics of the dataset of known malware to reduce global vulnerability.

O'Donnell and Sethu [86] leverage the diversity of software packages in operating systems and investigates 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 of setting a network that is tolerant to attacks. The experiments are based on a simulation, which 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. [20] find multiple different sequences of method calls in Javascript code, which happen to have the same behavior. They harness this redundancy to setup a runtime recovery mechanism for web applications.

Gorbenko et al. [43] propose an intrusion avoidance architecture based on multi-level software diversity and dynamic software reconfiguration in IaaS cloud layers. The approach leverages the natural diversity of off-the-shelf components 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 4 diverse operating systems that have different open vulnerabilities. They discuss how this mechanism reduces exposure to vulnerabilities.

Summary

In this subsection, we have focused on techniques that exploit the natural diversity of that can be found among off-the-shelf components or even as redundancy in programs. All 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 (say for network or for file input/output) of this abstract function are radically different. One responsibility of software abstractions is to capture this conceptual identity and to abstract over the diversity of implementation details. The goals of of this abstraction are reuse, modularity, maintainability, extensibility, etc.

For instance, Unix is well known because of Unix' 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, as well techniques related to the software product line research field.

3.3.1 Class Diversity

The object-oriented software paradigm is a rich paradigm with implications on understandability, reuse, etc. There is one point in this paradigm really related to managing the diversity: polymorphism.

Polymorphism is the mechanism enabling us to have code that calls other pieces of code in a non predefined 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 to manage 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 Bertrand Meyer [80] 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 means to design open software systems. Plugins are software units that encapsulate a given functionality as well as some information about its dependencies. As far as we know, Wijnstra [112] 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 [112]. He proposed to use 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 or 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 25000 plugins available for Wordpress, which 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 Ommering [108], who used a plugin-based architecture in which connections between plugins handle design-time or run-time diversity.

3.3.3 Software Product Lines

The techniques around software product lines can be considered as means of controlling a diversity of software solutions capable of handling a diversity of requirements (user requirements or environmental constraints) [24,89].

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*" [89]. This consists in building an explicit variability model, which 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 [57] or a decision model [101].

In the context of software product lines, the main challenge for software diversity management consists in providing systematic ways to reuse existing parts of software systems in order 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.'s survey "Software diversity: state of the art and perspectives" [100]. 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 [24]. 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 [4]. Hendrikson et al. [47] 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. [116] propose sound composition operations for UML 2.0 scenarii in order to automatically synthesize diverse statecharts inside a given product line, while Morin et al. [82] compose software components to derive software configurations at runtime. Other approaches rely on an orthogonal variability model associated to model transformations for product derivation, as is the case for the Common Variability Language [46] or the Orthogonal Variability Model [89], 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 [54]. At the source code level, there exist several mechanisms to manage a set of variants for a given program: feature-oriented programming [93] proposes a flexible model for object composition; delta-oriented programming [99] instantiates the concept of delta-modeling [23] 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 and colleagues have reported on the usage of aspect-oriented programming to hanlde variants in a product line and discuss the postive and negative effects on design stability [35]; preprocessing was one of the first language technology used to handle program variants and has been extensively analyzed, for example in the recent work by Liebig et al. [68].

3.3.4 Discussion

The main benefit of those 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 [2]. Both relate to diversity (the second one in a dual manner). In practice, there is an engineering tradeoff between the increase of diversity due to the very large number of possible combinations and the decrease of diversity due to massive reuse.

3.4 Summary

This section has focused on three areas of software engineering, which *manage* software diversity. The first was about multi-version design, an approach to fault-tolerance that aims at managing the manual development of diverse program versions. The second part was about managing and exploiting software diversity that naturally emerges in software markets or open source communities, in order to build fault or attack tolerant systems. The last part opened on a series of works dedicated to the management of functional diversity, in order 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 in order 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" [56] or "artificial" diversity (e.g. [74]). However, artificial literaly 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 human 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.

Automated software diversity is valuable in different contexts, for instance 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 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 specifities of the application domain or the technology used, this is presented in Section 4.2. Finally, the last part of this section – Section 4.3 – is about 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, for instance to improve security [13]. 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, set of unique executions for the very same program. As mentioned by [13], *"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, etc. 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, in particular the one of Keromytis and Prevelakis [61].

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 [114]) and discuss it in 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 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 which embed randomization points that themselves produce different executions.

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

4.1.1 Static Randomization

One of seminal papers on static randomization is by Forrest and colleagues [36], who highlight two families of randomization: randomly adding or deleting non-functional code and reordering code. Those transformations are also described by Cohen [25] in the context of operating system protection. Lin et al. [69] randomize the data structure of C code. Following the line of thought of Forrest et al. [36] they re-order fields of data structures (`struct` and `class` in C/C++ code) and insert garbage ones.

The concept of instruction-set randomization has been invented in 2003 in two independent teams [11,59] It consists of creating a unique mapping between artificial CPU instructions and real ones. This mapping is encoded in a key which 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 which is the essence of the counter-measure against code injection.

In some execution environments (e.g. x86 CPUs), there exists a "NOP" instruction. It means "no operation" and it has been invented for the sake of optimization, in order to align instructions with respect to some alignment criteria (e.g. memory or cache). Merckx [79] and later Jackson [50] have 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. [9] 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 in a program P' which is distributed. However, since obfuscation is automated, it is often possible to generate several different obfuscated versions of the same program (as proposed by Collberg et al. [26] for example). 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 to the now classical taxonomy by Collberg and colleagues [27]. For an example of a concrete obfuscation engine for Java programs, we refer to [28] and its Figure 1. When obfuscation happens at runtime, it is a kind of execution diversity and we discuss it in 4.1.2.

4.1.2 Dynamic Randomization

Chew and Song [22] 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 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. Bathkar et al. [13,14] 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 diverse version of the same program at compilation time, 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 and colleagues at the University of California Irvine [49]. 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 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 [75]. This is done for sake of security and is considered one of the strongest obfuscation mechanism [75].

Ammann and Knight's "data diversity" [3] 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" [106] 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 [37] 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 emerging but has no guarantee on the behavior of the recombination.

Schulte et al. [102] describe a property of software that has never been reported before. 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 in order to get functionally equivalent variants of a program. Doing this in advance is called "proactive diversity". The authors present a set of experiments that show that this proactive diversity is able to fix certain bugs.

In our previous work [12], we experiment 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 that 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 [104]. In this paper, Sidiroglou et al. 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 subsection, 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) since they dare changing the execution semantics of the program being diversified.

4.2 Domain-specific Diversity

The techniques we have 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 in order to access unauthorized data or corrupt existing one. Boyd et al. [18] 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 [33] 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, i.e. that the approach is effective for the generation of a kind of failure diversity.

Oh et al. [87] 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 anti-virus systems. Inventors of computer viruses of course 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 anti-virus 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 and Mé [17].

In the domain of sensor networks, Alarifi and Du [1] propose an approach to diversifying sensor software in order 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 very 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 to be valuable in test cases as well. Adaptive random testing [21] 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*". Feldt et al.'s VAT model is an example of adaptive random testing [34]. 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 softare diversity. Software diversity is not specific to a paradigm, a platform, a language, an application domain, and it appears that it can be applied to any software object. The concept seems to be as general as "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. [110] propose a multi-level 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. [65]. Wang et al's architecture 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. [13] aim at developing a technique for address obfuscation in order 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. Since all 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.

Knight et al., in a report of the DARPA project Self-Regenerative System (SRS) [63], 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 5 forms of diversity: Address Space Randomization (ASR), Stack Space Randomization (SSR), Simple Execution Randomization (SER), Strong Instruction Set Randomization (SISR), 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 [113]. Diversity transformations are applied 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: calling sequence diversity and instruction set diversity.

Jacob et al. [52] propose superdiversification as a technique that integrates several forms of diversification to synthesize individualized versions of programs. The approach, inspired by compilation superoptimization, consists in 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 [38] 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 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 since 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 as one of the three "game-changing" themes to cyber security the President's Cyber Policy Review announced. The software component of MTD integrates spatial and temporal software diversity, in order to "limit the exposure of vulnerabilities and opportunities for attack" [53]. 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, who suggested multiple kinds of program transformations to diversify software [25], Collberg et al. [26] compose multiple forms of diversity and code replacement in a distributed system in order 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. recently proposed to use software diversification in multiple components of web applications [2]. They combine different software diversification strategies, from the deployment of different vendor solutions, to fine-grained code transformations, in order to provide different forms of protection. Their form of multi-tier 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 [90] 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. [29] introduce the idea of N-variant systems, which consists in automatically generating variants of a given program and then running them in parallel in order 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, since address space partitioning protects against attacks that rely on absolute memory addresses, while 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 [83].

Salamat and colleagues find a nice name for this concept: "multi-variant execution environment" [51, 96].

A multi-variant execution 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 [97,98]. The execution differences allow some kind of analysis and reasoning on the program behavior. For instance, in [96], multi-variant execution enables the authors to detect malicious code trying to manipulate the stack.

Locasto and colleagues [74] introduced the idea of collaborative application communities. The same application (e.g. a web server) is run on different nodes. In presence of bugs (invalid memory accesses), each node tries a different runtime fix alternative. If the fix proves to be successful, a controller shared 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 work have thus experimented with the integration of multiple forms of diversity in a system, ot 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, 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 paper, they are fully automated, which is different from the natural diversity discussed in Section 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, 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 off-the-shelf components 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 70's, a large number of studies have 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 proposal to leverage diversity and redundancy in software for 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. While 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 Levenson [64] provided 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 that were 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. [16] summarized 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 over-power 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 proposed theoretical frameworks to analyze and quantify the effects of N-version design on reliability. Eckhardt and Lee [32] have developed 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 what are the limits of coincidental errors on the effect of N-version design. Littlewood and colleagues have refined the work of Eckhardt, first by considering the diversity of development methods [71], and more recently by adding further hypotheses and studying two-channel systems [72]. They show that methodological diversity, analyzed as the diversity of development decisions, is very likely to produce behavioral diversity. Popov and Strigini [91] proposed 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. [81] defined metrics to quantify diversity in N-version designs and highlighted 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 [84] proposed 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 [45] 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, vs. 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 focuses [58] on a cost analysis of developing 2 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, while the lowest if for functional specification). These results confirm other observations from controlled experiments, with actual data from industrial software development.

Partridge and Krzanowski [88] start from the framework of Littlewood and Miller 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.

More recently, van der Meulen and Revilla [107] 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 Website, 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 website concept. van der Meulen and Revilla use the frameworks of Eckhardt and Lee [32] and Littlewood and Miller [71]. The authors classify different categories of faults that occur in different versions, and then, through random selections of pairs of versions, 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 really stress the statistical validity of their findings.

Salako et al. [95] question the independent sampling assumption posed by the models of Eckhardt and Lee [32] and Littlewood and Miller [71]. 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 finer 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, the diversity of developers, of languages or of execution environments. In Section 3,we have discussed how natural diversity can be used to establish reliable software systems (Section 3.2). In this section, we resume on natural diversity and discuss the literature that studies and describes this existing natural diversity. The different studies presented in this section explore different kinds of software diversity: in software components, in source code, as well as in the social behaviors in open source communities.

Gashi et al. [41] have studied bug reports for 4 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. The study consisted in selecting bugs for each of the servers, collect the test cases that trigger the bug on a server and run 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 4 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, since it is possible to copy the state of a correct database in a failed one. They have proposed to use this natural diversity to design an architecture for a fault-tolerant database management system [42].

Barman et al. [10] focus on host intrusion detection systems (HIDS) deployed on all machines of entreprise 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 De Vale [66] 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 6% and 19% of failure rate. Then, the authors perform a multi-version comparison to analyze the diversity of failures and thus the usability of these POSIX versions for N-version fault-tolerance. The results demonstrate that multi-versions can be used to increase robustness, yet, with the 2 most diverse solutions, there is still a 9.7% common mode failure exposure for system calls.

Han et al. [44] analyze the diversity of off-the-shelf components with respect to their diversity of vulnerabilities. They provide a systematic analysis of the ability of multi version systems to prevent exploits. The study is based on 6000 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 these components can run on multiple operating systems in order 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 study the natural diversity or redundancy that emerges in large-scale source code. Gabel and Su [39] analyze uniqueness in source code through the analysis of 6000 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 [55] propose an approach for the identification of functionally equivalent source code snippets in large software projects. This approach consists in extracting code snippets of a given length, randomly generating input data for these snippets and identify the snippets that produce the same output values (which are considered functionally equivalent, w.r.t the set of random test inputs). They run their analysis on the Linux kernel 2.6.24 during several days and find a large number of functionally equivalent code fragments, most of which are syntactically different. Both studies explore the tension between redundancy and diversity that exists in software.

Mendez et al. [78] analyze the diversity in source code at the level of usages of Java classes. They analyze hundreds of thousands of Java classes, looking for type usages, i.e. sets of methods called on an object of a

given type. They find 748 classes with more than 100 different usages of the API, the most extreme case being the `String` of the Java library, for which they found 2460 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. [92] analyze the focus of developers (whether they contribute to few or 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 less defects, while artifacts that receive contributions from several developers tend to have more defects. Vasilescu et al. [109] studied the development of the GNOME community and observed diversity both from the point of view of contributors (how diverse are the activities of different project contributors) as well as from the point of view of project (how diverse are the 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 techniuqes 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 3 decades of works that analyzed N-version programming and proposed several statistical methods and foundational assumptions that underly the effectiveness of this technique for fault-tolerant software systems. The second part discusses novel work that analyze the implication and the effectiveness of natural software diversity (as presented in Section 3.2) for building resilient systems.

6 Conclusion

In this paper, we provided a global picture of the software diversity landscape. We decided to broaden the standard scope of diversity, in order 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 these works from a temporal perspective, we realize that the interest for diversity has always existed in the last 40 years. The latest studies even discover phenomena of natural diversity emergence, i.e. 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 towards 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 [30, 77]. 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?

Acknowledgements

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

References

[1] Abdulrahman Alarifi and Wenliang Du. Diversify sensor nodes to improve resilience against node compromise. In *Proceedings of the fourth ACM workshop on Security of ad hoc and sensor networks*, pages 101–112. ACM, 2006.

[2] Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, and Maxime Tricoire. Multi-tier diversification in web-based software applications. *IEEE Software*, To appear, 2014.

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

[4] Colin Atkinson. *Component-based product line engineering with UML*. Pearson Education, 2002.

[5] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[6] Algirdas Avizienis. The methodology of n-version programming. *Software fault tolerance*, 3:23–46, 1995.

[7] Algirdas Avizienis. Design diversity and the immune system paradigm: Cornerstones for information system survivability. In *3rd Information Survivability Workshop ISW-2000*, 200.

[8] Algirdas Avizienis and John PJ Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80, 1984.

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

[10] Dhiman Barman, Jaideep Chandrashekar, Nina Taft, Michalis Faloutsos, Ling Huang, and Frederic Giroire. Impact of it monoculture on behavioral end host intrusion detection. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 27–36. ACM, 2009.

[11] Elena Gabriela Barrantes, David H Ackley, Trek S Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. Technical Report TR-CS-2003-10, University of New Mexico, Feb 2003.

[12] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 149–159, CA, USA, 2014.

[13] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the USENIX Security Symposium*, 2003.

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

[15] Peter Bishop, Robin Bloomfield, Ilir Gashi, and Vladimir Stankovic. Diversity for security: a study with off-the-shelf antivirus engines. In *Proc. of the Int. Symp. on Software Reliability Engineering (ISSRE)*, pages 11–19, 2011.

[16] Peter Bishop, David G. Esp, Mel Barnes, Peter Humphreys, Gustav Dahll, and Jaakko Lahti. Pods – a project on diverse software. *IEEE Transactions on Software Engineering*, 12(9):929–940, 1986.

[17] Jean-Marie Borello, Eric Filiol, and Ludovic Mé. 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, 2010.

[18] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. In *In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.

[19] Juan Caballero, Theocharis Kampouris, Dawn Song, and Jia Wang. Would diversity really increase the robustness of the routing infrastructure against software defects? In *NDSS*, 2008.

[20] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 237–246. ACM, 2010.

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

[22] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical Report CS-02-197, Carnegie Mellon University, 2002.

[23] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. *ACM Sigplan Notices*, 46(2):13–22, 2011.

[24] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Addison-Wesley, 2002.

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

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

[27] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[28] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[29] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *Proc. of the Conf. on USENIX Security Symposium*, USENIX-SS'06, 2006.

[30] Marcus Aloizo Martinez De Aguiar, Michel Baranger, EM Baptestini, L Kaufman, and Y Bar-Yam. Global patterns of speciation and diversity. *Nature*, 460(7253):384–387, 2009.

[31] Yves Deswarte, Karama Kanoun, and Jean-Claude Laprie. Diversity against accidental and deliberate faults. In *Proceedings of the Conference on Computer Security, Dependability, and Assurance: From Needs to Solutions*, CSDA '98, pages 171–, Washington, DC, USA, 1998. IEEE Computer Society.

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

[33] Robert Feldt. Generating diverse software versions with genetic programming: an experimental study. *Software, IEE Proceedings*, 145(6):228–236, Dec 1998.

[34] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. "searching for cognitively diverse tests: Towards universal test diversity metrics". In *Proceedings of 1st Search-Based Software Testing Workshop (SBST'08)*, pages 178–186, 2008.

[35] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sérgio Soares, Fabiano Ferrari, Safoora Khan, Francisco Dantas, et al. Evolving software product lines with aspects. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 261–270. IEEE, 2008.

[36] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, pages 67–, Washington, DC, USA, 1997. IEEE Computer Society.

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

[38] Michael Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proc. of the workshop on New security paradigms*, pages 7–16. ACM, 2010.

[39] Mark Gabel and Zhendong Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 147–156. ACM, 2010.

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

[41] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault diversity among off-the-shelf sql database servers. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 389–398. IEEE, 2004.

[42] Ilir Gashi, Peter Popov, and Lorenzo Strigini. 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, 2007.

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

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

[45] Leslie Hatton. N-version design vs. one good version. *IEEE Software*, 14(6):71–76, 1997.

[46] Oystein Haugen, Birger Moller-Pedersen, Jon Oldevik, Gøran K Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 139–148. IEEE, 2008.

[47] Scott A Hendrickson and Andre van der Hoek. Modeling product line architectures through change sets and relationships. In *Proceedings of the 29th international conference on Software Engineering*, pages 189–198. IEEE Computer Society, 2007.

[48] M.A. Hiltunen, R.D. Schlichting, C.A. Ugarte, and G.T. Wong. Survivability through customization and adaptability: the cactus approach. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 1, pages 294 –307 vol.1, 2000.

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

[50] Todd Jackson. *On the Design, Implications, and Effects of Implementing Software Diversity for Security.* PhD thesis, University of California, Irvine, 2012.

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

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

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

[54] Jean-Marc Jézéquel. Reifying configuration management for object-oriented software. In *Proceedings of the 1998 International Conference on Software Engineering.*, pages 240–249, 1998.

[55] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 81–92. ACM, 2009.

[56] James E. Just and Mark Cornwell. Review and analysis of synthetic diversity for breaking monocultures. In *Proceedings of the 2004 ACM workshop on Rapid malcode*, WORM '04, pages 23–32, New York, NY, USA, 2004. ACM.

[57] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990.

[58] Karama Kanoun. Cost of software design diversity an empirical evaluation. In *Proc. of the Int. Symp. onSoftware Reliability Engineering (ISSRE)*, pages 242–247. IEEE, 1999.

[59] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

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

[61] Angelos Keromytis and Vassilis Prevelakis. A survey of randomization techniques against common mode attacks. Technical Report DU-CS-05-04, Drexel University, Department of Computer Science, 2005.

[62] John C. Knight. Diversity. In *Lecture Notes in Computer Science 6875*, 2011.

[63] John C. Knight, J.W. Davidson, D Evans, A Nguyen-Tuong, and C Wang. Genesis: A framework for achieving software component diversity. Technical report, DTIC Document, 2007.

[64] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming*. *IEEE Transactions on software engineering*, 1986.

[65] John C. Knight, Chenxi Wang, Kevin J Sullivan, and Matthew C Elder. Survivability architectures: Issues and approaches. In *DARPA Information Survivability Conference and Exposition,*, volume 2, page 1157. IEEE Computer Society, 2000.

[66] Philip Koopman and John DeVale. Comparing the robustness of posix operating systems. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 30–37. IEEE, 1999.

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

[68] Jörg Liebig, Sven Apel, Christian Lengauer, C Kästner, and Michael Schulze. 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*, volume 1, pages 105–114, 2010.

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

[70] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: a review. *ACM Computing Surveys (CSUR)*, 33(2):177–208, 2001.

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

[72] Bev Littlewood and John Rushby. 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, 2012.

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

[74] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Software self-healing using collaborative application communities. In *NDSS*, 2006.

[75] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[76] Kevin Shear McCann. The diversity–stability debate. *Nature*, 405(6783):228–233, 2000.

[77] Carlos J Melián, David Alonso, Diego P Vázquez, James Regetz, and Stefano Allesina. Frequency-dependent selection predicts patterns of radiations and biodiversity. *PLoS computational biology*, 6(8), 2010.

[78] Diego Mendez, Benoit Baudry, and Martin Monperrus. 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)*, 2013.

[79] Gert Merckxt. Software security through targeted diversification. Master's thesis, Master's thesis, Katholieke Universiteit Leuven, 2006.

[80] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1988.

[81] Subhasish Mitra, Nirmal R. Saxena, and Edward J. McCluskey. A design diversity metric and reliability analysis for redundant systems. In *Proc. of the Int. Test Conference, 1999.*, pages 662 –671, 1999.

[82] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Model driven engineering languages and systems*, pages 782–796. Springer, 2008.

[83] Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, and Jack W. Davidson. Security through redundant data diversity. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 187–196. IEEE, 2008.

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

[85] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Cloudav: N-version antivirus in the network cloud. In *USENIX Security Symposium*, pages 91–106, 2008.

[86] Adam J O'Donnell and Harish Sethu. On achieving software diversity for improved network security using distributed coloring algorithms. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 121–131. ACM, 2004.

[87] Nahmsuk Oh, Subhasish Mitra, and Edward J McCluskey. Ed4i: error detection by diverse data and duplicated instructions. *Computers, IEEE Transactions on*, 51(2):180–199, 2002.

[88] Derek Partridge and W Krzanowski. Software diversity: practical statistics for its measurement and exploitation. *Information and software technology*, 39(10):707–717, 1997.

[89] Klaus Pohl, Günter Böckle, and Franck Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer-Verlag New York Inc, 2005.

[90] Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. 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*, pages 41–50, 2012.

[91] Peter Popov and L. Strigini. The reliability of diverse systems: A contribution using modelling of the fault creation process. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 5–14. IEEE, 2001.

[92] Daryl Posnett, Raissa D'Souza, Premkumar Devanbu, and Vladimir Filkov. Dual ecological measures of focus in software development. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 452–461. IEEE, 2013.

[93] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443. Springer, 1997.

[94] Brian Randell. *System structure for software fault tolerance*. Springer, 1978.

[95] Kizito Salako and Lorenzo Strigini. When does "diversity" in development reduce common failures? insights from probabilistic modelling. *Dependable and Secure Computing, IEEE Transactions on*, PP(99):1–1, 2014.

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

[97] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 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*, pages 33–46. ACM, 2009.

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

[99] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond*, pages 77–91. Springer, 2010.

[100] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14:477–495, 2012.

[101] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. A comparison of decision modeling approaches in product lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, pages 119–126. ACM, 2011.

[102] Eric Schulte, , Zachary Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness and proactive diversity. Technical report, University of New Mexico, 2011.

[103] Eric Schulte, Zachary Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[104] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 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, pages 124–134, New York, NY, USA, 2011. ACM.

[105] Eric Totel, Frédéric Majorczyk, and Ludovic Me. Cots diversity based intrusion detection and application to web servers. In *Recent Advances in Intrusion Detection*, pages 43–62. Springer, 2006.

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

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

[108] Rob Van Ommering. Building product populations with software components. In *Proceedings of the 24th international conference on Software engineering*, pages 255–265. ACM, 2002.

[109] Bogdan Vasilescu, Alexander Serebrenik, Mathieu Goeminne, and Tom Mens. On the variation and specialisation of workload—a case study of the gnome ecosystem community. *Empirical Software Engineering*, pages 1–54, 2013.

[110] Chenxi Wang, Jack Davidson, Jonathan Hill, and John C. Knight. Protection of software-based survivability mechanisms. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN)*, pages 193–202. IEEE, 2001.

[111] Rong Wang, Feiyi Wang, and Gregory T Byrd. Design and implementation of acceptance monitor for building intrusion tolerant systems. *Software: Practice and Experience*, 33(14):1399–1417, 2003.

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

[113] Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security and Privacy*, 7(1):26–33, January 2009.

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

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

[116] Tewfic Ziadi, Loic Helouet, and Jean-Marc Jezequel. Revisiting statechart synthesis with an algebraic approach. In *Proceedings of the 26th International Conference on Software Engineering*, pages 242–251. IEEE Computer Society, 2004.