Securing Legacy Code with the TRACER Platform

Kostantinos Stroggylos[*] Dimitris Mitropoulos[*] Zacharias Tzermias[†]
Panagiotis Papadopoulos[†] Fotios Rafailidis[‡] Diomidis Spinellis[*] Sotiris Ioannidis[†]
Panagiotis Katsaros[‡]

[*]Department of Management Science and Technology
[*]Athens University of Economics and Business
{circular,dimitro,dds}@aueb.gr

[†]Institute of Computer Science
[†]Foundation for Research and Technology - Hellas
{tzermias, panpap, sotiris}@ics.forth.gr

[‡]Department of Informatics
[‡]Aristotle University of Thessaloniki
katsaros@csd.auth.gr, frafaili@yahoo.gr

ABSTRACT

Software vulnerabilities can severely affect an organization's infrastructure and cause significant financial damage to it. A number of tools and techniques are available for performing vulnerability detection in software written in various programming platforms, in a pursuit to mitigate such defects. However, since the requirements for running such tools and the formats in which they store and present their results vary wildly, it is difficult to utilize many of them in the scope of a project. By simplifying the process of running a variety of vulnerability detectors and collecting their results in an efficient, automated manner during development, the task of tracking security defects throughout the evolution history of software projects is bolstered. In this paper we present TRACER, a software framework and platform to support the development of more secure applications by constantly monitoring software projects for vulnerabilities. The platform allows the easy integration of existing tools that statically detect software vulnerabilities and promotes their use during software development and maintenance. To demonstrate the efficiency and usability of the platform, we integrated two popular static analysis tools, *FindBugs* and *Frama-C* as sample implementations, and report on preliminary results from their use.

Keywords

Static Analysis, Software Security, Trusted Applications, Legacy software.

1. INTRODUCTION AND MOTIVATION

A security vulnerability is a programming error that introduces a potentially exploitable weakness into a computer system [1]. Such defects can severely affect an organization's infrastructure [2], and may cause significant financial damage to an organization [3]. Whereas a software bug can cause a software artifact to fail, a security bug can allow a malicious user to alter the execution of the entire application for his or her own gain. Such defects could give rise to a wide range of security and privacy issues, icluding access to sensitive information, destruction or modification of data, and denial of service. Moreover, security issue disclosures can create negative publicity and decrease the market value of a software vendor.

Software vulnerabilities may or may not depend on the programming language used to develop an application. The former set involves the infamous buffer overflow attacks.Such defects are possible due to the lack of memory-safety mechanisms in some programming languages (most notably C and C++), and do not appear in all kinds of applications. The latter set involves attacks that can be performed against numerous applications regardless of the programming language used, and by utilizing different attack techniques [4].

One of the most common approaches to identify software vulnerabilities is *static analysis* [5]. This kind of analysis is performed by automated tools either on the program's source or object code and without actually executing it [6]. Usually, such analysis takes place by security auditors at the end of, or during the development of the program. Since the requirements for running such tools and the formats in which they store and present their results vary wildly, it is inherently difficult to utilize a number of them effectively on a software project.

In this paper we present TRACER, a framework and software platform to support the ongoing maintenance and secure development of applications. TRACER simplifies this process by providing a platform to run such tools in an automated manner. Moreover, by using a common representation for metrics and results regardless of the tool that was used to generate them, it enables their analysis, presentation and visualization in a homogenous way.

TRACER monitors multiple data sources associated with the develoment of a software project, such as the source code repository and bug management tracking system, and automatically analyzes each revision. Therefore it can be used to track security defects throughout the evolution [7, 8] of a project. Such observations can set the basis for discussing issues that may improve vulnerability discovery models [9] and identify recurring patterns of vulnerabilities among projects [10]. In this way we can improve the security and reliability of a system, which are two of the main pillars of a trusted application.

2. RELATED WORK

There are numerous efforts that employ different tools and methodologies, in order to help programmers secure their applications or provide the research community with results regarding the evolution of security bugs. TRACER differs from most of the existing plain bug detection frameworks, since its scope includes an open, automated way to distinguish security bugs and their frequency of occurrence during the software development process. In this manner, our platform helps programmers to better understand which defects are most common in their software and as a result to improve their development skills and build more trusted applications.

2.1 Software Quality

There are some frameworks that are similar to TRACER, but they focus more on software quality rather than software security. For instance, the FLOSSM*etrics project* [11] (Free/Libre Open Source Software Metrics) aims to construct, publish and analyse a large scale database with information and metrics about libre software development coming from several thousands of software projects, using existing methodologies, and tools already developed. To achieve this the project integrates already available tools to extract and process results about identification of best practices, productivity measurement and others.

Hipikat [12], is an Eclipse plug-in that collects data from repositories, task reports, mailing lists and bug databases to suggest project related software development artifacts (e.g. similar changes, related email discussions, relevant project Web pages) that a developer could consider during important architecture decisions. Hipikat though, is a purpose specific system that was designed to be an automated store of project memory. In contrast, TRACER offers more generic abstractions of the underlying data sources and defers processing to external tools (plug-ins).

Hackystat [13] is an open source framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data. In particular, it uses telemetry data in order to improve software development management. Hackystat users are able to attach software "sensors" to their development tools, which unobtrusively collect and send "raw" data about development to a web service. TRACER is similar to Hackystat since it can also process data from both the software project and the development process. Moreover, TRACER is more flexible than Hackystat as it does not require any changes to the project's configuration or the developer's toolset.

SonarQube (formerly *Sonar*) [14] is an open source web platform that provides code quality measurements, reviews and remediations. It uses various tools (including Findbugs) to measure code for bugs and possible violation of code style policies. SonarQube supports more than 20 different languages, including Java, C# and others, and allows users to add their own rules on those languages. Finally, like TRACER, it is extendable through a library of plug-ins.

2.2 Software Security

Hakiri[1] is a cloud security platform that examines GitHub repository branches in order to measure the occurrence of security vulnerabilities in Ruby on Rails projects. Such vulnerabilities include SQL injection [4] and others.

The *Parfait* [15] [16] framework developed by Sun Labs, is a bug detection framework for C and C++ code, designed for scalability and precision [17]. It is built on top of the LLVM framework [18], a low-level virtual machine for various languages. It also uses *BegBunch* [19], a bug benchmarking suite that contains existing synthetic benchmarks. Parfait is able to classify the bug types in real bugs, no bugs and potential bugs. Parfait also aims to identify security vulnerabilities using taint analysis during an additional pre-processing step. They also evaluate their approach on a subset of the SAMATE[2] benchmarks related to buffer overflow anomalies.

The commercial *Klockwork suite*[3] provides code refactoring, reporting and metrics, source code analysis, and code review for C, C++, Java and C# programs. It builds a data repository that has several different uses such as metrics generation, enforcement of architectural constraints and on-the-fly detection of potential faults and security issues.

3. FRAMEWORK DESIGN

TRACER is a platform aimed at identifying security vulnerabilities in software systems. Instead of designing and implementing yet another such platform from the ground up, we selected to build TRACER on top of the open source *Alitheia Core* platform. A high-level representation of the TRACER platform can be seen in Figure 1.

3.1 The Alitheia Core platform

Alitheia Core [20] is a platform designed for facilitating large scale quantitative software engineering studies. It employs an extensible, service-oriented architecture to preprocess software repository data (both source code and development process artifacts, such as emails and bug reports) into an intermediate format that allows researchers to develop custom analysis tools. It is based on a three-tier architecture that automatically distributes the processing load on multiple processors or cluster computing nodes, while enabling both programmatic and REST API based access to the raw data, the metadata, and the analysis results. The processing core is modeled on a system bus architecture based on OSGi[4], with services that are attached to the bus and are accessed via a service interface. Extensibility is available through plug-ins for data analysis and raw data access support. 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 processes a local mirror of the project's source code repository, mailing list archive and bug tracking database. The analysis itself is split in predefined phases (e.g. data extraction, data inference, metric extraction etc), during which a set of pre-defined data extraction and analysis plug-ins are automatically applied. During this phase project metadata is collected and stored in the system's database using an internal representation that

[1] `https://hakiri.io`

[2] `http://samate.nist.gov`

[3] `http://www.klocwork.com/products/insight/`

[4] `http://www.osgi.org/`

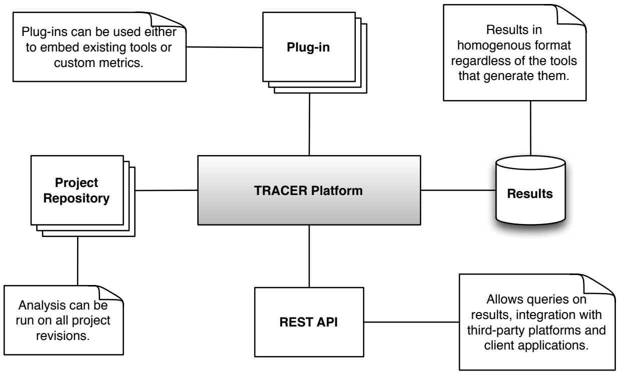

Plug-ins can be used either to embed existing tools or custom metrics.
Plug-in
Results in homogenous format regardless of the tools that generate them.
Project Repository
TRACER Platform
Results
Analysis can be run on all project revisions.
REST API
Allows queries on results, integration with third-party platforms and client applications.

Figure 1: TRACER high level architecture.

is used throughout the system. This information can be used to promptly respond to queries relating to the properties of the resources examined by the various plug-ins, while the original format of all resources are still available in the raw data stores. 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. This data is also exposed via a REST interface that can be used to perform similar queries.

Since the platform operates in an OSGi container environment, each plug-in is a self-contained OSGi bundle. This way plug-ins can define dependencies on the services provided by each one and can share dependencies to third party libraries, in order to minimize code duplication and promote code reuse.

Alitheia Core metric plug-ins implement a common interface - in fact all metric plug-ins inherit from an abstract implementation and only need to provide implementations of a minimal number of methods. Each plug-in is associated with a set of activation types, which indicate that the plug-in must be activated in response to a change to a project resource. It also can calculate several different metrics, each associated to the scope of its applicability, i.e. a scope that defines the types of project resources it can be calculated against. Metric plug-ins can access the services provided by the Alitheia Core components to retrieve raw data, or results from other plug-ins required for their own operation.

3.2 The TRACER platform design and implementation

While Alitheia Core aims to allow for efficient estimation of the quality of software projects in general, TRACER was designed with a focus on software security. Its primary objective is to provide a platform for continuously monitoring the development of software projects, in order to support the timely identification of security vulnerabilities. Since a significant part of the infrastructure required for supporting this operation was already implemented in Alitheia Core, TRACER was designed to leverage the already existing functionality it provided. To support the specific objectives of TRACER, a set of new components providing extensions to existing ones was added at each level of the Alitheia Core architecture.

The integration of third party tools for static analysis or other techniques of vulnerability detection is implemented by building on top of the plug-in infrastructure offered by the Alitheia Core platform. One of the major advantages of the framework is the independence from the programming language used to develop the analyzed projects. Projects that are implemented on diverse programming languages can be tested for software bugs and vulnerabilities, provided that analysis tools for the respective languages have been incorporated into the platform. The platform is also independent of the programming languages used to develop the third party tools that detect software vulnerabilities.

3.2.1 Data model for software vulnerability representation

All TRACER functionality is based upon a model for representing and persisting information about software vulnerabilities. A scheme based on roles and privileges is used to allow the detection of vulnerabilities on specific software artifacts, or specific vulnerability reporting in a multi-user and multi-project environment. The following are the core entities used for representing software vulnerabilities:

Vulnerability Type A vulnerability type describes the category in which a detected software vulnerability can be classified

Security Profile A security profile is a logical grouping of vulnerability types, enabling the end user of the platform to perform a detection of specific types of vulnerabilities

Vulnerability A vulnerability represents a specific software security vulnerability of a given type, detected on a given software project version and / or artifact (e.g. file, module, method).

This scheme is simple enough to support the easy development of vulnerability detectors. At the same time, when combined with the data model provided by Alitheia Core for storing metrics, it can cater for complex scenarios, reporting at various levels of granularity and detailed analyses.

3.2.2 Support for software vulnerability detection

There are a number of software vulnerability detection techniques and tools available, as presented in Section 2. Reimplementing them as part of the TRACER platform would be an exercise in futility, since each one has different operating requirements and makes different assumptions. Moreover, by simplifying the integration of third party tools for vulnerability detection, a higher level of expandability of the platform can be achieved.

In most cases, the detection of vulnerabilities on a software artifact involves only two steps: invoking an external tool created for this purpose with specific arguments as required, and evaluating the results returned by the tool. Typically these operations can be further simplified by writing custom tools or scripts to handle the tedious parts of the process. Therefore, the integration of such external tools in TRACER can be supported by implementing simple driver plug-ins that only need to implement these two steps and store the results using the data model provided by the platform.

Such an external tool driver is called a vulnerability detector plug-in, and it leverages the plug-in mechanism provided by Alitheia Core to handle automatic activation, as well as storage and retrieval of results. Each vulnerability detector

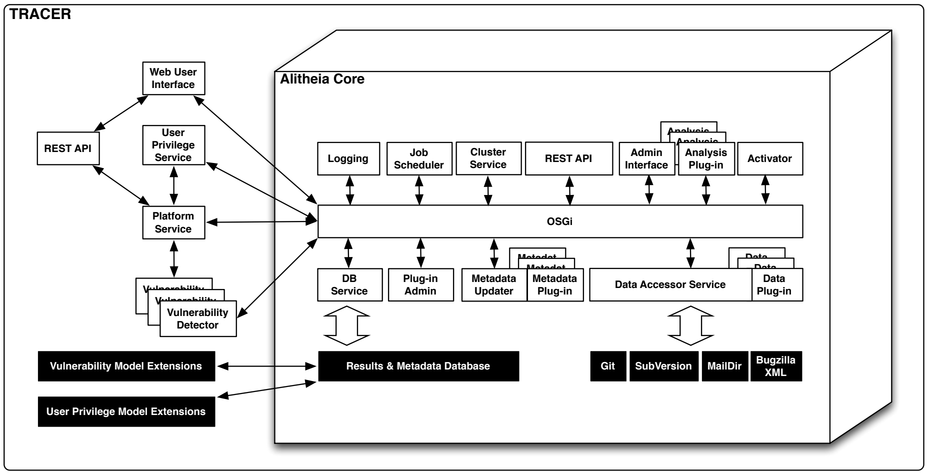


Figure 2: TRACER extensions on the Alitheia Core architecture.

is associated with a set of vulnerability types that it can detect, so that the platform may automatically trigger its execution when needing to check if a software project or artifact is vulnerable to a specific type of attacks. Moreover, each vulnerability detector is associated to the types of different software artifacts or programming constructs that it can analyze, so that it may be triggered when a new artifact is submitted to the system for evaluation.

The vulnerability detector plug-ins also implement the metric plug-in interface with each supported vulnerability type corresponding to one metric. This allows them to be transparently supported by existing tools built on top of Alitheia Core, so that aggregate and detailed statistics on the number of specific types of vulnerabilities can be calculated and reported for each software project or artifact.

An OSGi archetype for creating vulnerability detector plug-ins that are preconfigured to use the existing infrastructure is provided to simplify the integration of external tools for static analysis. By deriving from provided abstract classes that handle most of the tedious details, only a minimal number of methods need to be implemented in order to add support for a third party tool to the platform. Plug-ins for two third party tools have already been implemented using this archetype (see Section 4). However developers may choose to create their own plug-ins that implement the required interfaces from scratch and manage the communication with the other components of the platform themselves.

3.2.3 Support for client applications

Extending the TRACER platform is not achievable solely by creating plug-ins. A major role for effectively utilizing the functionality available is played by the client-side applications provided. In order to facilitate the development of client applications, a RESTful service is implemented, providing a REST API for manipulating the platform by external entities and applications. This API allows access to the vulnerabilities data model and contains methods for manipulating entities such as security profiles. It also supports performing tasks such as triggering the analysis of a project and the retrieval of analysis results.

To demonstrate the usability of the API, the front-end of the platform has been implemented as a modern web application developed with AngularJS,[5] which uses the API to communicate with the platform. At the same time, since REST APIs are universal and not dependent on implementation, this allows the easy integration of the TRACER platform with other applications and systems.

4. TOOLS INTEGRATION

To evaluate our platform, we have created plug-ins to integrate two different tools for vulnerability detection, namely: *FindBugs* [21], and *Frama-C* [22]. The former analyzes applications written in Java, while the latter examines applications written in C. This highlights the fact that our platform is independent of the programming language used to develop a project being analyzed.

4.1 FindBugs

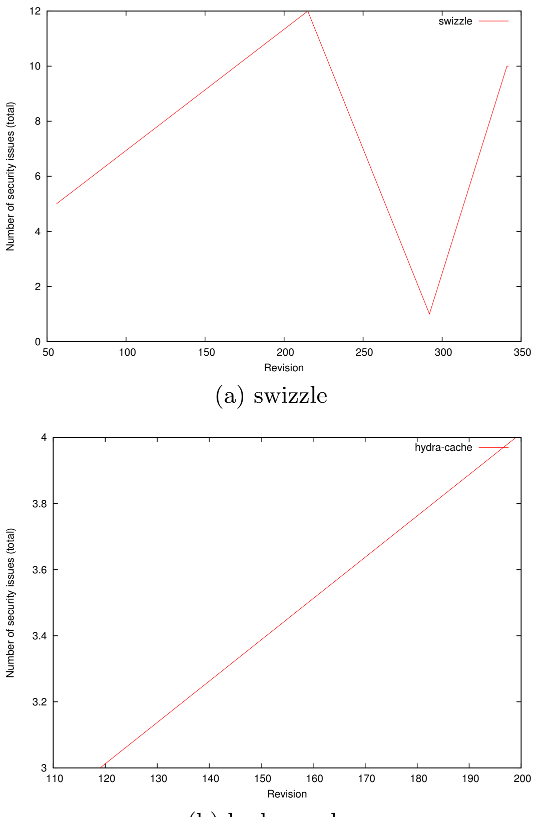
FindBugs[6] [21] is an open source static analysis tool that has been used many times for both commercial and research needs [23]. It searches for software bugs by examining the compiled Java virtual machine bytecode of an application.

To integrate Findbugs in the TRACER platform, we created a plug-in using the provided archetype. Since FindBugs runs on bytecode, our plug-in also builds the project before running FindBugs. 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 automate some of these tasks, modern build systems such as Maven[7] have been implemented. Such tools typically follow a standard directory structure for code and build artifacts. The Findbugs plug-in exploits the conventions used by Maven to automatically build each project and retrieve both the generated bytecode archives and the package structure. After the build completes, the Findbugs binary is invoked with appropriate arguments, so as to examine the bytecode that is created by the sources that belong to the specified project and not by its dependencies. The report that contains all the detected bug descriptions is generated in XML format. This report

[5] `http://angularjs.org/`

[6] `http://findbugs.sourceforge.net/`

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



(a) swizzle

(b) hydra-cache

Figure 3: Security bug frequency for two Java projects.

can be easily parsed in order to collect the bugs that we are interested in. The bugs are then associated with file revision information stored in the TRACER database through path name matching.

We have examined two open source projects that use the Maven build system, namely *swizzle*[8] and *hydra-cache*.[9] Figure 3 depicts the evolution of security bugs of both measurements for each project. Note that both of the projects deal with untrusted input, therefore they could become targets for exploits. The most interesting observation that we can make is that the security bugs are *increasing* as projects evolve, contrary to what one would hope.

4.2 Frama-C

Frama-C[10] [22] provides source code analysis in C programs. It implements a plug-in architecture over a kernel that controls the whole analysis. Custom plug-ins and user-defined properties written in a behavioural specification language extend the tool's functionality.

The STAC [24] analysis tool is a Frama-C plug-in that provides an approach for characterizing the execution paths which may lead to exploitable vulnerabilities. This is achieved by applying a sound taint analysis with a taint dependency sequence calculus, that makes the sequences of control and data dependencies which must be met in order for a variable to become tainted explicit. We have defined five configurations based on STAC in order to detect defects like format string vulnerabilitiesand SQL injection vulnerabilities [4].

The process of integrating Frama-C with the STAC plug-in within the TRACER platform was similar to that used for Findbugs. After integrating the tool, we applied it on two commercial projects: *Clearsilver-0.10.5*,[11] and MC*rypt-2.6.8*.[12] The results for both projects are described in Table 1.

Project	Vulnerability	Untainted
Clearsilver-0.10.5	Format string	✔
	Double free	✔
	User kernel trust error	✔
	SQL injection attack	✔
	Cross-site-scripting	✔
MC*rypt-2.6.8*	Format string	✔
	Double free	✔
	User kernel trust error	✔
	SQL injection attack	✔
	Cross-site-scripting	✔

Table 1: Occurrences of security bugs in the projects examined by Frama-C.

5. CONCLUSIONS AND FUTURE WORK

Software vulnerabilities are an ongoing security concern due to the continued use of unsafe programming languages, bad development practices and insufficient or ineffective testing. In this paper we present TRACER, a software platform to support the development of more secure applications by constantly monitoring software projects for vulnerabilities during development and maintenance. The platform allows the easy integration of third party tools that detect software vulnerabilities as plug-ins and handles their activation in an efficient, automated manner. By simplifying the process of running a variety of such tools and collecting their results in an automated manner during development TRACER can be also used to track security defects throughout the evolution history of software projects. Finally, it provides programmatic interfaces for performing queries on the analysis results, metrics, and metadata in a homogenous way, regardless of the tools that were used to generate them.

Even though we used two static analysis tool in our proof of concept, 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 code to detect software defects that could be integrated in TRACER [2]. 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 examine source code should allow us to run our framework on more projects and enrich our results. In this manner, 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.

[8] http://swizzle.codehaus.org/

[9] http://hydra-cache.codehaus.org/

[10] http://frama-c.com/

[11] http://www.clearsilver.net/downloads/

[12] http://sourceforge.net/projects/mcrypt

Acknowledgments

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).

This research has been co-financed by the European Union (European Social Fund—ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF)—Research Funding Program: Thalis—Athens University of Economics and Business—Software Engineering Research Platform.

6. REFERENCES

[1] McGraw, G.: Software Security: Building Security In. Addison-Wesley Professional (2006)

[2] Shahriar, H., Zulkernine, M.: Mitigating program security vulnerabilities: Approaches and challenges. ACM Comput. Surv. **44**(3) (June 2012) 11:1–11:46

[3] Telang, R., Wattal, S.: Impact of software vulnerability announcements on the market value of software vendors - an empirical investigation. In: Workshop on the Economics of Information Security. (2007) 677427

[4] Ray, D., Ligatti, J.: Defining code-injection attacks. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '12, New York, NY, USA, ACM (2012) 179–190

[5] Chess, B., West, J.: Secure programming with static analysis. Addison-Wesley Professional (2007)

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

[7] Ozment, A., Schechter, S.E.: Milk or wine: does software security improve with age? In: Proceedings of the 15th conference on USENIX Security Symposium - Volume 15. USENIX-SS'06, Berkeley, CA, USA, USENIX Association (2006)

[8] Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., Turski, W.M.: Metrics and laws of software evolution - the nineties view. In: Proceedings of the 4th International Symposium on Software Metrics. METRICS '97, Washington, DC, USA, IEEE Computer Society (1997) 20–

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

[10] Bozorgi, M., Saul, L.K., Savage, S., Voelker, G.M.: 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, New York, NY, USA, ACM (2010) 105–114

[11] Herraiz, I., Izquierdo-Cortazar, D., Rivas-Hernández, F.: Flossmetrics: Free/libre/open source software metrics. In: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering. CSMR '09, Washington, DC, USA, IEEE Computer Society (2009) 281–284

[12] Cubranic, D., Murphy, G.: Hipikat: recommending pertinent software development artifacts. In: Software Engineering, 2003. Proceedings. 25th International Conference on. (May 2003) 408–418

[13] Johnson, P., Kou, H., Paulding, M., Zhang, Q., Kagawa, A., Yamashita, T.: Improving software development management through software project telemetry. Software, IEEE **22**(4) (July 2005) 76–85

[14] Campell, A., Papapetrou, P. In: SonarQube in Action, Manning Publications (October 2014)

[15] Cifuentes, C., Scholz, B.: Parfait: Designing a scalable bug checker. In: Proceedings of the 2008 Workshop on Static Analysis. SAW '08, New York, NY, USA, ACM (2008) 4–11

[16] Cifuentes, C., Keynes, N., Li, L., Scholz, B.: Program analysis for bug detection using parfait: Invited talk. In: Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. PEPM '09, New York, NY, USA, ACM (2009) 7–8

[17] Chatzieleftheriou, G., Katsaros, P.: Test-driving static analysis tools in search of c code vulnerabilities. 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops **0** (2011) 96–103

[18] Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis transformation. In: Code Generation and Optimization, 2004. CGO 2004. International Symposium on. (March 2004) 75–86

[19] Cifuentes, C., Hoermann, C., Keynes, N., Li, L., Long, S., Mealy, E., Mounteney, M., Scholz, B.: Begbunch: Benchmarking for c bug detection tools. In: Proceedings of the 2Nd International Workshop on Defects in Large Software Systems. DEFECTS '09, New York, NY, USA, ACM (2009) 16–20

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

[21] Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. **39** (December 2004) 92–106

[22] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. In: Proceedings of the 10th International Conference on Software Engineering and Formal Methods, Berlin, Heidelberg, Springer-Verlag (2012) 233–247

[23] Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: 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, New York, NY, USA, ACM (2007) 1–8

[24] Gyrard, A., Bonnet, C., Boudaoud, K.: The STAC (Security Toolbox: Attacks & Countermeasures) Ontology. In: 22nd International World Wide Web Conference. , Rio de Janeiro, Brazil (May 2013) 165–166