Multi-tier diversification in Web-based software applications

Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, Maxime Tricoire

Multi-tier diversification in Web-based software applications

Simon Allier [1] Olivier Barais [1,3] Benoit Baudry [1] Johann Bourcier [1,3]
Erwan Daubert [1] Franck Fleurey [4] Martin Monperrus [1,2] Hui Song [4]
Maxime Tricoire[1]

(1) Inria, France (2) University of Lille, France
(3) University of Rennes 1, France (4) Sintef, Norway

Reuse! Modularity! Among all computing domains, Internet Computing has long been an example of the application of those two software engineering mottos.

Web applications are formed from reusable components all over the software stack. Both on the client and the server side, web-specific libraries and frameworks enable creative developers to wrap up rich applications within very short time. Massive reuse happens both on the client side (on the browser) and on the server side (in the data-center). For instance, jQuery is a popular client-side library written in JavaScript, which drastically improves the development of responsive web applications. Spring is an example of a server-side technology which is used on millions of servers. The open-source philosophy and ecosystem is one of the backbones of this massive reuse in web applications.

In addition some very modular web applications have become extremely successful. For instance, according to our empirical data, more than 100k+ websites use Wordpress, a blogging and content management system (cf. Sidebar 2). The architecture of Wordpress is extremely modular. Every single point of the application can be tweaked or extended through "plugins" in an easy way. Combined with a high-quality documentation and a vibrant community, this results in thousands of plugins that can be instantly installed on any server running Wordpress.

Application monoculture

Reuse and modularity are key for liberating creativity and entrepreneurship in the Internet world. However, this bright world has a darker side. The problem is that they participate in creating a new form of massive-scale monoculture. This is what we discuss and address in this paper.

What is monoculture in computing? The concept of software monoculture refers to a computing environment that is largely dominated by the same software application [10]. For instance, the Windows operating system has long been considered as a monoculture on desktop machines. The term "monoculture" comes from agriculture, where it has been shown that exploiting the very same species on large areas is a bad practice. Similarly, "software monoculture" has a negative connotation [10]. The main problem of software monoculture are so-called BOBE attacks: break once, break everywhere. With a large monoculture, attackers can exploit flaws and common failure modes on a massive scale.

Operating systems (OS) and database monoculture is known for a long time [9], both are key components of the general computing infrastructure. However, Internet Computing has introduced a new kind of monoculture, which could be called "application monoculture". The novelty is that monoculture appears in application-level code (libraries, frameworks, the application itself), where developers rely on leaky abstractions and are more concerned about time-to-market and useful features for their clients than securing their code This monoculture might present even more risks than the monoculture of operating systems, which are developed over long periods of time, with security concerns in mind.

Let us consider the example of the content management application Wordpress. In the top 500.000 web sites [1], we found 106.412 sites running Wordpress. Among these sites, 65.558 (64%) use the Akismet plugin, which checks potential spams in Wordpress comments. 21.849 web sites (22,6%) use the Jetpack plugin, which has a known SQL injection vulnerability, even in the latest version. This demonstrates two levels of application monoculture: at the level of the application (Wordpress), and at the level of plugins. A single attack on a zero-day flaw on Wordpress is able to compromise thousands of websites.

On the one hand, reuse and modularity favor much

[0]BB, FF, MM wrote the paper; OB, BB, JB, FF designed the MDMS application; SA, OB, JB, ED, FF, HS, MT all participated in the development, the deployment and the diversification of the MDMS web application

[1]according to `http://www.alexa.com/`

writing the next killer application. On the other hand, reuse and modularity facilitates much the next massive BOBE attack. There is a tension between reuse and dependability. Should we give up one or the other? Not necessarily.

In this paper, we propose using software diversification in multiple components of web applications to reconcile those two aspects. We identify key enablers for the effective diversification of software, especially at the application-code level. Our vision is that it is possible to 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. We report on an innovative diversification experiment consisting in injecting multi-tier software diversity in a prototypical web application.

Multi-tier software diversity

A web application is typically composed of server-side and client-side code working in concert. The client side code is mostly written in JavaScript and runs in a browser. The server side of a web application is a software stack composed of an operating system and a web server, a set of libraries, frameworks, and application-specific code.

Software diversity has long been promoted to enhance software dependability. Since the seminal work of Cohen[3] and Forrest [5], there has been significant efforts towards automatic software diversification (see Sidebar 1 for more details). Current automated software diversification techniques operate at assembly-code level and are meant to mitigate memory safety vulnerabilities (e.g., stack overflow). Some of these techniques are successfully implemented in mainstream operating systems, making each operating system installation different from the others and thus mitigating the massive reuse of exploits. For example, address space layout randomization is implemented in all recent versions of Windows, Linux, Mac OS.

We believe that software diversification must address software layers beyond assembly code, to face the emergence of a new form of monoculture. The novelty of our proposal, with respect to the software diversity state of the art, is to diversify the application-level code (for example, diversify the business logics of the application), focusing on the technical layers found in web applications. In particular, web server deployment usually adopts a form of the *Reactor architecture pattern*, for scalability purposes: multiple copies of the server software stack, called request handlers, are deployed behind a load balancer, which dispatches all incoming requests. Currently all handlers are deployed as clones, but this

Sidebar 1: Software Diversification

Cohen [3] described 14 code diversification techniques to be combined for protecting operating systems. Forrest and colleagues [5] emphasized the need for building diverse computing systems and suggested diversification based on code manipulation. Since these seminal works, several approaches have implemented automatic diversification transformations, at machine-code level, and have combined them to increase the dependability of software systems. Each kind of transformation targets a specific kind of vulnerability, and several work have started to combine them to get a more complete protection. These 'integrated" software diversity techniques are particularly interesting with respect to multi-tier diversification. We now have a brief look at some of them and their defense objectives.

Bhatkar et al. [12] thwart code injection attacks by integrating various forms of randomization: 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. Jacob et al. [13] target tamper-resistance, through superoptimization to identify semantically equivalent instruction sequences, and other transformations that change the set of instructions and operands. The GENESIS project mitigates return-to-libc attacks and code injection by implementing a virtual machine that integrates calling sequence diversity and instruction set randomization through software dynamic translation [16]. Wang et al. [15] obfuscate critical software modules in survivability infrastructures by combining control-flow flattening and introduction of aliases.

For the readers interested in more references, we refer to the recent survey by Larsen and colleagues [14], and to the different levels of moving target defenses summarized by Okhravi and colleagues [8].

kind of architecture provides a natural setting for diversification.

We call multi-tier diversification the fact of diversifying several application software components simultaneously. This approach can rely on both *natural software diversity* and *automatic diversity*. Natural software diversity designates diverse software modules that provide equivalent functionalities but that are developed by different communities or companies. For example, there is a natural diversity of Java Virtual Machines. This can be exploited through the simultaneous deployment of some request handlers that run the IBM JVM and others that run the Oracle JVM. Application-level automatic diversity is provided by code transformations that generate diverse versions of some application components. Examples include method body intermixing [3], randomization of database query language [2] or our recent work on the synthesis of sosie programs (variants that are functionally similar but computationally diverse) [1].

"Multi-tier software diversification is a way to break the multi-level monoculture of web applications."

The diversification of application software code is expected to provide a diversity of failures and vulnerabilities in web server deployment. Thanks to the multiplicity of request handlers running in a web server, we can simultaneously deploy multiple combinations of diverse software components. Then, if one handler is hacked or crashes, the others should still be able to process client requests. The natural diversity of some components, exhibits diverse failure modes. For example there exist vulnerabilities that are present in the Oracle JVM only (CVE-2014-4244 and CVE-2014-2490). Meanwhile, automatic diversity can produce large quantities of local changes in the code, which affect specific kinds of vulnerabilities. For example method bodies merging changes the layout of the binary code, preventing the reuse of a single buffer overflow attack, or monkey patching; SQL queries randomization mitigates SQL injections; in the source code it is possible to diversify filesystem paths or urls to prevent certain forms of settings hijacking; sosies, which modify the flow of computation, can modify vulnerabilities such as lack of input validation or business logic vulnerabilities.

Proof-of-concept: multi-tier diversity on a blogging system

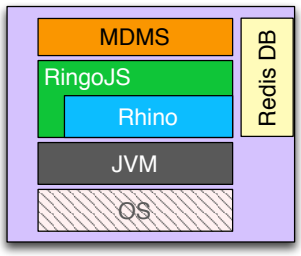
As a proof of concept, we have diversified a prototypical web-application that uses common, off-the-shelf components. The application is a multi-user blog-

Sidebar 2: Wordpress, a fragile monoculture?

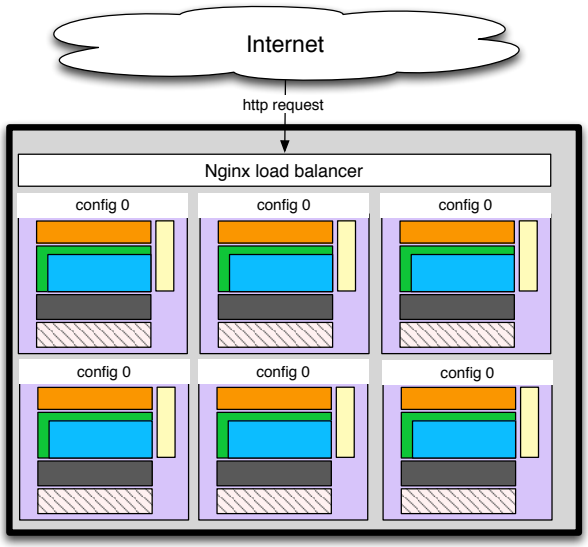
Wordpress is a Web content management system that supports massive customization through "plugins". While the vibrant Wordpress community keeps growing the number and the diversity of plugins for all possible tasks, we observe a paradoxical trend towards a monoculture of some popular plugins. This specific plugin monoculture is a potential threat for the Wordpress ecosystem, in particular when the plugins that are widely used have a defect. As an example of massive threat, we found that 22,6% out of a 100000 sites sample, use the Jetpack plugin that is known to be vulnerable to attacks. Jetpack provides users who deploy their own installation of Wordpress with features that are available on wordpress.com, such as advanced handling of social media or multimedia documents.

Yet, it is possible to exploit, the natural diversity of Wordpress plugins to mitigate this threat. For example, it is possible to replace Jetpack with wp-symposium or disqus, which offer compatible functionalities with a completely different implementation. This example shows that, despite the emergence of monocultures in the Wordpress ecosystem, the community actively continues to develop diverse solutions. This provides fertile ground to experiment with the automatic diversification of Wordpress sites, by exploiting the natural diversity of functionally similar plugins, to break this fragile application monoculture.

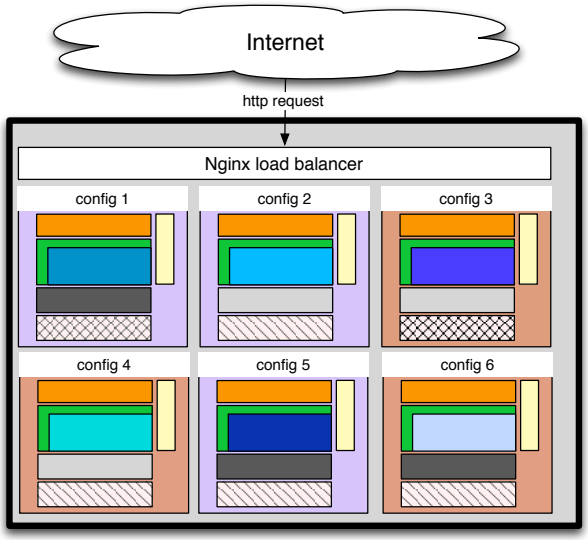
All results about the monoculture in Wordpress are publicly available [2]



(a) The MDMS software stack



(b) Monoculture deployment of MDMS



(c) Multi-diversified deployment of MDMS

Figure 1: The MDMS architecture: (a) the standard MDMS software stack; (b) homogeneous deployment of identical MDMS configurations; (c) multi-tier diversified deployment of different MDMS configurations

ging system called MDMS [3]. It allows users to view, create, edit and delete blog posts. The MDMS web application is implemented in JavaScript and runs on top of the RingoJS server-side framework. RingoJS is written in Java and complements the open-source JavaScript engine called Rhino. The data of the MDMS application is stored in a Redis distributed database. This application stack is illustrated in Figure 1a. This architecture reflects the characteristics of a large number of web applications (server side scripting language, server-side framework, NoSQL database).

In terms of deployment, we deploy MDMS following a typical Reactor pattern, to allow the application to elastically scale over time. MDMS is deployed on request handlers, which are arranged behind a load balancer. Following current practices, our initial deployment is as follows: all handlers provide the same RingoJS environment, running on top an identical Java virtual machine. This usual way of deploying the Reactor pattern results in a "non-diversified" deployment of the MDMS system, illustrated in Figure 1b. This is an example of what we call an "application monoculture": all servers behind the load balancer are clones, from the operating system level, up to the libraries and the application software.

Now, we inject diversity in the MDMS system at the following levels: deployment infrastructure, virtual machine, JVM and JavaScript library. Diversification at each level exploits either natural diversity or novel techniques for automatic diversity.

OS diversification: MDMS has no dependencies to any particular operating system and a mix of different distributions of Linux, BSD and Windows can be used. In our experiments, MDMS is randomly deployed on Windows and Linux.

JVM diversification: The next layer is the Java virtual machine. There are 3 major suppliers of Java virtual machines (IBM, Oracle and OpenJDK) with several versions for each of them. While the OpenJDK and the Oracle JVM share the same code base, the latter has some built-in commercial or open source tools that are not integrated in OpenJDK. These differences are significant enough to have vulnerabilities that are present only in the Oracle one. In our experiments, RingoJS alternatively runs on Oracle-jdk1.7.0_45 (Sun/Oracle), IBM-Java-x86-71 (IBM), and Java-7-openjdk (GNU).

Framework diversification: The next layer on the stack is the RingoJS framework. This is an open-source component for which there is no competing functionally equivalent alternatives. To diversify this level, we use our recent technique for the automatic synthesis of sosie programs [1]. A sosie is a vari-

[3] all the code for this experiment is available at `http://diversify-project.github.io/`

ant a program, which exhibits the same functionality (passes the same test suite) and a diverse computation (different control or data flow). Sosie synthesis is based on the transformation of the original program through statement deletion, addition or replacement. We synthesized 70 RingoJS sosies for MDMS using this new diversification technique.

Deployment diversification: Using the Cloud and its deployment interface, MDMS can be randomly deployed in 2 different data centers that are geographically distant (in Europe and in the US). In our experiments, we have used the Amazon cloud and a private cloud running LXC container. We use the CloudML (Cloud Modelling Language) platform [4] in order to model and automatically deploy the application among different Cloud provider and operating systems.

In total we have: 2 versions of the operating system; 3 diverse JVMs; 70 Rhino sosies; 2 datacenters. Since the layers are mostly independent from each others, those alternatives for each layer can be combined together to create an exponential number of possible server deployments. For example our setup has the potential to run 840 different diversified web servers for delivering the MDMS application.

Then, we deployed 17 instances of the server stack behind a Nginx web server. In particular, we deployed 17 diverse versions of the RingoJS library, a functional component that is never diversified in other diversification approaches. We designed 8 functional test scenario for validating the global functionality of MDMS. We checked that these scenario actually execute the diversified part of RingoJS, i.e., that the modified statements in sosie programs is executed. All scenario pass correctly on the original, non-diversified server, as well as on the diversified server. Yet, some of the modified statements in RingoJS (added, removed or replaced) are executed thousands of times (max is 148000 times for a sosie in the `setLP` method that is used by the JS interpreter to handle switches). The diversified system is available for use at `http://cloud.diversify-project.eu/`. This result demonstrates the ability to generate large quantities of software diversity by combining natural and automatic diversity in a realistic web-based architectural setting.

Insights from application diversity

This initial proof-of-concept demonstrates the feasibility of multi-tier diversification in web applications. It also illustrates the challenges to operationalize this vision. We discuss three of them here.

Automatic diversification. This is the key ingredient for a true diversity radiation. The main factor for the 840 versions of MDMS handlers is provided by the automatic synthesis of 70 sosies. However, there exist few solutions for the automatic diversification of application-level source code, because it requires transformations that are on the edge between functional correctness and quality of service [8]. The recent works on unsound program transformations open the way for such novel and massive diversification techniques. They include the proposals of Rinard et al. on loop perforation [7], Weimer et al.'s approach to code transformation for automatic patching [11], or our work on sosie synthesis [1].

Listing 1: Transformation example in one RingoJS sosie

```
/*The following snippet */
case JAVA_OBJECT_TYPE:
    return arg;
default:
    throw new IllegalArgumentException();


/*is replaced by*/
case JAVA_OBJECT_TYPE:
    if (arg != null)
      return arg;
default:
  throw new IllegalArgumentException();
```

To explain the potential effects of these transformations, we look at the RingoJS sosies. All of them exhibit a difference in the control or data flow, with respect to the original RingoJS. Looking in more details, we observe different reasons for this diversification. An attribute can be assigned a different value than in the original program, yet this side effect has no visible impact on the application; some method calls can be removed, removing a complete part of the program's computation, yet the program still provides the service. Both these cases are explained either because the changes have no side effect (e.g., the variable is never used), or, more interestingly, because they occur in parts of the computation that tolerate variations, which we call plastic zones. These plastic zones can appear in algorithms that compute some form of heuristic or in redundant code. Plastic zones are interesting from a security point of view because they indicate zones in which some potentially vulnerable code can be removed. Code replacement has a different impact. For example, Listing 1 illustrates a statement replacement, which ends up adding an input validation. In general, such reduction of the input space, if it still provides the service, is good for security. Eventually, we need to understand how to

navigate the functional neighborhood of programs in order to provide different degrees of application level diversity while maintaining functional compatibility.

Integrating multi-tier diversification in development processes. The second challenge of multi-tier diversification is about its integration in web software engineering practices, to master its impact and leverage the full potential for dependable web applications. First, we can deploy the diverse handlers in different ways. In our experiment we pick 17 different handlers out of 840 and deploy them, to provide a form of spatial diversity. Then the incoming requests are sent to one of the handlers, round-robin. Several other strategies are possible, for example: pick a single handler and deploy it 17 times; constantly deploy new versions of the handler, providing a temporal diversity to form a moving target defense [6]; use the diverse handlers in a multi-version fashion, with a voting mechanism. We now need to understand what strategy best fits a given dependability goal. Second, diversification has an impact on distribution and maintenance. For example, when the binary code of an application must be signed by a third-party, the production of millions of diverse variants becomes a challenge. One solution consists in diversifying the bytecode at installation time (after the signed version has been shipped), and have the transformation itself recognized as legitimate (and not detected as a malware). Another example, is dump trace analysis or incremental updates. This will require accurate traceability of variants and reversible code transformations, as well as new forms of code analysis for automatic patching.

Runtime environments. Our proof-of-concept implementation taught us that the integration of multiple levels of diversification poses several technical challenges. We need a reconfigurable load balancer, which can reason about the number of handlers it sends a request to, detect potentially vulnerable handlers and stop using them or decide to replace them; architectures that allow the composition of multiple levels of diversity; manage the application state consistency between the diverse handlers; support dynamic reconfiguration and deployment of software in a distributed infrastructure (VM, containers, software modules). The MDMS architecture has been designed to natively support multi-tier diversification. The diverse request handlers store data in a Redis database, a distributed nosql solution, instead of a file system. We have adapted the NGinx load balancer with specific distribution and recovery policies when one of the handlers fails. We experimented with both Kevoree [4] and CloudML [5] to manage the deployment of software modules on diverse and distributed virtual machines. Both frameworks provide utilities to seamlessly handle the heterogeneity of technologies for virtual machines (e.g., Vmware, VirtualBox), system containers (e.g.,docker, lxc, jails), app containers (e.g., servlet, android, osgi). They provide flexible configuration models with built-in architecture model exploration capabilities and the ability to orchestrate coherent and transactional reconfiguration of the platform, infrastructure and service levels. In the future, we plan to experiment also with Mesos to manage virtual machines or container deployment.

[4] `http://kevoree.org/`
[5] `http://cloudml.org/`

The end of monoculture?

We have emphasized the growing monoculture in web applications, which emerges from the success of a few frameworks and libraries. These reusable components are essential to support the engineering of large applications. Yet, they also come with potential drawbacks due to the massive distribution of bugs or vulnerabilities that is associated to monoculture. We propose to counter this phenomenon by extending software diversification beyond the machine-code level. The way to go is to diversify the different layers, up to the functional code. We have experimented with a realistic blog application to demonstrate the feasibility of multi-tier diversification. This experiment highlights the challenges that are ahead of software engineers if they want to systematically break the application monoculture of web applications. In particular, we believe that unsound program transformations open the way for the true explosion of application code diversity. May multi-tier diversification be the end of multi-tier monoculture!

Acknowledgement

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreements No. 600654, DIVERSIFY project, FP7-ICT-2011-9 and No. 611337, the HEADS project, FP7-ICT-2013-10.

References

[1] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2014.

[2] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. In

Proceedings of the Applied Cryptography and Network Security Conference, pages 292–302, 2004.

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

[4] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *Proceedings of the International Conference on Cloud Computing*, pages 887–894, 2013.

[5] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 67–72, Washington, DC, USA, 1997.

[6] Luanne Goldrich and Carl E. Landwehr. Moving target [guest editors' introduction]. *IEEE Security & Privacy*, 12(2):14–15, 2014.

[7] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *Proceedings of the International Conference on Software Engineering*, pages 25–34. ACM, 2010.

[8] Hamed Okhravi, Thomas Hobson, David Bigelow, and William Streilein. Finding focus in the blur of moving-target techniques. *IEEE Security &Privacy*, 12(2):16–26, Mar 2014.

[9] David Lorge Parnas. Which is riskier: OS diversity or OS monopoly? *Communications of the ACM*, 50(8):112, 2007.

[10] M. Stamp. Risks of monoculture. *Communications of the ACM*, 47(3):120, 2004.

[11] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering*, pages 364–374, 2009.

Sidebar references

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

[13] 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. 2008.

[14] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Automated software diversity. In *Proceedings of IEEE Security & Privacy*, 2014.

[15] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 193–202, 2001.

[16] 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.