Measuring the Occurrence of Security-Related Bugs through Software Evolution

Dimitris Mitropoulos
Department of Management Science and Technology
Athens University of Economics and Business
76 Patission Str, GR10434, Athens, Greece
dimitro@aueb.gr

Georgios Gousios
Department of Management Science and Technology
Athens University of Economics and Business
76 Patission Str, GR10434, Athens, Greece
gousiosg@aueb.gr

Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
76 Patission Str, GR10434, Athens, Greece
dds@aueb.gr

ABSTRACT

A security-related bug is a programming error that introduces a potentially exploitable weakness into a computer system. This weakness could lead to a security breach with unfortunate consequences. Version control systems provide an accurate historical record of the software code's evolution. In this paper we examine the frequency of the security-related bugs throughout the evolution of a software project by applying the FindBugs static analyzer on all versions of its revision history. We have applied our approach on four projects and we have come out with some interesting results including the fact that the number of the security-related bugs increase as the project evolves.

Keywords

Alitheia Core, FindBugs, Software Defects, Static Analysis, Software Evolution.

1. INTRODUCTION

The majority of software vulnerabilities derive from a small number of common programming errors [39, 25]. According to SANS (Security Leadership Essentials For Managers),[1] two software bugs alone were responsible for more than one and a half million security breaches during 2008. This is because most programmers have been trained in terms of writing code that implements the required functionality without considering its many security aspects [16, 18]. One of the most common approaches to identify software vulnerabilities is *static analysis*. This kind of analysis is performed by automated tools either on the program's source or object code and without actually executing it [9, 28]. Usually, such analysis takes place by security auditors at the end of, or during the development of the program.

[1] `http://www.sans.org/`

To manage large software projects, developers employ version control systems (VCS) like *Subversion*[2] and *Github*[3]. Such systems can provide project contributors with major advantages like: automatic backups, sharing on multiple computers, maintaining different versions and others. For every new contribution, which is known as a *commit*, a VCS system goes to a new state which is called a *revision*. Every revision stored in a repository represents the state of every single file at a specific point of time.

In this work we introduce a framework that examines how security-related bugs evolve into a software repository, through time. To achieve this we automatically analyze every revision of the project from its early revisions to the latest commits. Our framework combines *FindBugs*,[4] an effective static analysis tool that has already been used in research [6, 3, 15], and *Alitheia Core*, an extensible platform designed for performing large-scale software quality evaluation studies [13, 12]. To show how the number of bugs change through time, we have applied our framework to four different open source projects. Our initial observations set the basis for discussing issues that may improve vulnerability discovery models [27, 29, 35, 38], identify recurring vulnerabilities [8, 17]. Finally, we highlight security-related issues like the *domino effect* [34].

2. FRAMEWORK DESCRIPTION

Our framework includes a static analysis tool as a bug detector and a platform that provided us an efficient way to access different projects and their repositories.

2.1 FindBugs

FindBugs [14, 10] is an open source static analysis tool that searches for software bugs. It works by examining the compiled Java virtual machine bytecodes of the programs it checks, using the bytecode engineering library (BCEL) [11]. It supports plug-in bug detectors and it has an extensive mechanism for reporting errors, both through a GUI and by textual output. To detect a bug, FindBugs uses various formal methods. For example, to detect `null pointer bugs` it utilizes *control flow* and *data flow analysis*. It has also other

[2] `http://subversion.tigris.org/`
[3] `https://github.com/`
[4] `http://findbugs.sourceforge.net/`

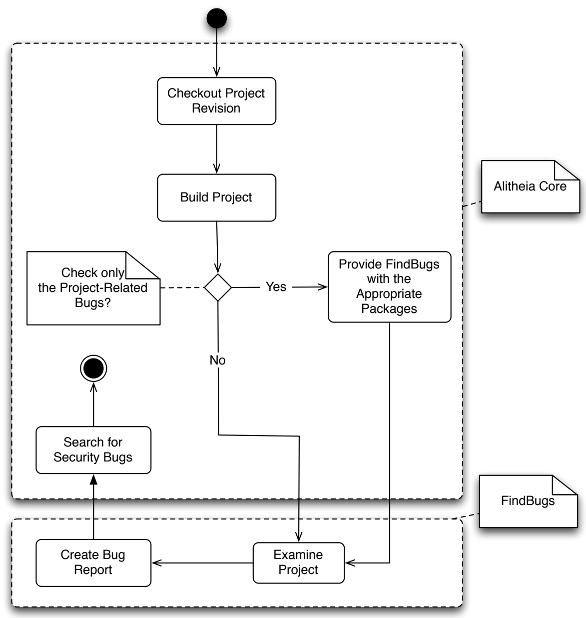


Figure 1: A state diagram indicating the steps taken by our framework.

detectors that employ *visitor patterns* over classes and methods by using *state machines* to reason about values stored in variables or on the stack. FindBugs warnings are grouped into *bug patterns* which in turn are grouped into *categories* such as correctness, malicious code vulnerability and bad practice. In our experiment we are interested only in two categories namely: *security* and *malicious code vulnerability*.

Findbugs has been used many times either for commercial or research needs. For instance, it was used to analyze all available builds of JDK [5] while Google has also incorporated it into its software development process [4, 32]. It has also been extended to verify API calls [33] and discover bugs in AspectJ applications [32].

2.2 Alitheia Core

Alitheia Core [13] is a platform designed for facilitating large scale quantitative software engineering studies. To do so, it preprocesses software repository data (both source code and also process artifacts, such as emails and bug reports) into an intermediate format that allows researchers to provide custom analysis tools. Alitheia Core automatically distributes the processing load on multiple processors while enabling both programmatic and REST API based access to the raw data, the metadata, and the analysis results. Alitheia Core is extensible through plug-ins, in both the analysis tool front and also the raw data access from. A wealth of services, notably a metadata schema and automated tool invocation, is offered to analysis tool writers by the platform.

To analyse a project, Alitheia Core needs a local mirror of the project's source code, mailing list and bug repository. The analysis itself is split in pre-defined phases (e.g. data extraction, data inference, metric extraction etc), during which Alitheia Core automatically applies a set of pre-defined data extraction and analysis plug-ins. At the end of the process, the researcher can either query the results database directly or browse the results using a simple web based interface.

2.3 Integration

To integrate FindBugs with Alitheia Core we have created a new Alitheia Core metric plug-in that works in the following steps (Figure 1 depicts these steps as a UML state diagram): for every project and every revision of this project, the metric creates a build. Then it invokes FindBugs to examine this build and create an analysis report. A user can select whether to examine the project alone or the project together with its dependencies. Finally, from this report, it retrieves the security-related bugs and updates the database. Figure 2 presents how the two components are integrated.

Building a software project is a multistep process that involves discovering and downloading the project dependencies, invoking the project's build script and retrieving the build artifacts. To automatate some of these tasks, modern build systems such as Maven[5] include support for resolving and downloading dependencies declared in the project's build file, while they also follow a standard directory structure for code and build artifacts. The Findbugs plug-in exploits the conventions supported by Maven to automatically build each project and retrieve the generated bytecode archives. For example, it knows that source code is placed into the `src/main/java` directory, while build artifacts are placed under `target/`. It is therefore sufficient to walk the directory structure and find the bytecode archives (`jar` files) in order to retrieve the project's (or any sub-project's) package structure and compiled code, respectively.

[5] `http://maven.apache.org/`

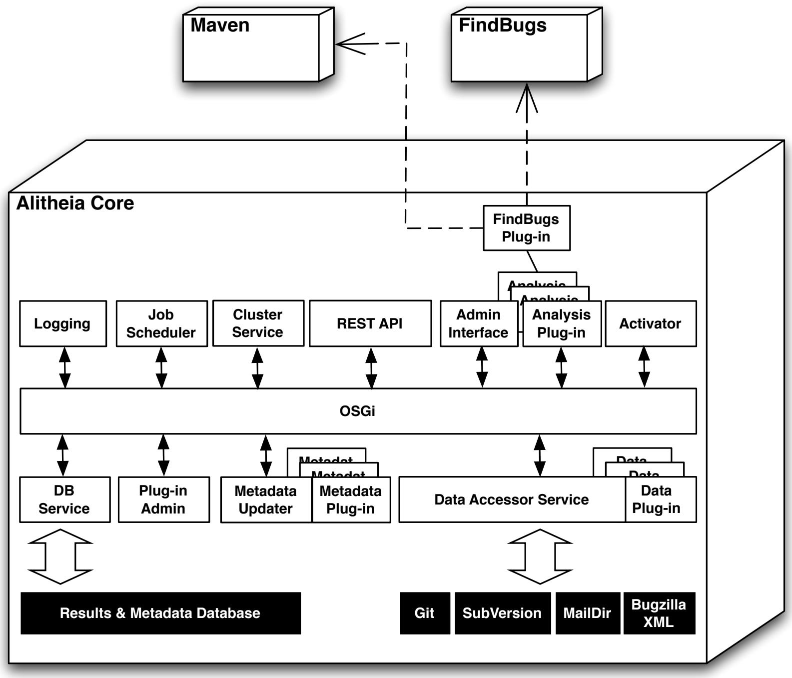


Figure 2: Alitheia Core and FindBugs integration.

After a build the Findbugs binary is invoked. In order to examine the bytecode that is created by the sources that belong to the specified project and not by its dependencies we collect all the corresponding project packages and then we use the `-onlyAnalyze` option of FindBugs to pass them as one parameter. By using the `-xml` option the report that is made contains all the bug descriptions in an XML format. As a result, we can easily parse this report in order to collect the bugs that we are interested in. The bugs are then associated with file revision information that Alitheia Core stores in its database, through path name matching and thus results can be stored with respect to each file version. To speed up searches, the Findbugs plug-in also stores summaries of number of incidents found per project version.

3. INITIAL RESULTS

We have examined four open source projects that are based on the Maven build system namely: *xlite*,[6] *sxc*,[7] *javancss*,[8] and *grepo*.[9] Our experiment included two measurements. First, for every revision, we applied FindBugs only to the bytecode of a specified project. Then for our second measurement, we also included the dependencies of this project. Figure 3 depicts the results of both measurements for every project.

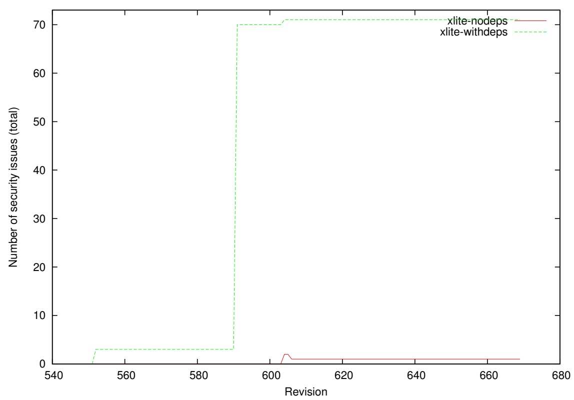
The most interesting observation that we can make is that the security bugs are increasing as projects evolve. This is particularly noteworthy and shows that bugs should be fixed in time to decrease the effort and cost of the security audits after the end of the development process. Another observation is the existance of the *domino effect*. The usage of external libraries introduces new bugs. As we can see in all cases the sum of the security related bugs in the second measurement is bigger or equal than the first one. A mathematical representation of the this could be the following: If *bop* is the variable that represents the sum of the security related bugs of every project for every revision, *boa* the sum of the bugs that also concern the dependencies of the project for every revision and i is the number of a project revision, the following expression stands for every project:

$$\sum_{i=0}^{n} boa \geq \sum_{i=0}^{n} bop \quad (1)$$

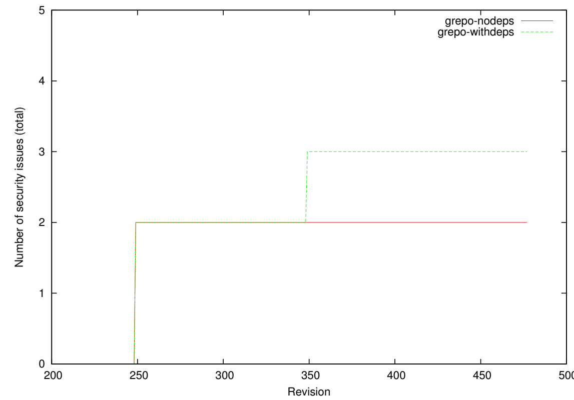
There are also cases where there is no security bug in the majority of the revisions of a project but there are bugs in the libraries that it includes i.e. in the *javancss* project. Still, there is a case (the *sxc* project) where there are no bugs in the libraries that the project depends on.

The Alitheia Core framework provides ways to check what changes have been made after a commit. By taking advantage of this feature we observe that there are situations where the total bugs are increased after the addition of a new library. For instance, in the 591st revision of the *xlite* project, the 349th revision of the *grepo* project and the 30th revision of the *javancss* project, developers have added new libraries in their project. On the other hand, the number of bugs decreases when developers update a library (for ex-

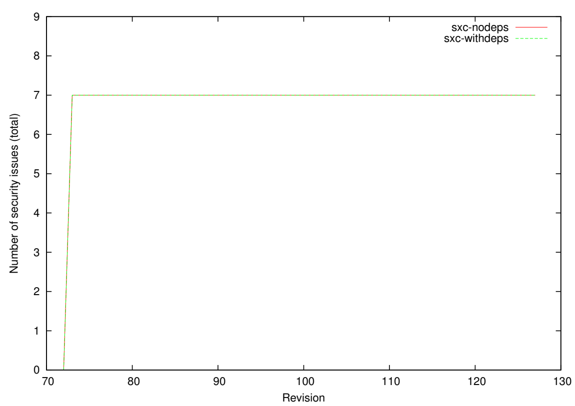
[6]`http://xircles.codehaus.org/projects/xlite`
[7]`http://xircles.codehaus.org/projects/sxc`
[8]`http://xircles.codehaus.org/projects/javancss`
[9]`http://xircles.codehaus.org/projects/grepo`



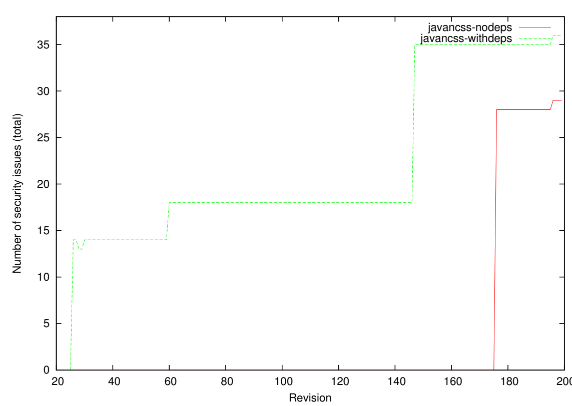
(a) xlite



(b) grepo



(c) sxc



(d) javancss

Figure 3: Bug frequency for all four projects.

ample in the 30th revision of the *javancss* project). Thus, project libraries should always be updated not only because of the additional functionality that they provide, but also for security reasons. In general we can observe how the changes made in third-party software can affect the evolution of a project while its developers are unaware of that.

Another interesting issue regards the bugs themselves. Table 3, shows for the last revision of every project the security bugs that have been found. As we can see, even after hundreds of revisions there are trivial bugs but there are also bugs that could be severe for the application. For instance, the last revision of *javancss* includes a code fragment that creates an SQL prepared statement from a non-constant string. If this string is not checked properly, an SQL injection attack is prominent.

By determining the occurance of a bug for a large number of projects, and by examining all revisions, we could generate the frequency of the appearance of this bug. Such an estimation could be crucial for vulnerability discovery models.

4. RELATED WORK

There are many approaches introduced by the research community that are used to extract conclusions by observing the history, or the changes of sotware repositories. Such conclusions concern the evolution of a software project, the identification of programming errors between revisions, the impact of a change on the whole project, the prediction of bugs and providing the developers with useful data.

One of the first approaches to be introduced, involves a system called CocR, that analyzes the whole history of the software repository of a project in order to provide the developers with data in an efficient way [24]. Specifically, the analysis includes the history of every developer, the creation of a graph of mails among developers and the usage of a backtracking system that keeps track of the various requests for changes. By analyzing this elements, CocR can search for specific code fragments, introduced by the same contributor and in a specific period of time. A similar approach called *history slicing*, involves the generation of a graph that links every line of code in a repository, with its corresponding previous revision [31]. By utilizing this graph, a contributor can locate specific versions that contain changes for the lines of code of his interest and their exact details (including contributor, filenames, and others). To help find the right person to resolve a bug report, an approach that incorporates a machine learning algorithm has been introduced [2]. First, this algorithm is applied to the bug reports that appear in the repository and when a new report arrives, the classifier that is produced by the algorithm suggests specific developers that can resolve the report. Furthermore, an approach that observes the API-level refactorings through the evolution of large projects has shown that there a is an increase of bug fixes after a refactoring [19]. Also the time taken to fix a bug after a refactoring is smaller than before.

Some approaches involve the detection of the variations between revisions. In particular, Sieve [30] is an automated tool, that is based on *impact analysis* [21] to test if the changes introduced in a new revision comply to the invariants assumed in the previous one. Another similar tool called PARCS (performance-aware revision control support) uses *calling context tree* (CCT) [1] profiling and performance differencing [40] to provide feedback to the project developers. This feedback concerns how the changes after a commit affect the performance and behavior of the whole application. In addition, a technique called *change classification*, has been introduced for predicting bugs for every revision [20]. To detect a bug the technique builds classification models by extracting specific features (log messages, reports and others) from the history of the repository by facilitating another tool called Kenyon [7]. Then, for every new contibution, it compares the commited code to the trained model to check for existing bugs.

Apart from bug detectors that act between revisions, there are others based on repository mining. Menzies et al. [26] base their approach on using techniques like *data mining* and static analysis to detect bugs in large repositories. Dynamine [22] is a tool that combines software repository mining and dynamic analysis to discover common use patterns and code patterns that are likely errors in Java applications. In a similar way, PR-Miner mines common call sequences from a code snapshot and then marks all non-common call patterns as potential bugs [22]. A method to examine source code change history minning is also used for bug detection [37]. This method involves a static checker that searches for commonly fixed bugs and at the same time it utilizes information mined directly from the project repositories to refine its results.

Our work partially differs from the bug detection approaches since we are not aiming to only provide this functionality. We also want to provide an automated way to show the frequency of security-related bugs during the software development process and either provide valuable information to the developers of a project or assist the project planning of a new one.

5. DISCUSSION AND FUTURE WORK

Observing the changes and history of the software development environment has provided the research community with many useful inductions. In this paper we provided initial results concerning the appearance of security bugs through the evolution of a software project. To achive this we have combined two tools that have been previously used in research. Our experiment included four maven-based open source projects. Our preliminary observation had to do with the increase of the bugs as the project evolves. Other ovservations included the existance of the domino effect and the dependence of a software project from its libraries. No matter how well a programmer secures a software component it won't matter if she is using another library with existing vulnerabilities. In addition, programmers should use the latest versions of the libraries that their project depends on. Finally, measuring the occurance of a security bug through the revisions could lead to useful input for defect identification models.

Even if we used one static analysis tool in our approach, the key idea behind our framework is to combine more tools in order to have more substantial results. Currently, there are numerous tools that analyze Java code and could be easily imported to Alitheia Core for our purposes [23, 36].

Project Name	Bug Description (taken from the FindBugs website)	Occurance
javancss	Dm: Hardcoded constant database password	2
javancss	EI: May expose internal representation by returning reference to mutable object	8
javancss	MS: Field isn't final and can't be protected from malicious code	3
javancss	MS: Field should be moved out of an interface and made package protected	4
javancss	MS: Field should be package protected	14
javancss	MS: Field isn't final but should be	4
javancss	SQL: A prepared statement is generated from a nonconstant String	1
sxc	EI: May expose internal representation by returning reference to mutable object	7
xlite	MS: Field should be both final and package protected	1
xlite	EI: May expose internal representation by returning reference to mutable object	8
xlite	MS: Public static method may expose internal representation by returning array	1
xlite	MS: Field should be package protected	1
xlite	MS: Field isn't final but should be	60
grepo	EI: May expose internal representation by returning reference to mutable object	5

Table 1: Occurences of security bugs in the last revision of every project.

In addition, using FindBugs raises restrictions in the automation of the process since FindBugs runs on bytecode. Hence our projects should be based on a build system that allows automated builds and keep a standard directory structure for code and build artifacts. Using static tools that run over source code should allow us to run our framework on more projects and enrich our results.

By running our framework on more projects we could validate the statistical significance of our results and draw even more conclusions like: finding overlapping vulnerable dependencies, if there is a correlation between the lines of code and the security bugs of a project and others.

6. ACKNOWLEDGEMENTS

The project is being co-financed by the European Regional Development Fund (ERDF) and national funds and is a part of the Operational Programme "Competitiveness & Entrepreneurship" (OPCE II), Measure "COOPERATION" (Action I).

7. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32:85–96, May 1997.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM.

[3] N. Ayewah and W. Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*, DEFECTS '08, pages 1–5, New York, NY, USA, 2008. ACM.

[4] N. Ayewah and W. Pugh. The google findbugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010. ACM.

[5] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM.

[6] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Using findbugs on production software. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 805–806, New York, NY, USA, 2007. ACM.

[7] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. *SIGSOFT Softw. Eng. Notes*, 30:177–186, September 2005.

[8] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '10, pages 105–114, New York, NY, USA, 2010. ACM.

[9] B. Chess and J. West. *Secure programming with static analysis.* Addison-Wesley Professional, 2007.

[10] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens. Improving your software using static analysis to find bugs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 673–674, New York, NY, USA, 2006. ACM.

[11] M. Dahm. Byte code engineering. In *IN JAVA-INFORMATIONS-TAGE*, pages 267–277. Springer-Verlag, 1999.

[12] G. Gousios and D. Spinellis. Alitheia core: An extensible software quality monitoring platform. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 579–582, Washington, DC, USA, 2009. IEEE Computer Society.

[13] G. Gousios and D. Spinellis. A platform for software engineering research. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 31–40, Washington, DC, USA, 2009. IEEE Computer Society.

[14] D. Hovemeyer and W. Pugh. Finding bugs is easy.

SIGPLAN Not., 39:92–106, December 2004.

[15] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '07, pages 9–14, New York, NY, USA, 2007. ACM.

[16] M. Howard and D. LeBlanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, second edition, 2003.

[17] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 91–104, New York, NY, USA, 2005. ACM.

[18] S. K. Katsikas and D. Gritzalis. *Information systems security: facing the information society of the 21st century*. Chapman & Hall, Ltd., London, UK, UK, 1996.

[19] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA, 2011. ACM.

[20] S. Kim, E. J. Whitehead, Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, 34:181–196, March 2008.

[21] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 308–318, Washington, DC, USA, 2003. IEEE Computer Society.

[22] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[23] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[24] M. Matsushita, K. Sasaki, and K. Inoue. Coxr: Open source development history search system. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 821–826, Washington, DC, USA, 2005. IEEE Computer Society.

[25] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[26] T. Menzies, J. S. D. Stefano, C. Cunanan, and R. M. Chapman. Mining repositories to assist in project planning and resource allocation. In *Proceedings of the 1rst Workshop on Mining Software Repositories*, 2004.

[27] S. Neuhaus and T. Zimmermann. The beauty and the beast: vulnerabilities in red hat’s packages. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 30–30, Berkeley, CA, USA, 2009. USENIX Association.

[28] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black. Effect of static analysis tools on software security: preliminary investigation. In *Proceedings of the 2007 ACM workshop on Quality of protection*, QoP '07, pages 1–5, New York, NY, USA, 2007. ACM.

[29] A. Ozment. Improving vulnerability discovery models. In *Proceedings of the 2007 ACM workshop on Quality of protection*, QoP '07, pages 6–11, New York, NY, USA, 2007. ACM.

[30] M. K. Ramanathan, A. Grama, and S. Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 241–252, Washington, DC, USA, 2006. IEEE Computer Society.

[31] F. Servant and J. A. Jones. History slicing. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 452–455, 2011.

[32] H. Shen, S. Zhang, J. Zhao, J. Fang, and S. Yao. Xfindbugs: extended findbugs for aspectj. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '08, pages 70–76, New York, NY, USA, 2008. ACM.

[33] D. Spinellis and P. Louridas. A framework for the static verification of api calls. *J. Syst. Softw.*, 80:1156–1168, July 2007.

[34] J.-E. J. Tevis and J. A. Hamilton. Methods for the prevention, detection and removal of software security vulnerabilities. In *Proceedings of the 42nd annual Southeast regional conference*, ACM-SE 42, pages 197–202, New York, NY, USA, 2004. ACM.

[35] Y. Wang, W. M. Lively, and D. B. Simmons. Software security analysis and assessment model for the web-based applications. *J. Comp. Methods in Sci. and Eng.*, 9:179–189, April 2009.

[36] G. Wassermann, C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. *ACM Trans. Softw. Eng. Methodol.*, 16, September 2007.

[37] C. Williams and J. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, June 2005.

[38] Y. Wu, R. A. Gandhi, and H. Siy. Using semantic templates to study vulnerabilities recorded in large software repositories. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, pages 22–28, New York, NY, USA, 2010. ACM.

[39] G. Wurster and P. C. van Oorschot. The developer is the enemy. In *NSPW '08: Proceedings of the 2008 workshop on New security paradigms*, pages 89–97, New York, NY, USA, 2008. ACM.

[40] X. Zhuang, S. Kim, M. i. Serrano, and J.-D. Choi. Perfdiff: a framework for performance difference analysis in a virtual machine environment. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 4–13, New York, NY, USA, 2008. ACM.