TESIS DOCTORAL

METODOLOGIA DE EVALUACIÓN DE HERRAMIENTAS DE ANÁLISIS AUTOMÁTICO DE SEGURIDAD DE APLICACIONES WEB PARA SU ADAPTACIÓN EN EL CICLO DE VIDA DE DESARROLLO

ASSESSMENT METHODOLOGY OF WEB APPLICATIONS AUTOMATIC SECURITY ANALYSIS TOOLS FOR ADAPTATION IN THE DEVELOPMENT LIFE CYCLE

JUAN RAMÓN BERMEJO HIGUERA

Ingeniero en Informática por la Universidad Nacional de Educación a Distancia

Tesis presentada en el

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y CONTROL

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

como parte de los requerimientos para la obtención del

Grado de Doctor

2014

DEPARTAMENTO DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y CONTROL
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES
UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Título de la Tesis:

METODOLOGIA DE EVALUACIÓN DE HERRAMIENTAS DE ANÁLISIS AUTOMÁTICO DE SEGURIDAD DE APLICACIONES WEB PARA SU ADAPTACIÓN EN EL CICLO DE VIDA DE DESARROLLO

ASSESSMENT METHODOLOGY OF WEB APPLICATIONS AUTOMATIC SECURITY ANALYSIS TOOLS FOR ADAPTATION IN THE DEVELOPMENT LIFE CYCLE

Autor:

JUAN RAMÓN BERMEJO HIGUERA

Ingeniero en Informática por la Universidad Nacional de Educación a Distancia

Director de la Tesis:

Dr. GABRIEL DÍAZ ORUETA

Agradecimientos

A Ana, Paula y Marta por la cantidad de horas de atención robadas.

A mi hermano Javier por el apoyo y ánimo constante.

A Gabriel, por haberme dado esta oportunidad y atención.

Resumen

Como técnicas de análisis de seguridad de una aplicación, las pruebas de caja blanca y de caja negra realizadas manualmente (revisión de código, test de penetración), sufren de falta de cobertura de la superficie de ataque que tienen las aplicaciones y lo más probable es que con estas pruebas de penetración manuales se tenga una gran pérdida de detección de vulnerabilidades de seguridad. La dificultad de realizar estos test manualmente conduce al desarrollo de técnicas automáticas de análisis de la seguridad. Este trabajo de tesis trata de fijar el "*estado del arte"* en cuanto a las últimas tendencias de herramientas de análisis automáticas: análisis estático de caja blanca (SAST), dinámico de caja negra (DAST) y análisis dinámico de caja blanca (RAST / IAST) en tiempo real. También hay disponibles desarrollos de herramientas híbridas combinando varias de los tipos anteriores, con el objetivo de reducir los falsos positivos y negativos que tienen las herramientas de análisis estático y dinámico. Todas estos tipos de herramientas son evaluadas de acuerdo a una metodología desde una perspectiva global para establecer el grado de eficacia de las herramientas en cuanto detecciones correctas (verdaderos positivos), falsas alarmas (falsos positivos), grado de cobertura de vulnerabilidades, etc.

La metodología de evaluación de las herramientas consiste en ejecutar cada herramienta contra aplicaciones *benchmark* que contienen vulnerabilidades de seguridad conocidas. Las aplicaciones *benchmark* utilizadas han de tener la capacidad de comprobar falsos positivos, es decir, porciones de código seguras controladas donde las herramientas no deberían informar de la existencia de una vulnerabilidad concreta. Al resultado de las ejecuciones se les aplican posteriormente métricas seleccionadas y ampliamente aceptadas para establecer un ranking en cuanto a la efectividad de análisis de seguridad de cada herramienta.

El objetivo final del resultado de la evaluación de los diferentes tipos de herramientas mencionados, es la derivación de un modelo de ciclo de vida de desarrollo seguro de

software (SSDLC), aplicando en cada fase los tipos de herramientas más adecuados para conseguir un resultado de conjunto lo más optimizado posible. La diferente naturaleza de cada tipo de herramienta, e incluso entre distintas herramientas del mismo tipo, hace necesario estudiar la sinergia existente entre ellas cuando se combinan para reducir el porcentaje de falsos positivos y aumentar el porcentaje de verdaderos positivos.

El modelo de **ciclo de vida de desarrollo seguro de software** resultante siempre estará en función de los tipos de herramientas disponibles, del personal disponible para realizar las tareas de análisis, del tiempo y otros factores como los cambios de tendencias de la frecuencia y peligrosidad de las vulnerabilidades con el tiempo, de la evolución de las propias herramientas y de la aparición de otras nuevas. Estos factores de cambio implican analizar esta evolución para adaptar continuamente el modelo de SSDLC.

Summary

As security analysis techniques of an application, white-box and black testing performed manually (code review, penetration testing), suffer from lack of coverage to analyze the applications attack surface and most likely manual penetration testing has a great loss of security vulnerability detection. The difficulty in performing these tests manually leads to the development of automatic techniques of security analysis. This dissertation will set the "state of the art" in terms of the latest trends in automatic analysis tools: static analysis white box (SAST), Dynamic black box (DAST) and white-box dynamic analysis (RAST / IAST) in real time. Also, available hybrid tool developments combining several of the above types, with the goal of reducing false positives and negatives that static and dynamic analysis tools suffer. All these types of tools are evaluated according to a methodology, from a global perspective, to establish the performance of the tools in terms of correct detections (true positives), false alarms (false positives), degree of vulnerability coverage, etc.

The methodology of evaluation of the tools is to run each tool against benchmark applications containing known security vulnerabilities. Benchmark applications used must have the ability to check false positives with controlled safe portions of code where tools should not report the existence of a particular vulnerability. Subsequently widely accepted metrics are selected and applied to the results of the executions to establish a ranking in terms of the performance of security analysis of each tool.

The final objective of the result of the evaluation of different types of tools mentioned above, is the derivation of a secure software development lifecycle (SSDLC), using at each stage the types of tools best suited for a result as optimized as possible. The different nature of each type of tool and even between different tools of the same type obliges us to explore

the synergy between them when combined to reduce the percentage of false positive detections and increase the percentage of true positives.

The secure software development lifecycle model resulting will always depend on the types of tools available, the personnel available to perform the analysis tasks, time and other factors such as changes in trends in the frequency and danger of vulnerabilities with time and also evolutions of the tools themselves. These factors involve analyzing these evolutions and changes to adapt continuously the SSDLC model.

Preface

This thesis is the result of six years of research in the area of software security analysis accomplished with white and black box analysis security tools at the Electrical Engineering Department (*Departamento de Ingeniería Eléctrica, Electrónica y de Control*) of UNED (Spanish University for Distance Education). During this time, I have dedicated myself to be informed about the latest software security analysis tools tendencies. This knowledge is necessary to perform an efficient analysis in a reasonable time, so I have been developing the most adequate methodology to evaluate many of the available automatic tools to perform the best and most efficient security test of an application.

I started investigating about security vulnerabilities that developers introduce inadvertently in the code, their nature, attack vectors, the most frequent ones and the most dangerous vulnerabilities, with the aid of information distributed by important security standards. Also, I studied the ways to perform a software security analysis with the aid of specialized tools, static with source and executable code, black box analysis in runtime and white box analysis in runtime. I investigated the available tools, the skills of each tool and also their problems, as false negatives rates, false positives rates or vulnerabilities coverage degree.

Next, I continued investigating on developing a methodology for evaluating the security tools to establish a strict rank of tool performance in software security analysis to help the practitioners to make the best election to accomplish a security analysis.

I participated in security courses as “Security of Technologies of information and Communications” “Risk analysis tools” and “Cryptologist specialist” performed in the Spanish Cryptology National Center (CCN), belonging to the Spanish Intelligent National Center (CNI), which gave me a good perspective of the state of software security.

I have published a research paper about source code analysis titled “Static analysis of source code security: Assessment of tools against SAMATE tests” in Information and Software Technology journal, from Elsevier.

I have participated in several conferences about “Static analysis for source code security and “Automatic Analysis tools for security of applications” at the University of Alcala de Henares and e-Madrid e-learning project.

I have promoted and advised the implementation of a software security analysis project in the Marañosa Institute of technology of Spanish Defense Ministry, with the purpose of analyzing the security of source code software of all projects belonging to the Defense Ministry.

Index

Glossary

API	Application Program Interface
CBMC	C Bounded Model Checking tool
CLASP	Comprehensive, Lightweight Application Security Process
CMDI	Command Injection
CMM	Capability Maturity Models
CMMI	Capability Maturity Model Integration
CSRF	Cross Site Request Forgery
CVE	Common Vulnerability Enumeration
CVSS	Common Vulnerability Score System
CWE	Common Weak Enumeration (MITRE Corporation)
DAST	Static Application Security Testing
DISA	Defense Information Systems Agency
DOS	Denegation of service
FAA-iCMM	Federal Aviation Administration integrated Capability Maturity Model
FN	False Negative
FP	False Positive
GPL	General Public License
HTML	Hyper Text Marked Language
HTML5	Hyper Text Marked Language version 5
IAST	Interactive Application Security Testing
ICS	Industrial Control Systems
ICT	Information and Communication Technology
IETF	Internet Engineering Task Force
J2EE	Java 2 Enterprise Edition
JSON	JavaScript Object Notation

LFI	Local File Inclusion
MVC	Model View Controller
NFA	Non deterministic Automaton
NFC	Near Field Communications
NIST	National Institute of Standards and Technologies U.S.A.
NSA	National Security Agency U.S.A.
OASIS	Open Control Standards for the Information Society
OISSG	Open Information Systems Security Groups
OSVDB	Open Source Vulnerability Database
OWASP	Open Web Application Security Project
PCI DSS	Payment Card Industry Data Security Standard
PQL	Program Query Language
RAST	Real Time Application Security Testing
RFI	Remote File Inclusion
RIA	Rich Internet Applications
RIA	Rich Internet Applications
SAMATE	Software Assurance Metrics and Tool Evaluation
SANS	Institute for security training
SAST	Static Application Security Testing
SAT	Satisfiability
SCA	Source Code Analyzer
SCADA	System Control Architecture and Data Acquisition
SDL	Security Development Lifecycle
SEI	Software Engineering Institute
SOA	Service Oriented Architecture
SOA	Service Oriented Architecture

SQLI	SQL Injection
SSDLC	Secure Software Development Life Cycle
SSE-CMM	Systems Security Engineering Capability Maturity Model
TC	Test Case
T-CMM/TSM	Trusted CMM/Trusted Software Methodology
TN	True Negative
TP	True Positive
WAF	Web Application Firewall
WASC	Web Application Security Consortium
WAVSEP	Web Application Vulnerability Evaluation Project
WHID	Web Hacking Incident Database
XHTML	eXtended Hyper Text Marked Language
XML	eXtended Marked Language
XMLI	XML Injection
XSF	cross-frame scripting
XSS	Cross-site scripting
XXE	XML eXternal Entity
ZDI	Zero Day Initiative

Figure Index

Table index

1. INTRODUCTION

1.1. MOTIVATION

Communication and Information systems are facing today some of the biggest security challenges because of different kinds of risks: trojans for identity theft, spyware, traditional sniffing of secure information, electronic warfare and many more forms of cyberattacks. Many industries depend on software coming from vendors, open source or third parties. The number of distributed systems with more complex interactions is growing as well, comprising operating system and application patches coming from Internet or from collaboration through different distributed sites. Applications are used to shop, communicate, banks transactions, logistic and personal management in companies and organizations. The security needs for command and control air systems, avionics systems [Black, 2007], social networks and Internet Portals [Yahoo, 2013], or the most recent virus attack [Flame, 2012] demonstrate that information security is a multifaceted problem where organizations, enterprises and users need security assurance on the software they use. According to the Veracode Volume 5 report [Veracode, 2012], that examines data collected over an 18 month period from January 2011 through June 2012 ,from 22,430 applications builds uploaded (web, non-web and mobile), 70% of them failed to comply with enterprise security policies and 87% with the OWASP Top 10 [Owasp, 2013]. The web applications were over 75% of the total in Veracode report. The security vulnerabilities, risk and attacks applications can suffer should force organizations to make a security analysis to avoid as many threats as possible.

This dissertation thesis is about a kind of vulnerabilities and security threats different of security problems emanating from poor management configuration, server application type, database management system, development framework adopted or security services

according to the specification or developing technology used. Among these security services are the access authentication and authorization for access to application resources, such as database, access to operating system resources where you installed the application server machine even client machines. However, some types of vulnerabilities found in the code may lead to vulnerabilities of privileges escalation allowing access to forbidden resources. This may suggest that the problem is poor security administrator permissions regarding security based on users and roles for instance, when the real problem is a vulnerability of privileges escalation. To solve this kind of problems, vulnerabilities in the code must be patched.

The most desirable mean to avoid vulnerabilities in applications code is prevention, developers should have been trained in security programming to avoid making "mistakes" involving programming vulnerabilities. Even when a very good training of programmers exists, there will always be vulnerabilities in the code and there will be no choice once developed the first version of the application. Software engineers must consider a variety of strategies to build secure software before release. Achieving this goal is only possible by using various techniques and automatic tools to ensure security in all phases of SSDLC.

As a testing technique, white box and black box manuals skills (code auditing, penetrating tests), still suffer from lack of coverage and therefore likely they miss a large fraction of vulnerabilities. Manual code auditing to analyze the security of a web application with thousands of lines of code can become an arduous and painstaking work, quite time consuming. Also, a manual penetration test has the problem of covering all inputs to the application (attack surface) of the application and testing all user roles. This is really difficult and there are aspects not able to test with this method.

The difficulty of performing such tests manually leads us to examine last tendencies in automatic techniques. Indeed the tools used for software development and maintenance can

supply developers with information for assurance cases. This information must be gathered to get software secure enough for its intended use. Different types of automatic tools can be used for examining source code, the entire application deployed included in a secure software development lifecycle. Automated security analysis tools of source and executable code, automatic vulnerabilities scanners and interactive real time analysis tools are increasingly used today and taken into account in software development strategies. Those tools are designed to detect vulnerabilities: flaws, faults, bugs and other errors in software code that, if left unaddressed, could lead to exploitable security vulnerabilities.

1.2. PURPOSE

The goal of this thesis is to help practitioners to select appropriate tools for a security review process of software in all phases of Secure Software Development Life Cycle (SSDLC). This thesis establish a well-defined and **repeatable methodology** to evaluate the tools selected for each type of automatic security analysis, and how those ones can be integrated in the SSDLC, correlating their results and obtaining the most secure possible software as a whole result. The tools can have several problems while performing an analysis:

- False positives: A tool can report a security vulnerability in a program that is not really a vulnerability.
- False negatives: a security vulnerability in the code which is not detected by the tool.
- Coverage degree of vulnerabilities detection. A security tool must have a complete coverage of most dangerous and frequent vulnerabilities according to OWASP TOP TEN 2013 [Owasp, 2013] or SANS TOP 25 [Sans, 2013].

To accomplish the goal, this thesis:

- Examines the state of the art of all automatic security analysis tools categories available to perform a security process in a SSDLC:
  - Static Application Security Testing (SAST). White box tools that perform a static analysis of source or executable code of the application.
  - Dynamic Application Security Testing (DAST). Black box tools that perform a dynamic analysis of the application.
  - Real time Application Security Testing (RAST) or Interactive Application Security Testing (IAST). White box tools that perform a runtime analysis of the application.
  - Hybrid tools SAST-DAST, SAST-DAST-RAST, SAST-RAST, and DAST-RAST.

- Each category is evaluated following a well-defined and repeatable methodology. For each security analyzer type (SAST, DAST, RAST), the performance degree about its vulnerabilities detection capacity is obtained. The methodology uses a selected benchmark with a well-known set of security vulnerabilities. A tool has the best performance against a benchmark if it has the best balance between detecting the highest number of *true positives* and having few *false positives*. **The result of the assessment of each security tool category is a strict rank of the security performance of all tools involved in its respective comparative.**

- Finally, the defined methodology is used firstly to compare distinct types of security tools, analyzing and obtaining conclusions about their different capabilities. Also the tools are evaluated from a whole perspective, correlating their results to obtain better results with more detection and less false positives, with the final objective of getting the best integration in the SSDLC.

The main conclusions obtained by this study are related with:

- Which kind of vulnerabilities each security tool type detects.
- Correlation detection results between distinct types of tools.
- The most appropriate phase of the SSDLC for using each security tool type.
- How the different types of security tools can be best combined in different phases of SSDLC to get the best whole result.
- Comparing the results obtained in by each tool in the same phase of SSDLC and the results obtained combining several tools.
- Recommendations to the personnel that must make security analysis of applications to obtain the best performance of automatic security analysis.

1.3. DOCUMENT ORGANIZATION

Table 1 shows the organization of this dissertation.

Table 1
Document organization

Stages	Thesis section
Introduction	**1.** INTRODUCTION
Background and State of the Art	**2.** LAST TENDENCIES IN APPLICATIONS DEVELOPMENT **3.** APPLICATIONS SECURITY PROBLEMS **4.** STATE OF THE ART IN APPLICATIONS SECURITY ANALYSIS
Automatic security tools evaluation	**5.** ASSESSMENT METHODOLOGY OF SECURITY ANALYSIS TOOLS
Discussion and research questions	**6.** DISCUSSION.
Related work	**7.** RELATED WORK AND DISCUSSION.
Conclusions and Future work	**8.** CONCLUSIONS AND FUTURE WORK

Chapter 1 describes the motivation, purpose and objectives of the thesis, showing the steps, methods and resources used to achieve them. It also contains the document structure.

Chapter 2 examines the last tendencies in applications development, attending to new architectures, frameworks and the most used development technologies.

Chapter 3 is a study of the most common security problems in applications, security vulnerabilities, risk and threats that applications are exposed to, and the attacks they can suffer as a consequence of these defects and lacks in security requirements derivation, design or implementation. It also shows the attacks vectors and vulnerabilities evolution in the last years, to aid understanding how to better protect an application.

Chapter 4 shows an updated state of the art of security artifacts, resources, tools, metrics, methodologies and benchmarks that can be used to improve the security of an application against the most dangerous attacks. In particular, we analyze:

- Software security organizations and Standards.
- Secure Software Development Life Cycle.
- Static Application Security Testing (SAST).
- Dynamic Application Security Testing (DAST).
- Real time Application Security Testing (RAST) or Interactive Application Security Testing (IAST).
- Hybrid tools SAST-DAST, SAST-DAST-RAST, SAST-RAST, and DAST-RAST.
- Assessment methodologies of security tools.
- Benchmarks used for assessments of security tools.

Chapter 5 describes in detail our well-defined methodology to assess each automatic security tools category, to obtain a strict rank of their security performance. Afterwards, the methodology is applied to perform the evaluation of:

- Application static analysis tools. This assessment is based on Juan R. Bermejo and Gabriel Díaz publication in Information and Software Technology journal: *Static*

analysis of source code security: Assessment of tools against SAMATE tests [Diaz, 2013].

- Web application static analysis tools. This assessment is based on master final work of the doctoral formation period of Juan R. Bermejo [Bermejo, 2011].

- Web application dynamic analysis tools and web application hybrid analysis tools. This assessment is performed in the thesis investigation period.

Chapter 6 analyzes and discusses the assessments results answering to four complementary research questions about performance, adequation of benchmarks, usability of the tools and how leveraging the assessments results to integrate all categories of security automatic tools in SSDLC. This will allow obtaining the best performance in the security review process of an application, as a whole result of exploiting the different individual skills in detection capabilities of tools.

Chapter 7 reviews different related works on automatic security analysis tools comparisons. It also compares the results of these studies with our own results, discussing differences and similarities between them.

Chapter 8 summarizes the main conclusions of this thesis. These conclusions are based on the tools comparisons results about the performance in vulnerability detection possibilities and false positive rate, vulnerabilities coverage and languages coverage degree or report quality and completeness. It also gives some guidelines on related future researchs.

2. LAST TENDENCIES IN APPLICATIONS DEVELOPMENT

Before starting the study of the tools available in the market and free software for automatic detection of security vulnerabilities in applications, it is necessary to give an overview of applications categories and the technologies, specifications and architectures commonly used to build an application. The choice of the application type, architecture, development specification or language has security implications that must be known. The developers must know the security characteristics of the languages that each technology use, such as java, C#, C++, HTML, scripting languages, etc. Differente languages implement different security checks in compilation time. The prevention degree of security vulnerabilities of each language depends on these previous security checks that define the security degree of a specific language. If a language incorporates implicitly compilation security checks, the developers do not have to include explicitly additional code to avoid vulnerabilities. For example, java implements security aspects such as implicit checking array limits preventing buffer overflow vulnerabilities [Long, 2005].

2.1. APPLICATION CATEGORIES.

This section is an overview of the benefits and considerations for the common application archetypes used to develop software applications. New architectures tendencies have appeared in web development as for example Rich Internet Applications (RIA) and Web services (WS) that need a review of the new security problems they introduce. Also Control Systems as Supervisory Control and Data Acquisition (SCADA) are being objective of attacks according to HP 2012 cyber security risk report [HP report, 2012]. In particular, the data for 2012 showed how the number of vulnerabilities disclosed in Supervisory Control

And Data Acquisition (SCADA) systems increased from 22 in 2008 to 191 in 2012 (a 768 percent increase).

The most general classification of software applications includes the following three categories:

- **Non-Web Applications**
    - Client-Server Applications
    - Service Oriented Architecture (SOA) applications (Dcom, Java-RMI, Corba, etc.)
    - Embedded Systems for avionics, Supervision Control Systems as SCADA or Air Defense commander and control systems, etc.

- **Web applications**
    - Traditional N-tier Web Applications with Model View Controller (MVC) pattern design
    - Rich Internet Applications (RIA)
        - Flash [Flash, 2013]
        - Ajax [Connolly 2008]
        - JavaFX [JavaFX, 2013]
        - Microsoft SilverLight [SilverLight, 2013]
        - OPenLaszlo [OPenLaszlo, 2013]
        - HTML5 [HTML5, 2013]
    - Web services

- **Mobile applications**
    - Native Applications
    - Web applications

Another classification for applications is the one given by Microsoft Corporation [Microsoft, 2013]:

- **Mobile applications**. Applications of this type can be developed as thin client or rich client applications. Rich client mobile applications can support disconnected or occasionally connected scenarios. Web or thin client applications support connected scenarios only. Device resources may prove to be a constraint when designing mobile applications. Figure 1 shows the mobile applications architecture.

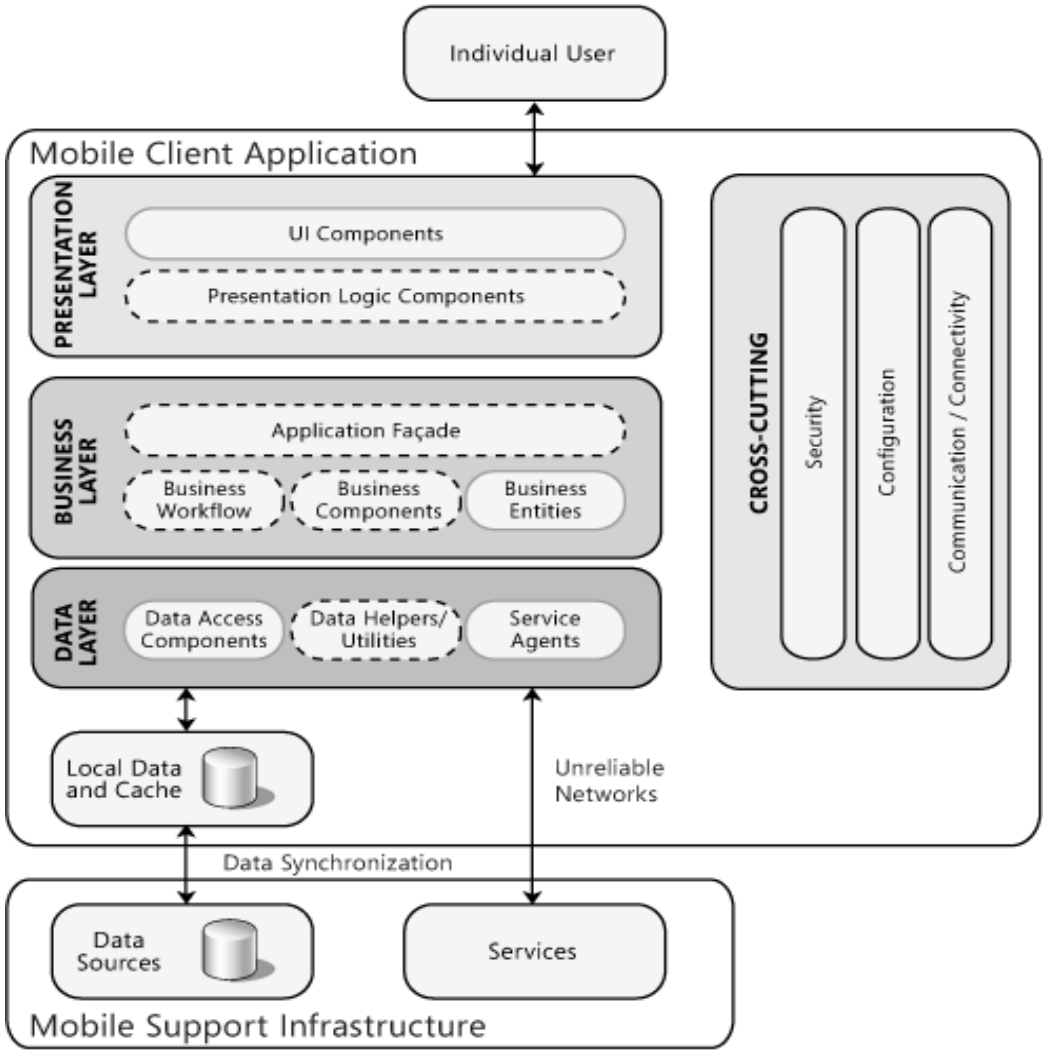

Figure 1. Mobile applications architecture [Microsoft, 2013]

- **Rich client applications**. Applications of this type are usually developed as stand-alone applications with a graphical user interface that displays data using a range of controls. Rich client applications can be designed for disconnected and occasionally connected scenarios if they need to access remote data or functionality. The most common security problems are design vulnerabilities and security vulnerabilities in the code. The choice of language to develop the application has to be as a function of its safety features. There are more security languages as *C#, Java, Python, Ruby* or *CCured* and *Cyclone* that check types and memory in compilation time. Figure 2 shows rich client applications architecture.

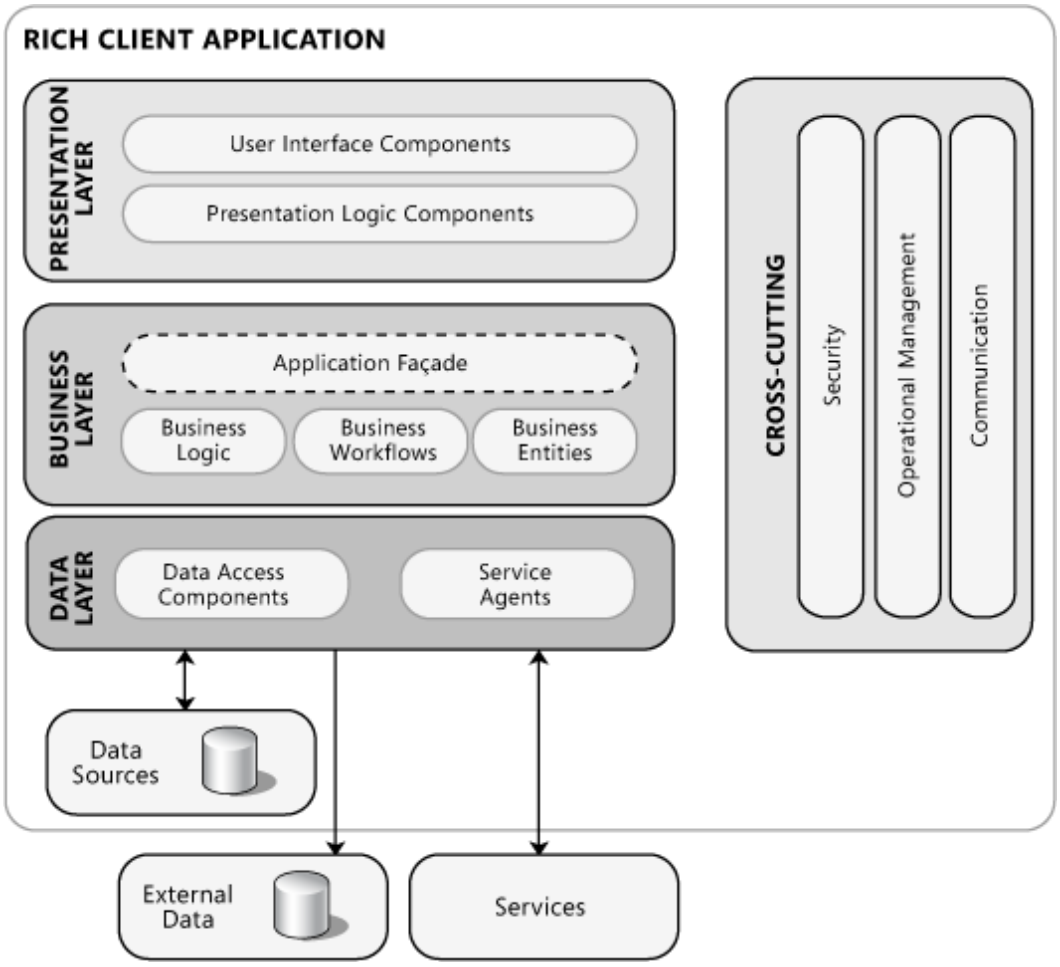

Figure 2. Rich client applications architecture [Microsoft, 2013]

- **Rich Internet applications**. Applications of this type can be developed to support multiple platforms and multiple browsers, displaying rich media or graphical content. Rich Internet applications run in a browser sandbox that restricts access to some features of the client. The principal new security problem that this architecture type introduces is because of client engine (usually in *javascript* code). The javascript engine can be visualized and reveals the logic application information and can be the source of more common vulnerabilities as XSS. Figure 3 shows the rich internet applications architecture.

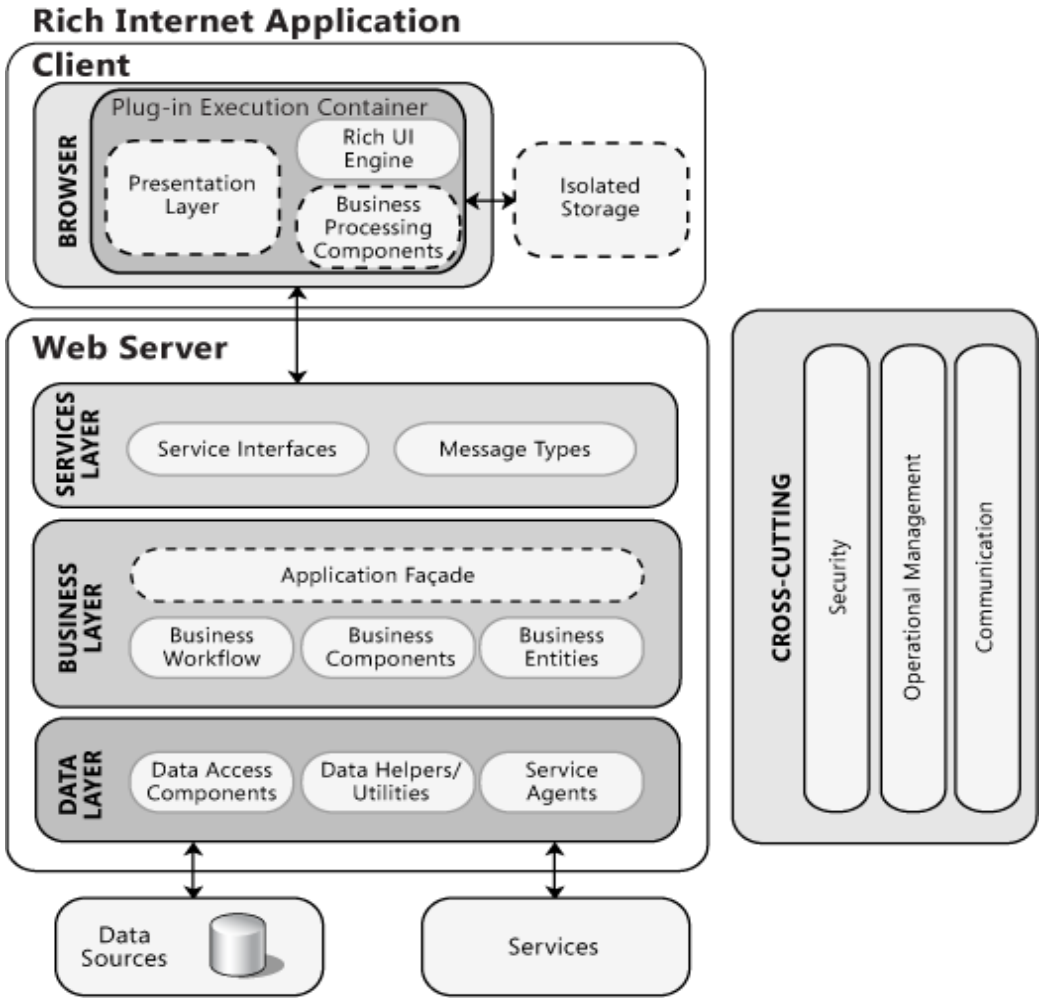

Figure 3. Rich Internet applications architecture [Microsoft, 2013]

- **Service applications**. Also called Service Oriented Architecture (SOA), services expose shared business functionality and allow clients to access them from a local or a remote system. Service operations are called using messages, based on XML schemas, passed over a transport channel. The goal of this type of application is to achieve loose coupling between the client and the server. In SOA, there are usually three entities: consumer of service, provider of service and Register for services offered by the provider. In that distributed environment to get all security elements as authentication, authorization, integrity, etc. is a great challenge. For getting the most secure web services applications, all entities should follow adequate security standards as a requisite to be compatible and interoperable. Figure 4 shows the service applications architecture.

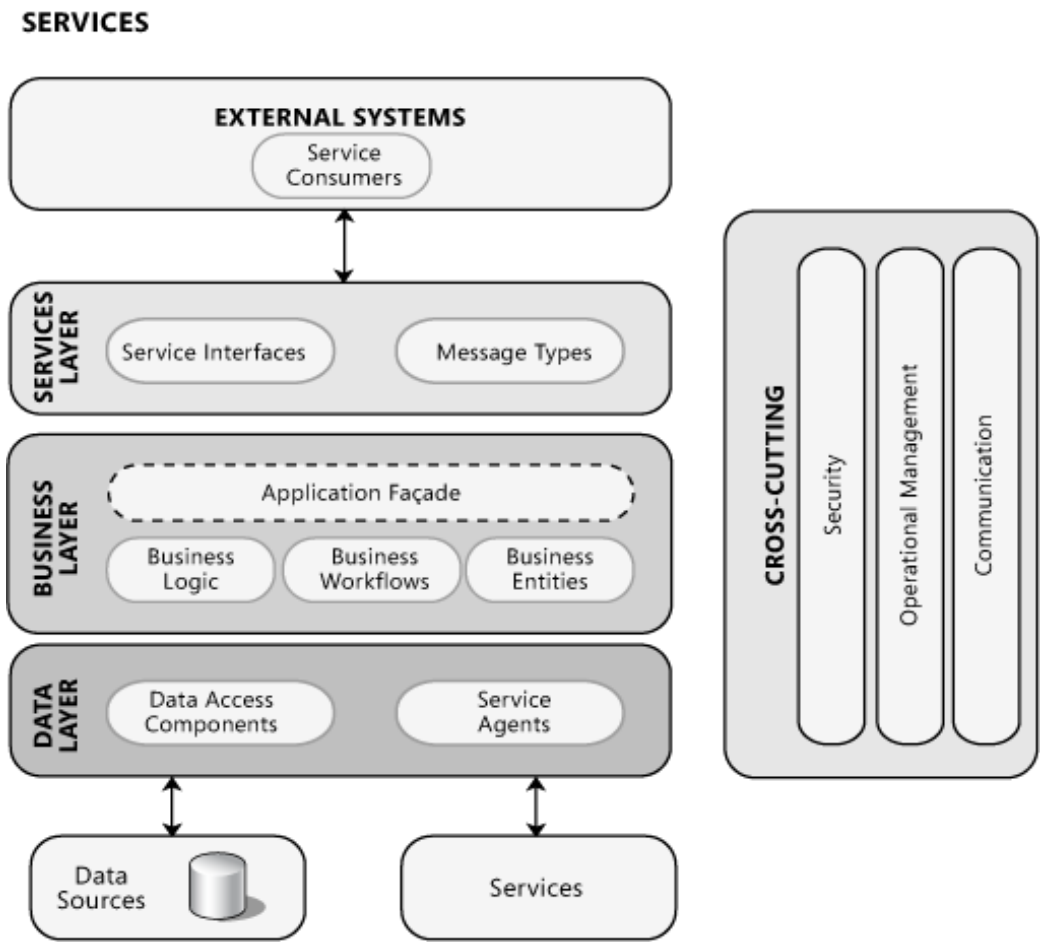

Figure 4. Service applications architecture [Microsoft, 2013]

- **Web applications**. Applications of this type typically support connected scenarios and can support different browsers running on a range of operating systems and platforms. The most used design pattern in development of web applications is Model View controller (MVC) with three tiers of software: the *model* consists of application data, business rules, logic, and functions. A *view* can be any output representation of data, such as a chart or a diagram. Multiple views of the same data are possible, such as a bar chart for management and a tabular view for accountants. The *controller* mediates input, converting it to commands for the model or view (figure 5).

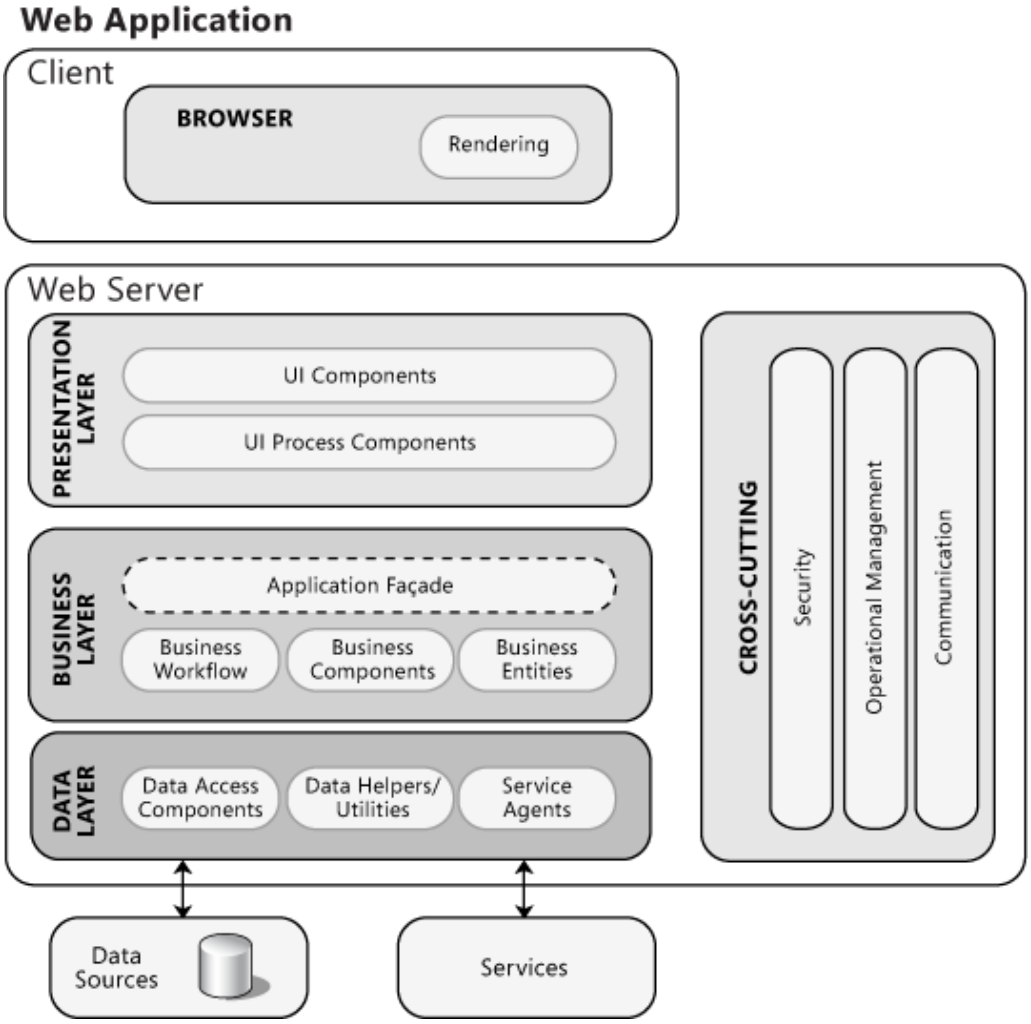

Figure 5. Web applications architecture [Microsoft, 2013]

Table 2 indicates the benefits and considerations for the common application archetypes. Each application type can be implemented using one or more technologies. Scenarios and technology constraints, as well as the capabilities and experience of the development team, will drive the choice of technology.

Table 2
Benefits and considerations for the common application archetypes [Microsoft, 2013]

Application type	Benefits	Considerations
Mobile applications	Support for handheld devices. Availability and ease of use for out of office users. Support for offline and occasionally-connected scenarios.	Input and navigation limitations. Limited screen display area.
Rich client applications	Ability to leverage client resources. Better responsiveness, rich UI functionality, and improved user experience. Highly dynamic and responsive interaction. Support for offline and occasionally connected scenarios.	Deployment complexity; however, a range of installation options such as ClickOnce, Windows Installer, and XCOPY are available. Challenging to version over time. Platform specific.
Rich Internet applications (RIA)	The same rich user interface capability as rich clients. Support for rich and streaming media and	Larger application footprint on the client compared to a Web application. Restrictions on leveraging client resources compared to a rich client

	graphical display. Simple deployment with the same distribution capabilities (reach) as Web clients. Simple upgrade and version updating. Cross-platform and cross-browser support.	application. Requires deployment of a suitable runtime framework on the client.
Service applications	Loosely coupled interactions between client and server. Can be consumed by different and unrelated applications. Support for interoperability.	No UI support. Dependent on network connectivity.
Web applications	Broad reach and a standards-based UI across multiple platforms. Ease of deployment and change management.	Dependent on continual network connectivity. Difficult to provide a rich user interface.

The percentage of each category of applications are shown in figure 6, according to Veracode Security State of Software report Volume 5 [Veracode, 2012], that examines data collected over an 18 month period, from January 2011 through June 2012, from 22,430 application builds uploaded and assessed by its platform. Web applications have the highest rate with a 73 percent of total applications examined.

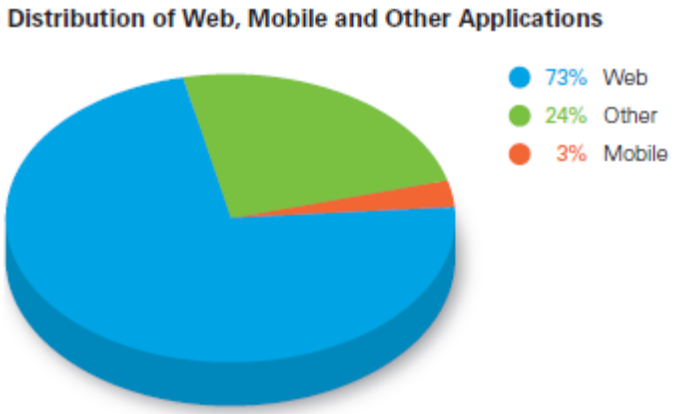

Figure 6. Distribution of applications by categories [Veracode, 2012].

Today the total of web and mobile applications are constantly increasing. Web applications are the most used over Internet and even in Intranets within organizations and are used to communicate, for bank transactions, e-commerce, e-learning education, logistic, human resources, shopping, etc. The great number of mobile devices available as mobile phones, tablets, smartphones, etc. gives the possibility of using mobile applications to communicate, m-commerce, etc. anywhere, anytime and this fact is motivating that more people every day use mobile device applications.

From the perspective of software application suppliers, figure 7 shows that approximately 22% of the applications analyzed were identified as third-party (commercial, open source and outsourced). The percentage of outsourced applications remains low at 1%. Often the outsourced nature of applications labeled "internally developed" is only revealed during remediation, when an outsourced part is assigned the task of fixing vulnerabilities. Probably the true percentage of "outsourced" code is higher than represented in figure 7. This fact obliged us to think in how to conduct a security analysis for third party software when the source code is not available. Security analysis tools for executable code can be very useful to perform code review in these cases.

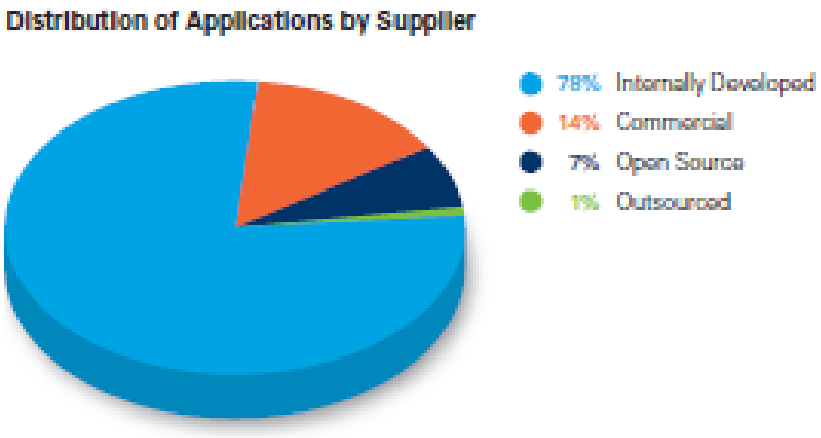

Figure 7. Distribution of applications by Supplier [Veracode 2012].

For commercial, open sourced and outsourced applications, it would be important knowing if the organization that acquires the application can rely on its security. The organization should be able to test all external application or having an official certification of approved security test, as the types addressed by this job accomplish over the SSDLC.

2.2. APPLICATIONS ARCHITECTURE STYLES.

The application categories described in previous section can be designed according to an architectural style. Architectural styles, sometimes called architectural patterns, are a set of principles, a coarse grained pattern that provides an abstract framework for a family of systems. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems. Architecture styles and patterns can be particularized as sets of principles that shape an application. Garlan and Shaw define an architectural style as [Garlan, 1994]:

"*Family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be*

used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition."

An understanding of architectural styles provides several benefits. The most important benefit is that they provide a common language. They also provide opportunities for conversations that are technology agnostic. This facilitates a higher level of conversation that is inclusive of patterns and principles, without getting into specifics. For example, by using architecture styles, somebody can talk about client/server versus *n*-tier. Architectural styles can be organized by their key focus area. Table 3 [Microsoft, 2013] lists the major areas of focus and the corresponding architectural styles.

Table 3
Common architectural styles [Microsoft, 2013]

Architecture style	Description
Client/Server	Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.
Domain Driven Design	An object-oriented architectural style focused on modeling a business domain and defining business objects based on entities within the business domain.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message Bus	An architecture style that prescribes use of a software system that can receive and send messages using one or more communication channels, so that applications can interact without needing to know specific details about each other.
N-Tier / 3-Tier	Segregates functionality into separate segments in much the same

	way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	A design paradigm based on division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Service-Oriented Architecture (SOA)	Refers to applications that expose and consume functionality as a service using contracts and messages.

Table 4 gives a somewhat abstract outline of the historical development of the various architectures according to A.T.M. Aerts, J.B.M. Goossenaerts, D.K. Hammer and J.C. Wortmann [Aerts, 2003]. They characterize each development phase by the dominant architectural model and a crude indication of the decade in which it became important, if not generally accepted. In this context, they were concerned with three domains in which architecture matters:

1. The business architecture defines the business system in its environment of suppliers and customers. The system consists of humans and resources (including ICT), business processes, and rules. It belongs to the disciplines of industrial engineering and management science.

2. The application architecture details the software application components and their interaction. Its details can be described using object or component models, or application frameworks. It belongs to the discipline of computer science.

3. ICT platform architecture is the architecture of the generic resource layer, which describes the computers, networks, peripherals, operating systems, data base management systems, UI frameworks, system services, middleware, etc. that will be used as a platform for the construction of the system for the enterprise. Its

description includes various platform paradigms such as mainframe-terminal, n-tier client–server, and mobile or wireless architectures. It belongs to the discipline of computer systems engineering.

Table 4
The historical development of the business, application, and ICT platform domains [Aerts, 2003]

	Business architecture	Application architecture	ICT architecture
1950s	Functional hierarchy	No	Limited
1960s	Functional hierarchy	Function oriented	Mainframe
1970s	Logistics imposed on functional hierarchy	Function oriented with DBMS	Information Islands
1980s	Business process	Two-tier C/S GUI	Networks of mainframe and minis
1990s	Supply chain	Enterprise applications	Multi-site, n-tier
Today	Web-enabled	(Generic) components OOUI	Ubiquitous computing

As commented in previous section according to application categories statistics, Web applications and ubiquitous computing with mobile applications are currently the last tendencies. The security efforts of organizations and companies must put emphasis in these new architectures and ubiquitous technologies without forgotting the other more traditional architectures and patterns.

2.3. DEVELOPMENT TECHNOLOGIES AND LANGUAGES

Before beginning the study of the tools available in the market and free software for automatic detection of security vulnerabilities in software applications, this thesis gives a small overview of the technologies and languages most commonly used for building applications. The aim of this work is to analyze the performance, when analyzing the code of these applications, of automatic detection tools for detecting vulnerabilities that could lead to the materialization of different threats (attacks).

In this sense this study emphasizes on a kind of vulnerabilities and security threats that are different from security problems emanating from poor management configuration in application servers, database management systems or client machines.

This section addresses mainly the language characteristics about security to get the most secure possible code. The security characteristics implemented implicitly by each language can help to avoid a concrete set of vulnerabilities. For example *java* language implements security aspects such as implicit checking *array* limits for preventing *buffer overflow* vulnerabilities [Long, 2005].

Table 5 shows a brief summary of the languages most commonly used in software application development according to Tiobe [Tiobe, 2013]. These include different languages and platforms such as C/C+ +, J2EE, ColdFusion, PHP and. NET, which are among the most commonly used today. These languages are used for all types of applications as web, service, mobile, non-web applications, etc. Periodically Tiobe publishes language statistics, updated including the degree of positive or negative tendency variation and the previous year position. C language is the most used with 18,729% followed by java with 16,914%.

Table 5
Index of languages more used up to May 2013 [Tiobe, 2013]

Position May 2013	Position May 2012	Delta in Position	Programming Language	Ratings May 2013	Delta May 2012	Status
1	1	=	C	18.729%	+1.38%	A
2	2	=	Java	16.914%	+0.31%	A
3	4	↑	Objective-C	10.428%	+2.12%	A
4	3	↓	C++	9.198%	-0.63%	A
5	5	=	C#	6.119%	-0.70%	A
6	6	=	PHP	5.784%	+0.07%	A
7	7	=	(Visual) Basic	4.656%	-0.80%	A
8	8	=	Python	4.322%	+0.50%	A
9	9	=	Perl	2.276%	-0.53%	A
10	11	↑	Ruby	1.670%	+0.22%	A
11	10	↓	JavaScript	1.536%	-0.60%	A
12	12	=	Visual Basic .NET	1.131%	-0.14%	A
13	15	↑↑	Lisp	0.894%	-0.05%	A
14	18	↑↑↑↑	Transact-SQL	0.819%	+0.16%	A
15	17	↑↑	Pascal	0.805%	0.00%	A
16	24	↑↑↑↑↑↑↑↑	Bash	0.792%	+0.33%	A
17	14	↓↓↓	Delphi/Object Pascal	0.731%	-0.27%	A
18	13	↓↓↓↓↓	PL/SQL	0.708%	-0.41%	A
19	22	↑↑↑	Assembly	0.638%	+0.12%	B
20	20	=	Lua	0.632%	+0.07%	B

2.3.1. LANGUAGES FOR NON-WEB APPLICATIONS

Table 6 shows the most commonly used languages for development of non-web applications:

Table 6
Languages more common used up for development of non-web applications [Tiobe, 2013]

Position May 2013	Position May 2012	Delta in Position	Programming Language	Ratings May 2013	Delta May 2012	Status
1	1	=	C	18.729%	+1.38%	A
2	2	=	Java	16.914%	+0.31%	A
3	4	↑	Objective-C	10.428%	+2.12%	A
4	3	↓	C++	9.198%	-0.63%	A
6	6	=	PHP	5.784%	+0.07%	A
7	7	=	(Visual) Basic	4.656%	-0.80%	A
13	15	↑↑	Lisp	0.894%	-0.05%	A
15	17	↑↑	Pascal	0.805%	0.00%	A
17	14	↓↓↓	Delphi/Object Pascal	0.731%	-0.27%	A

Table 6 is based on table 5, we have removed the languages non used for non-web applications development. It is important to see that Visual Basic, C++ and Java languages are used for web and non-web applications development and so table 6 reflects both uses in its statistics.

Another interesting statistic of languages use for non-web applications is *state of software report* of Veracode [Veracode, 2012], Figure 8 shows the statistics of most used languages in all applications analyzed in the year 2012. Java, .NET and C/C++ are the most used languages.

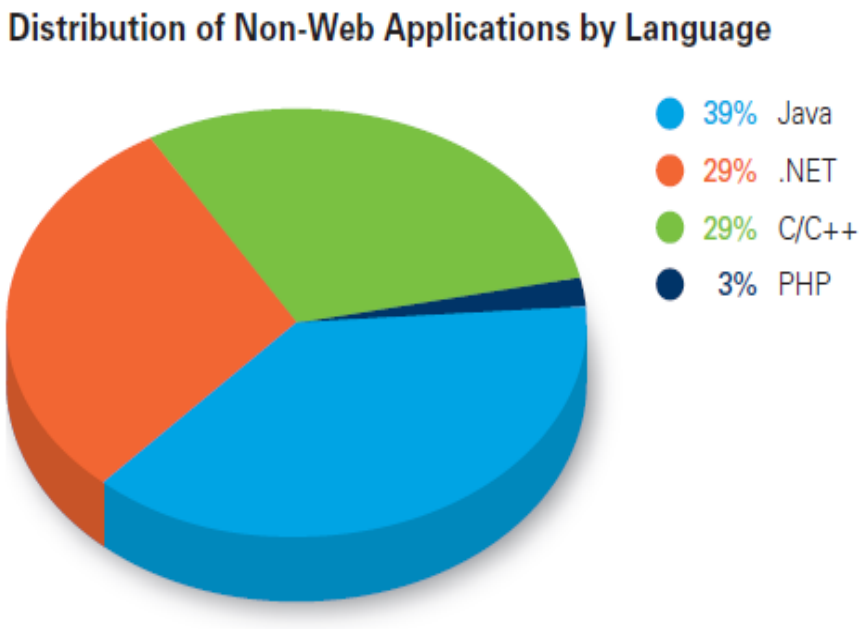

Figure 8. Languages more used for non-web applications [Veracode, 2012]

Next section analyzes the security characteristics of the most used languages to understand the precautions that a programmer should have when building an application.

2.3.2. LANGUAGES FOR WEB APPLICATIONS

The *state of software report volume 5* of Veracode [Veracode, 2012] of figure 6 showed that 73% of applications that Veracode analyzed in 2012 were web applications. The most used languages to develop web applications, according to that report, is shown in figure 9, the two dominant development specifications were J2EE with 56% and .NET with 28%. According to IBM x-force 2102 mid-year trend and risk report [IBM, 2012], the percentage of web applications vulnerabilities disclosures in 2012 were 47% of total, which gives an idea of the importance of putting a great effort to build web applications as secure as possible.

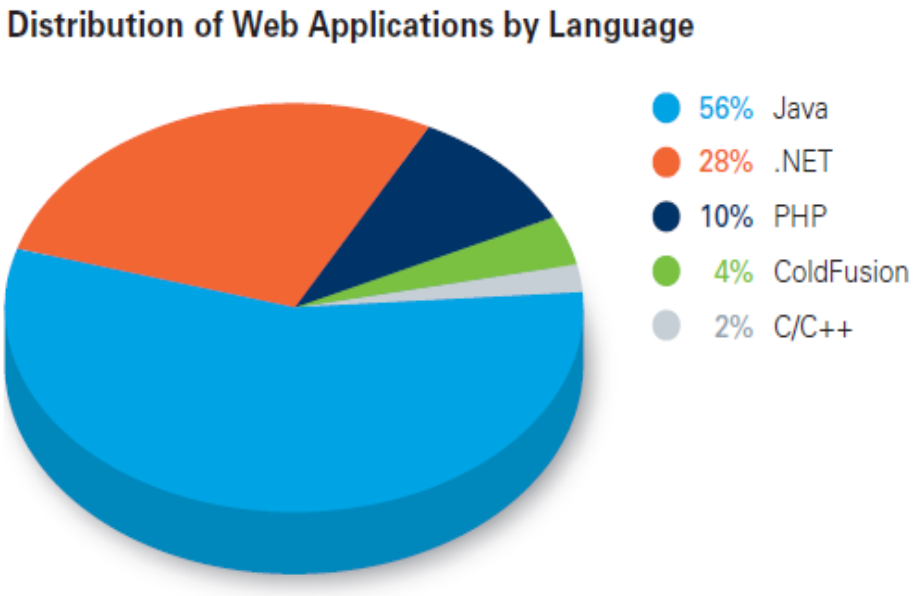

Figure 9. Languages more used for web applications [Veracode, 2012]

For building a web application, several language types are usually used:

- **Marked languages** as HTML, XHTML, HTML5, etc. for creating web pages and other information that can be displayed in a web browser. The can incorporate *script* code and can be downloaded from application servers and web servers or also from an AJAX engine.
- **Scripting languages** as PHP, *ColdFusion* or *javascript*, are not compiled and run interpreted inside the web server process. Those are used both in the client layer and in the server layer. The browsers usually have support for *javascript* [Javascript, 2013] code that is the language used by AJAX engines for building of *Rich Internet Applications.* The majority of scripting languages are slower than compiled programs; they are no strongly typed and do not promote good secure programming practices.

- **Specifications for web development** as J2EE [J2EE, 2013] or .NET [NET, 2013] for building large enterprise applications. Those specifications are a great set of features for implementing all different and necessary service requirements of web applications with a very good performance and with capacity of scaling as required.

2.3.3. PLATFORMS AND LANGUAGES FOR MOBILE APPLICATIONS

The furious rate of technological change and growth in the mobile market has made very challenging for developers to strategically plan a bespoke project, not only from a technical standpoint, but also because the market share for smartphones is changing rapidly between different systems.

Mobile Platforms. Until recently, the iPhone iOS dominated the mobile market, but Google Android has now demonstrably overtaken iPhone in terms of market share, due partly to the power of the Google brand and partly to the platform's openness. Other mobile operating systems include the Blackberry RIM OS and Windows 8. Figure 10 shows the distribution of mobile applications by platform [Veracode, 2012].

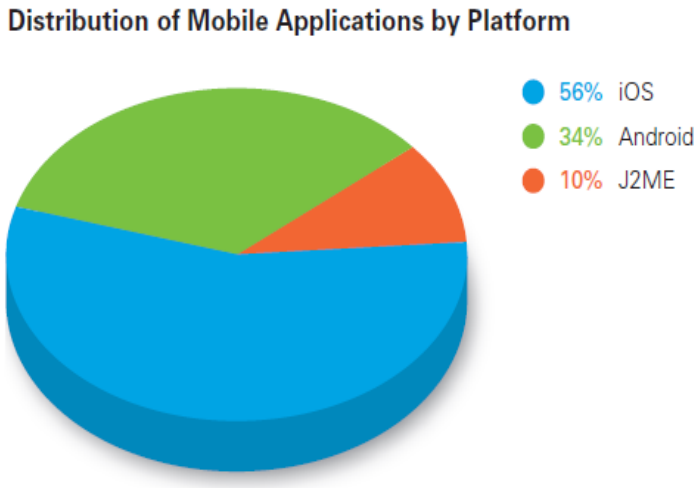

Figure 10. Distribution of mobile applications by platform [Veracode, 2012]

As well as the wealth of mobile platforms emerging, there are now more hardware manufacturers than ever producing mobile devices.

An additional complicating factor is the intense competition in mobile software, which is fuelling accelerated change within each of the platforms. Windows Mobile was replaced by Windows Phone 7, which has now become Windows 8. Both Apple iOS and Android have undergone significant changes with each release, making the task of supporting users a major undertaking, even for a single operating system.

Some of the platforms, Android in particular, are being deployed on devices produced by a long list of manufacturers, each of whom likes to modify the operating system to their particular requirements. Meanwhile, Android hardware is now available across a much wider market sector than ever before. In the early days of the smartphone, the technology was essentially only available to consumers shopping within the top price brackets. Now these handsets have become more accessible in terms of cost, and approximately 60 per cent of people in the UK currently use smartphones.

The result is that developers must consider a lengthy array of screen sizes, hardware specifications and configurations and ultimately a range of fundamentally differing models

Mobile applications types. The languages used to build mobile applications are the same that ones used for other devices as personal computers. There are two broad choices in deploying a system to mobile users [Mobile, 2013] [Smutny, 2012]:

- **Native apps** custom targeted at some or all of the major mobile platforms.
- **Web applications** optimized for mobile access.

There are a number of benefits and drawbacks to each approach, all of which need to be weighed up along with the specifics of any particular project. When considering how best to

incorporate mobile technology into an existing business model, the primary issue for both clients and developers is currently the choice between native apps and web applications or a combination of the two. The growth in mobile technologies has meant that businesses in certain sectors are even receiving most of their web traffic from users browsing in mobile contexts. When considering a mobile strategy, one of the big decisions for many clients is whether to focus resources on a Web application or on native apps targeted at specific mobile platforms. There is no "one size fits all" solution to this issue, because of the number of platforms operating, and the best approach for one business may be entirely different to that for another.

Native Apps targeted at specific mobile platforms advantages:

- Native access to the user interface creates a level of interaction that is difficult, if not impossible, to achieve through a Web browser.
- Native apps are in prime position to exploit the unique hardware and software facilities within mobile devices, such as GPS and localization tools, accelerometers and touch screens.

The primary consideration when implementing a service using mobile apps is the number of platforms. If targeting a sizeable chunk of the market is necessary, the resources required may be considerable.

Mobile apps distributed commercially through app stores are subject to sales transaction charges. The task of promotion and adoption by new users is also increasingly challenging, as many of the app stores, particularly Android, are becoming extremely overcrowded. For some purposes, a native app may be primarily used as a marketing resource, providing a supplementary service which highlights some larger branding or commercial objective.

Both short and long term native app development implications include:

- A diverse skill-set is necessary to develop apps for multiple platforms.
- There are significant maintenance implications, as the various operating systems, software and hardware contexts are in a constant state of flux.

Web applications accessible over the mobile network advantages:

- Only one system need be developed, optimized and enhanced to cater for users.
- Both development and maintenance are simpler and less labor intensive for a single application, even allowing for the enhancements necessary to cope with user platforms.
- The sophistication level in mobile Web browsers is advancing at such a rate that in some cases the gap between mobile and desktop functionality is diminishing.
- For commercial applications, early evidence suggests that consumers are more inclined to make purchases via mobile websites than native apps.

The fundamental consideration when focusing on the mobile Web is that, while there are tremendous advantages from both development and deployment perspectives, the network technologies and infrastructure have some way to come yet, in terms of specification and support at the client end.

Web applications are also somewhat limited in terms of both hardware exploitation and user interaction**. Innovative uses of scripting** can approximate a native experience within a web application, for example via HTML5 and jQuery. However, the native app presently has the ability to create a much more intuitive and immersive user experience.

Mobile web application languages. Emerging new and recent technologies are being used for mobile application development:

- **HTML5** [HTML5, 2013] is a developing standard which is already starting to have a dramatic impact on the mobile Web. Although the final specification is not expected for several years yet, some major sites have already begun focusing on HTML5, for example the Financial Times newspaper, which released a Web application instead of device-specific apps.
- **JQUERY** [Jquery, 2013] Mobile Web development relies heavily on external libraries and support tools some commercial, some open-source. One of these is *jQuery Mobile* an enhanced version of the *jQuery JavaScript* platform that is optimized for touch interaction. *JQuery* is a *JavaScript* library which makes cross-browser programming easier. The mobile version provides a unified set of user interface tools designed to be compatible across mobile browsers. Additional scripting tools combine with *jQuery* to create improved user interfaces for mobile Web applications, coming some way towards a native app experience on the Web. Other libraries include *The-M-Project, Mobi* and *jQTouch.*

In general, the extent to which Web technologies, including HTML5, can make use of valuable native mobile hardware and software tools is limited, but will almost certainly improve within the next few years. A comparative about development features (table 7) is provided by Manuel Palmieri [Palmieri, 2012].

Table 7

Comparison about development features [Palmieri, 2012]

Name	Language	Accessibility to native API's	IDE	Plug-in Extendibility
RhoMoblie	HTML, HTML5, CSS, JavaScript,	JavaScript	RhoStudio RhoHub, *	YES
PhoneGap	HTML, HTML5, CSS, CSS3, JavaScript	JavaScript	IDE native of the mobile OS (e.g Eclipse, Xcode)	YES

DragonRad	D&D	Na	DragonRad Designer	NO
MoSync	HTML, HTML5, CSS, JavaScript,	JavaScript, C/C++	Based on Eclipse	YES

There are ways in which applications can adopt some of the characteristics of both Web and native apps. In general, the loading issues in mobile devices require efficient applications to adopt well-defined coping strategies such as:

- Using minimal HTTP requests.
- Carrying out as much processing as possible at server side.
- Generally minimizing data and media content.

When these practices are adopted, there is increased scope for focusing platform specific development on creating lightweight interfaces, with server side processing usable across platforms. This model can allow projects to better maximize on development resources, while still catering for multiple user environments. For many organizations, deploying both native and Web apps is still seen as necessary. Users are still using them both, and in subtly different ways. There are also specific cases in which it makes sense to target one or more platforms with dedicated apps, where the unique features of that platform have heightened relevance, for example with the superior level of Google Maps support on the Android platform.

2.4. CONCLUSIONS.

Automatic security analysis tools must be designed according to the last tendencies in architecture and most used languages in software development. These tools must enable to

perform security analysis of client-server, standalone, web, web services and mobile applications. The choice of a specific language for application development must be based on the knowledge of language security characteristics.

To design automatic security analysis tools, it is important to take into account several facts:

- There are many different languages to use in application development.
- There are specific languages for specific architecture software as web, mobile applications or embedded software.
- Web and mobile development is increasing continuously over time.

The knowledge of architecture and languages for software development leads to the study of the state of art of these security tools. This study will help us to select the most adequate tools to be included in the assessments that are a partial objective of this thesis.

3. APPLICATIONS SECURITY PROBLEMS

Application vulnerabilities open the door to malicious data, scripts or code with the objective of capturing sensible data or remote unauthorized access to the application. These attacks can materialize in many diverse forms depending on the nature of vulnerability. **In general, an attack is any malicious act against a system or set of systems**. There are two very important concepts in this definition that it is worth clarifying. First, the act is done with malicious intent, without specifying any goals or objectives. Second, some attacks target a particular system, while others have no particular goal. The attacks may be based on defects, which may occur at any stage of the life cycle of software and systems development. According to [Cheswick, 2003], and taking into account any kind of vulnerability, the attacks can be of various types**:**

- Stealing Passwords
- Social Engineering
- Software vulnerabilities and Back Doors
- Authentication Failures
- Protocol Failures
- Information Leakage
- Exponential Attacks Viruses and Worms
- Denial-of-Service Attacks
- Botnets
- Active Attacks

A number of factors (see figure 11) are involved when a threat materializes in an application, exploiting an existing vulnerability and generating business impact:

- **Agent or threat**, the person making the attack.

- **Attack Vectors**, used to carry out the attack.
- **Security weakness vulnerability**
- Absence or **failure to control**.
- **Impact on any asset** of the information systems of the organization.
- **Impact on the business** of the Organization

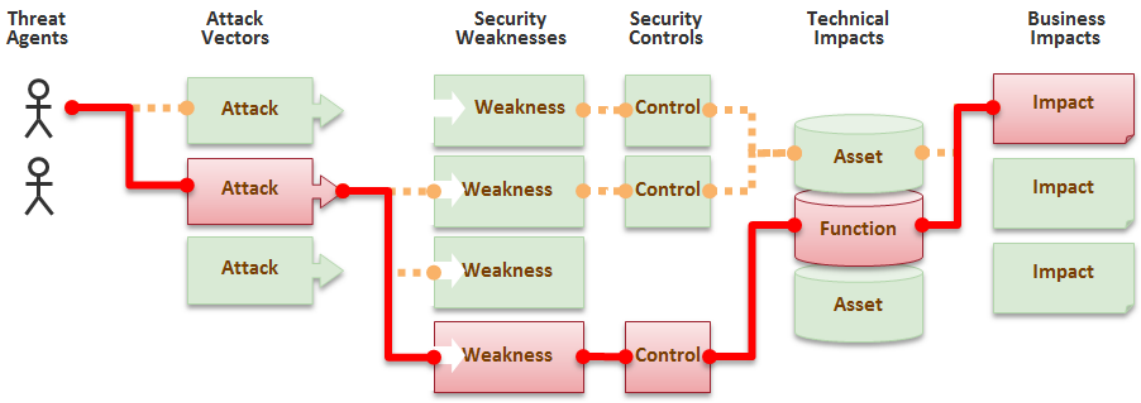

Figure 11. Materialization of a threat: Attack [Owasp, 2013]

Open security projects and security standards organizations, as OWASP [Owasp, 2013] and SANS [Sans, 2013], publish regularly the most dangerous security vulnerabilities. An illustrative example, with respect to applications security, is the study of State of Software report Volume 5 [Veracode, 2012], based in OWASP TOP 10 and SANS TOP 25 vulnerabilities classifications. Figure 12 illustrates the compliance upon initial application submission against two standard policies.

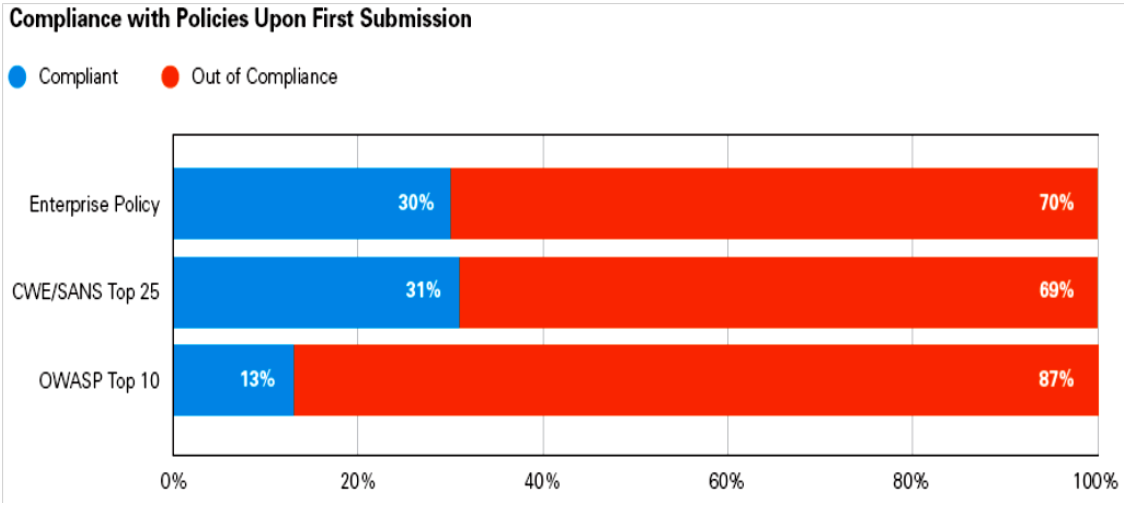

Figure 12. Applications compliance with Policies upon First Submission [Veracode 2012]

Web applications are assessed against the OWASP Top 10 and only 13% complied on first submission. Non-web applications are assessed against the CWE/SANS Top 25 and 31% complied on first submission. Only 30% of applications complied with enterprise defined policies. Compliance with policies upon first submission of an application can be a good indicator of the success or failure of "building-in" security as part of the software development lifecycle (SDLC).

According to CVE MITRE [CVE, 2013] a security vulnerability is a mistake in software that can be directly used by a hacker to gain access to a system or network. The types of vulnerabilities can be defined in terms of each phase of SSDLC. These include commonly accepted design, implementation and operational vulnerabilities according to *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* [Dowd 2006].

The security community generally accepts **design vulnerabilities** as defects in the software system architecture and specifications. They can be found in the requirements analysis phase, or in the specifications of the design phase.

The **implementation vulnerabilities** category logically refers to security errors done by developers when developing modules or objects of the system to meet the specifications.

The **operational vulnerabilities** category refers to security defects that arise in the deployment and configuration of the system developed in a particular environment

The following sections will analyze the security design aspects of applications architecture and will show both statistics about vulnerabilities detected in applications and statistics about materialized attacks in applications exploiting existing vulnerabilities.

3.1. SECURITY DESIGN ASPECTS OF ARCHITECTURE.

The security design of an application must be accomplished from the beginning of SSDLC. After the phase of security requirements analysis of an application, a selection for its archetype and design pattern must be done, based also in safety in addition to other aspects. All software applications types can have vulnerabilities in the code of all architecture tiers and design vulnerabilities in its architectural platforms and components. The number of attacks that an application can suffer depends on vulnerabilities in:

- Software architecture components and tiers.
- Platform and operating system.
- Client software security including operating system.
- Application Servers.
- Network.
- Database Management System.
- Development technology
- Programming languages.
- Security experience and knowledge of programmers.

- Online protections.

The individual security objectives can be used to divide the application architecture for further analysis, and to help identifying the application vulnerabilities. This approach leads to a design that optimizes the following security objectives:

- **Authentication**. Authentication is the process where one entity definitively establishes the identity of another entity, typically with credentials such as a username and password.
- **Authorization**. Authorization refers to how an application controls access to resources and operations.
- **Configuration Management**. Run context of the application, databases it connects, the way that the application is administered and how the resources are protected, Configuration management refers to how an application handles these operations and issues.
- **Confidentiality.** The application must protect the secrets, confidential user and application data and handle properly sensitive data. Sensitive data refers to how the application handles any data that must be protected either in memory, over the network, or in persistent storage. It is usually done by using cryptographic algorithms.
- **Integrity**. The application must check data or libraries for alterations. Random values must be cryptographically strong. It is also done by using cryptography.
- **Availability**. Maintain availability of information handled by a system or its resources.
- **Non-repudiation**. Provide proof that a particular transmission or reception has been made, the receiver / transmitter cannot deny that it occurred.

These security objectives can be used to make key security design decisions for an application, and document these as part of the architecture. For example, Figure 13 shows the security issues identified in a typical Web application architecture.

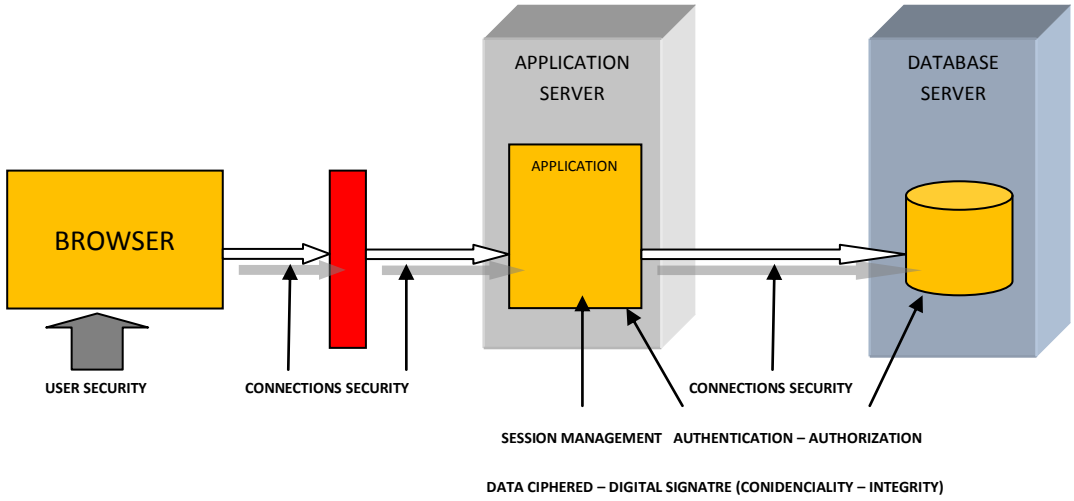

Figure 13. Security issues identified in a typical Web application architecture at SSDLC design phase.

The **Security Policy** of an organization is a set of rules that govern and determine what to do and what not in it. According to the IETF is defined as: "*A series of formal statements (rules) to which all people have access to any information organization and technology*" Some of the most important features of any security policy can be highlighted [IETF, 1997]:

- It sets which tools are needed and which procedures.
- It is used to communicate a consensus on the use of data and applications within the organization.
- It provides a basis for demonstrating the inappropriate use of resources, by employees or external.
- It defines the appropriate behavior for each case.

The security policy of an organization is of dynamic nature, always being updated, which allows taking into account that the maintenance of security is a living process and must be manageable in a structured and organized way. So security is a process that has to be achieved through the development of security policy. The policy must be understood as something dynamic that must be updated by following a number of **security principles.** Many security experts talk about security principles that should govern all design, most of them overlap and generally coincide thus being similar. Michael Howard and David LeBlanc promulgated in Writing Secure Code [Howard, 2003] principles as:

- Secure by design
- Secure by default
- Secure in deployment
- Principle of least privilege
- Principle of depth on defense
- Principle of diversity of defense
- Identifying weaknesses
- Centralized security management
- Principle of simplicity
- Learn from errors
- Reducing the attack surface to the minimum
- Use of default security
- Assume that external services are insecure
- Having plans in case of failure
- Fail to secure mode
- Secure components are not secure
- Do not mix code and data
- Fix security issues correctly

- Never depend on security through obscurity alone
- Backward compatibility will always give you grief

3.2. ARCHITECTURE SPECIAL CASES EXAMPLES.

The choice of a specific application category, archetype and architectural style has different security implications that must be analyzed, studied and tested to make the best secure selection possible. This section will show three examples of several applications types and architecture styles with different design security implications.

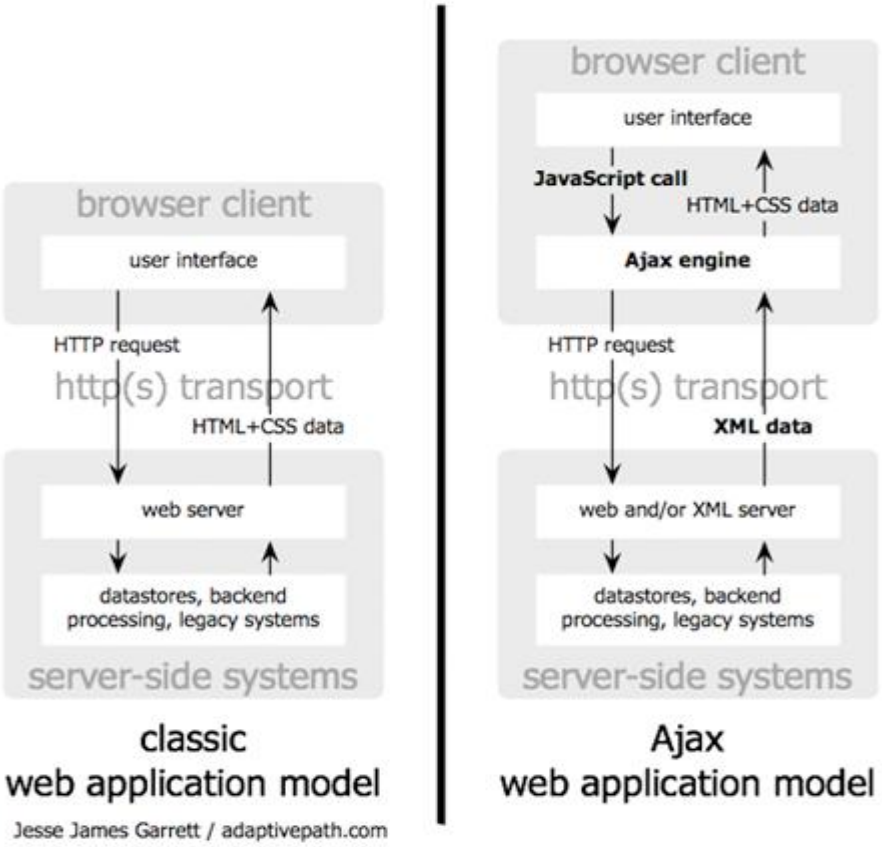

Figure 14. Ajax against traditional web applications architectures [Ajax, 2013]

By example, with respect to **application types**, the Ajax Rich Internet Applications have the advantage of providing richer client interfaces and the asynchronous communications

with the server makes it quicker. However, this type of application introduces new sources of security vulnerabilities in the code of the Ajax engine that the client runs as, for example, cross site scripting (XSS) and violations of same origin policy accessing other not allowed domains (figure 14).

Another example, with respect to the selection of **architectural style,** is Service Object Architecture (SOA). The entities involved in the provision of a service must implement the same security standards for providing all security goals as authentication, authorization, confidentiality, etc. This requisite can be difficult to achieve when heterogeneous entities have to interact themselves (Figure 15).

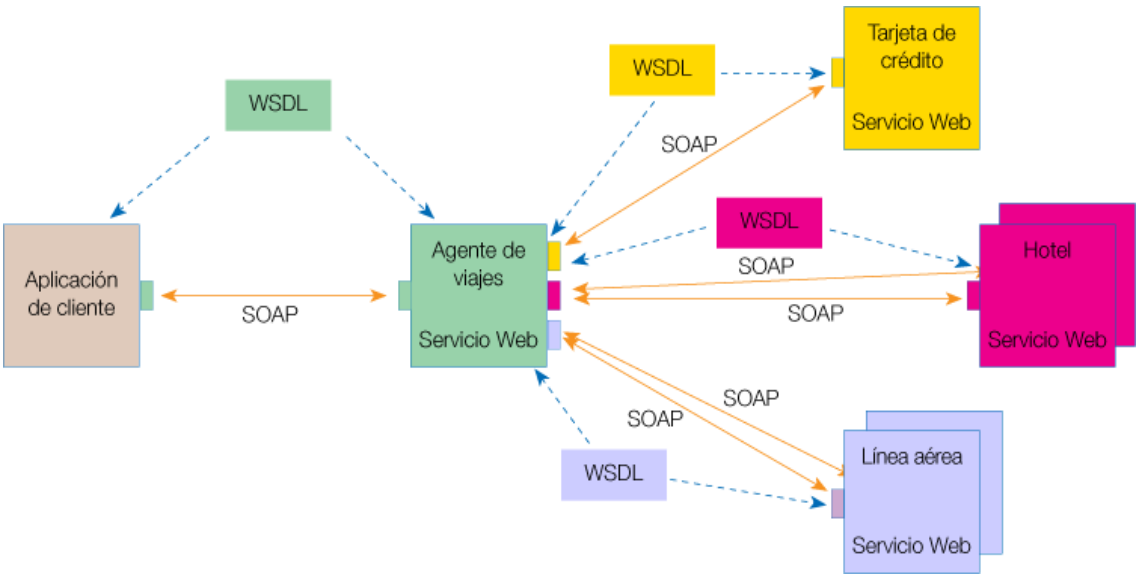

Figure 15. Web services architecture example [W3c, 2013]

Finally it is worth commenting on the characteristics of **System Control Architecture and Data Acquisition (SCADA)** and **Industrial Control Systems (ICS)** that are being attack objectives in recent years. According to the statistics [HP-report, 2012] SCADA systems which control automated industrial processes such as manufacturing, power generation, mining, and water treatment, rely on considerably more mature technology. These systems, which have historically operated over separate networks with proprietary protocols, have

begun to migrate to standard networks, and even the Internet, to simplify asset management, billing, and operations. As these systems have moved off their separate isolated networks, security problems that were once masked by a restricted attack surface have begun to manifest themselves. One of the most popular remote attack vectors on SCADA is fingerprinting of exposed industrial devices by its front-ends and server-side components through application protocols (HTTP, UPnP, SNMP, FTP, SSH, Telnet). It helps the hackers to detect critical infrastructures, as well as signatures of smart-metering devices, HVAC, medical devices. Projects as OWASP Scada Security Project [OWASP, 2013] have the objective of gathering information about the ways of improving the security measures in modern ICS environments and to create guidelines for its hardening. These systems control nuclear, electrical plants and other similar installations that need a 24x7 work continuously. The natural importance of this installation type requires a detailed and planned security design during its implantation during SSDLC. Only 76 vulnerabilities were disclosed in SCADA systems from 2008 through 2010. However, after the Stuxnet worm was discovered in an Iranian uranium enrichment plant in 2010 [Schneier, 2010], much attention has been focused on the security of SCADA systems. In 2011, there were 164 vulnerabilities disclosed in SCADA systems, and the number rose again to 191 in 2012, representing a 768 percent increase from 2008 numbers [HP-report, 2012]. A recent paper "*SCADA security in the light of Cyber-Warfare"* surveys ongoing research and provides a coherent overview of the threats, risks and mitigation strategies in the area of SCADA security [Nicholson, 2012]. A guide to get SCADA and ICS systems more secure can be found in special publications of NIST [SP-800-82, 2011].

3.3. SOFTWARE VULNERABILITIES

The implementation phase of SSDLC accomplishes the code development of the entire application. There are several aspects relative to security that must be developed in a correct secure pattern to accomplish with the **security objectives during the implementation** phase (figure 13):

- **Exception Management**. How the application acts when a method call fails and information is shown in error messages to end users. If it passes valuable exception information back to the calling code. If it fails gracefully. If it helps administrators to perform root cause analysis of the fault. Exception management refers to how exceptions are handling within the application.
- **Input and output Data Validation**. All inputs and outputs of the application must be validated. The content of all input data must be validated to check arrays length, check malicious content, data sources such as databases and file shares, etc.
- **Session Management**. The application must handle and protect user sessions using strong identifiers with solid random numbers, using a new identifier for each new session, with a session timeout by default, etc.
- **Auditing and Logging**. The application must register and audit security-related events to report how its continued operation is. Logging refers to how the application publishes information about its operation. The information to reveal should be the minimum necessary.

There are Organizations like OWASP, SANS and WASC that have as one of its goals to raise awareness about application security by identifying some of the most critical risks facing organizations. The OWASP Top 10 project is referenced by many standards, books, tools, and organizations, including MITRE, PCI DSS, DISA, FTC, and many more. The

2103 release of the OWASP Top 10 marks this project's tenth anniversary of raising awareness of the importance of application security risks. The OWASP Top 10 was first released in 2003, with minor updates in 2004 and 2007. The 2010 version was revamped to prioritize by risk, not just prevalence. The 2013 edition follows the same approach. Table 8 shows the top ten most dangerous vulnerabilities, according to OWASP [Owasp, 2013], comparing the top ten 2010 with the top ten 2013 to observe the evolution in a three years period:

Table 8
OWASP TOP TEN 2010 vs. 2013 vulnerabilities [Owasp, 2013]

OWASP TOP TEN 2013	OWASP TOP TEN 2010
A1-Injection	A1-Injection
A2-Broken Authentication and Session Management	A2-Cross Site Scripting (XSS)
A3-Cross-Site Scripting (XSS)	A3-Broken Authentication and Session Management
A4-Insecure Direct Object References	A4-Insecure Direct Object References
A5-Security Misconfiguration	A5-Cross Site Request Forgery (CSRF)
A6-Sensitive Data Exposure	A6-Security Misconfiguration
A7-Missing Function Level Access Control	A7-Insecure Cryptographic Storage
A8-Cross-Site Request Forgery (CSRF)	A8-Failure to Restrict URL Access
A9-Using Components with Known Vulnerabilities	A9-Insufficient Transport Layer Protection
A10-Unvalidated Redirects and Forwards	A10-Unvalidated Redirects and Forwards

Table 9 gives a short description of OWASP top ten 2013 vulnerabilities.

Table 9

OWASP TOP TEN 2013 vulnerabilities description [Owasp, 2013]

Vulnerability	description
A1-Injection	Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization
A2-Broken Authentication and Session Management	Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities
A3-Cross-Site Scripting (XSS)	XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites
A4-Insecure Direct Object References	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data
A5-Security Misconfiguration	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date
A6-Sensitive Data Exposure	Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser
A7-Missing Function Level Access Control	Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization
A8-Cross-Site Request Forgery (CSRF)	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are

	legitimate requests from the victim
A9-Using Components with Known Vulnerabilities	Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts
A10-Unvalidated Redirects and Forwards	Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages

SANS TOP 25 [Sans, 2013] is another project that deals with classifying vulnerabilities of applications, in terms of their importance related with the frequency with which they occur in the applications and the danger of the attacks that exploit them. In this project, the vulnerabilities are classified into three categories with expression of MITRE CWE (Common Weakness Enumeration) identifier (Table 10). Common Weakness Enumeration is a formal list or dictionary of common software weaknesses that can occur in software's architecture, design, code or implementation that can lead to exploitable security vulnerabilities. CWE was created to:

- Serve as a common language for describing software security weaknesses.
- Serve as a standard measuring stick for software security tools targeting these weaknesses
- Provide a common baseline standard for weakness identification, mitigation, and prevention efforts. Software weaknesses are flaws, faults, bugs, vulnerabilities, and other errors in software implementation, code, design, or architecture that if left unaddressed could result in systems and networks being vulnerable to attack.

Example software weaknesses include: buffer overflows, format strings, etc.; structure and validity problems; common special element manipulations; channel and path errors; handler

errors; user interface errors; pathname traversal and equivalence errors; authentication errors; resource management errors; insufficient verification of data; code evaluation and injection; and randomness and predictability.

MITRE CWE "*provides a unified, measurable set of software weaknesses that is enabling more effective discussion, description, selection, and use of software security tools and services, that can find these weaknesses in source code and operational systems, as well as a better understanding and management of software weaknesses related to architecture and design"* [Mitre, 2013].

Table 10
SANS TOP 25 Vulnerabilities. [Sans, 2013]

SANS TOP 25 Vulnerabilities.	
Software Error Category: Insecure Interaction Between Components (6 errors)	
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-434	Unrestricted Upload of File with Dangerous Type
CWE-352	Cross-Site Request Forgery (CSRF)
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
Software Error Category: Risky Resource Management (8 errors)	
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-494	Download of Code Without Integrity Check
CWE-829	Inclusion of Functionality from Untrusted Control Sphere

CWE-676	Use of Potentially Dangerous Function
CWE-131	Incorrect Calculation of Buffer Size
CWE-134	Uncontrolled Format String
CWE-190	Integer Overflow or Wraparound
Software Error Category: Porous Defenses (11 errors)	
CWE-306	Missing Authentication for Critical Function
CWE-862	Missing Authorization
CWE-798	Use of Hard-coded Credentials
CWE-311	Missing Encryption of Sensitive Data
CWE-807	Reliance on Untrusted Inputs in a Security Decision
CWE-250	Execution with Unnecessary Privileges
CWE-863	Incorrect Authorization
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-307	Improper Restriction of Excessive Authentication Attempts
CWE-759	Use of a One-Way Hash without a Salt

Finally, for web applications, another good vulnerabilities classification is the one of the Web Application Security Consortium (WASC) [Wasc, 2013] that explains the vulnerabilities and problems most characteristic in web applications. The greatest contribution of WASC is the variety of projects developed and in course about web application security. Special importance for this thesis is the Static Analysis Technologies Evaluation Criteria project, which is an excellent starting point to establish the main objectives to perform a SAST assessment.

3.4. VULNERABILITIES AND ATTACKS TENDENCIES.

This section will show:

- Statistics about vulnerabilities detected in applications obtained by companies and organizations by using manual and automatic methods and
- Statistics about materialized attacks in applications exploiting existing vulnerabilities.

The two types of statistics are distinct because not all vulnerabilities that an application has are exploitable or, even when a vulnerability is exploitable, an attack never is materialized on it because the vulnerability is not discovered. It is important to take into account both the vulnerabilities that an application has and the most dangerous and frequent attacks in order to perform an assessment of the detection of vulnerabilities and its posterior patching. The process of the patching for an application should be based on the most dangerous and frequent attacks.

3.4.1. VULNERABILITIES TENDENCIES.

This section shows vulnerabilities trend statistics of two reports from two important companies:

- HP cyber risk report 2012 [HP-report, 2012]
- Veracode State of software security report volume 5 [Veracode, 2102]

HP 2012 Cyber Risk Report, HP Enterprise Security provides a broad view of the vulnerability landscape, ranging from industry-wide data down to a focused look at different technologies, including Web and mobile. The goal of this report is to provide the kind of actionable security that intelligence organizations need to understand the vulnerability landscapes as well as best deploy their resources to minimize security risk. The report offers the following **key findings**:

- **Critical vulnerabilities are on the decline, but still pose a significant threat.** CVSS (Common Vulnerability Scoring System) is a vulnerability scoring system designed to provide an open and standardized method for rating IT vulnerabilities. High-severity vulnerabilities (CVSS score of 8 to 10) made up 23 percent of the

total scored vulnerabilities submitted to OSVDB (Open Source Vulnerability DataBase), in 2011 and dropped to 20 percent in 2012. OSVDB is an independent and open sourced web-based vulnerability database created for the security community. The goal of the project is to provide accurate, detailed, current, and unbiased technical information on security vulnerabilities) [Osvdb, 2013]. While this reduction is significant, the data shows that nearly one in five vulnerabilities can still allow attackers to gain total control of the target.

- **Mature technologies introduce continued risk.** As demonstrated by the recent Department of Homeland Security announcement recommending that the Oracle Java SE platform be universally disabled in Web browsers, seemingly mature technologies still suffer from new exploits. In particular, as commented before, 2012 data show the number of vulnerabilities disclosed in Supervisory Control And Data Acquisition (SCADA) systems increased from 22 in 2008 to 191 in 2012 (a 768 percent increase).
- **Mobile platforms represent a major growth area for vulnerabilities.** The explosive adoption of mobile devices and the applications that drive them has resulted in a corresponding boom in mobile vulnerabilities. The last five years have seen a 787 percent increase in mobile application vulnerability disclosures, with novel technologies, such as near field communications (NFC), introducing previously unseen vulnerability types.
- **Web applications remain a substantial source of vulnerabilities.** OSVDB data from 2000 to 2012 shows that of the six most submitted vulnerability types, four (SQL injection, cross-site scripting, cross-site request forgery and remote file includes) exist primarily or exclusively in Web applications.
- **Cross-site scripting remains a major threat to organizations and users.** Cross-site scripting (XSS) remains a widespread problem, with 44.5 percent and 44

percent of the applications in our data sets suffering from the vulnerability. In one case, analysis of a multinational corporation showed that just under half (48.32 percent) of their Web applications were vulnerable to some form of XSS. Furthermore, new methods of exploiting this vulnerability continue to be found.

- **Effective mitigation for cross-frame scripting remains noticeably absent.** The first documented cross-frame scripting (XFS) vulnerability, the root cause behind *clickjacking* attacks, was discovered over 10 years ago. Since then, *clickjacking* has become a household name, yet less than one percent of 100,000 URLs tested included the best-known mitigation, the X-Frame-Options header.

The HP cyber risk report also analyzes the **vulnerability trends.** Understanding technical security risk begins with knowing how and where vulnerabilities occur within an organization. This section of the report uses data from the Open Source Vulnerability Database (OSVDB) [Osvdb, 2013] and the HP Zero Day Initiative (ZDI) [Zdi, 2103] to demonstrate the following global vulnerability trends:

- **The vulnerability arms race total vulnerability disclosures in 2012 increased 19 percent from 2011.** The total number of vulnerabilities reported provides a snapshot into the world of vulnerabilities and serves to illustrate the nature of a constantly changing threat landscape, as seen in figure 16.

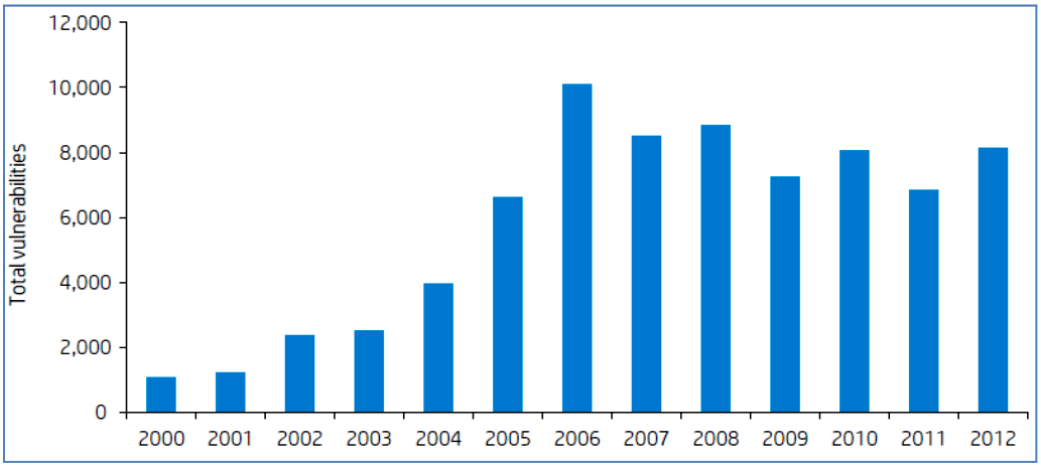

Figure 16. Vulnerabilities trend [HP-report, 2012]

- **Evolving marketplaces and increasing complexity impact discovery and reporting.** Vulnerability disclosure data highlights how changes in the vulnerability marketplace and the technical complexity of systems impact both the number and severity of reported vulnerabilities.
- **Web applications continue to introduce significant technical risk to organizations.** A small number of critical Web application vulnerabilities still represent a large minority of the overall vulnerabilities disclosed in 2012.
- **The maturity of a technology does not govern its vulnerability profile.** Data in 2012 shows an increase of more than 700 percent in vulnerability disclosures impacting both SCADA systems (primarily legacy technology), see figure 17, and mobile devices (the next frontier for IT).

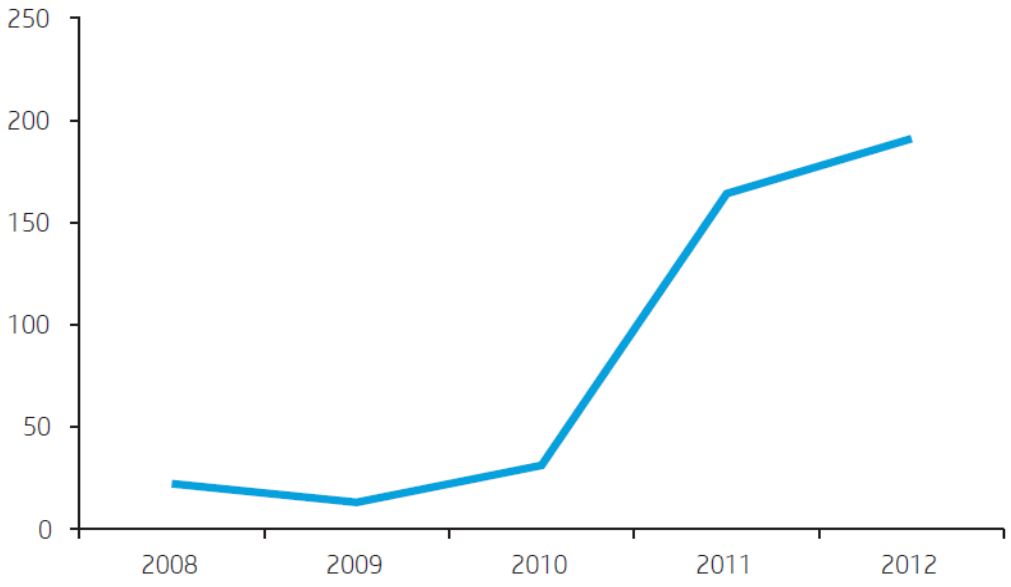

Figure 17. SCADA Vulnerabilities trend [HP-report, 2102]

Table 11 shows the top 10 mobile vulnerabilities.

Table 11. Top 10 mobile vulnerabilities in 2012 [HP-report, 2012]

Top 10 mobile vulnerabilities in 2012	
Unauthorized access	18%
Cross site scripting	15%
Insecure session handling	11%
Cookie handling vulnerabilities	9%
Improper encryption	9%
Poor logging practices	8%
Autocomplete on sensitive form fields	6%
Cleartext credentials	6%
Poor error messages	6%

Veracode state of software security report volume 5 [Veracode, 2012] examines data collected over an 18 month period from January 2011 through June 2012 from 22,430 application builds uploaded and assessed by its platform. The principal findings are the following:

- 70% of applications failed to comply with enterprise security policies on first submission.
- SQL injection prevalence has plateaued, affecting approximately 32% of web applications.
- Eradicating SQL injection in web applications remains a challenge as organizations make tradeoffs around what to remediate first.
- Cryptographic issues affect a sizeable portion of Android (64%) and iOS (58%) applications.

Veracode "state of software security report volume 5" shows statistics of vulnerabilities trend for non-web applications, web applications and mobile applications.

- **State of Web Application Security.** Figure 18 shows how the top ten vulnerability categories for web applications have varied over the last three states of software security reports of Veracode Company.

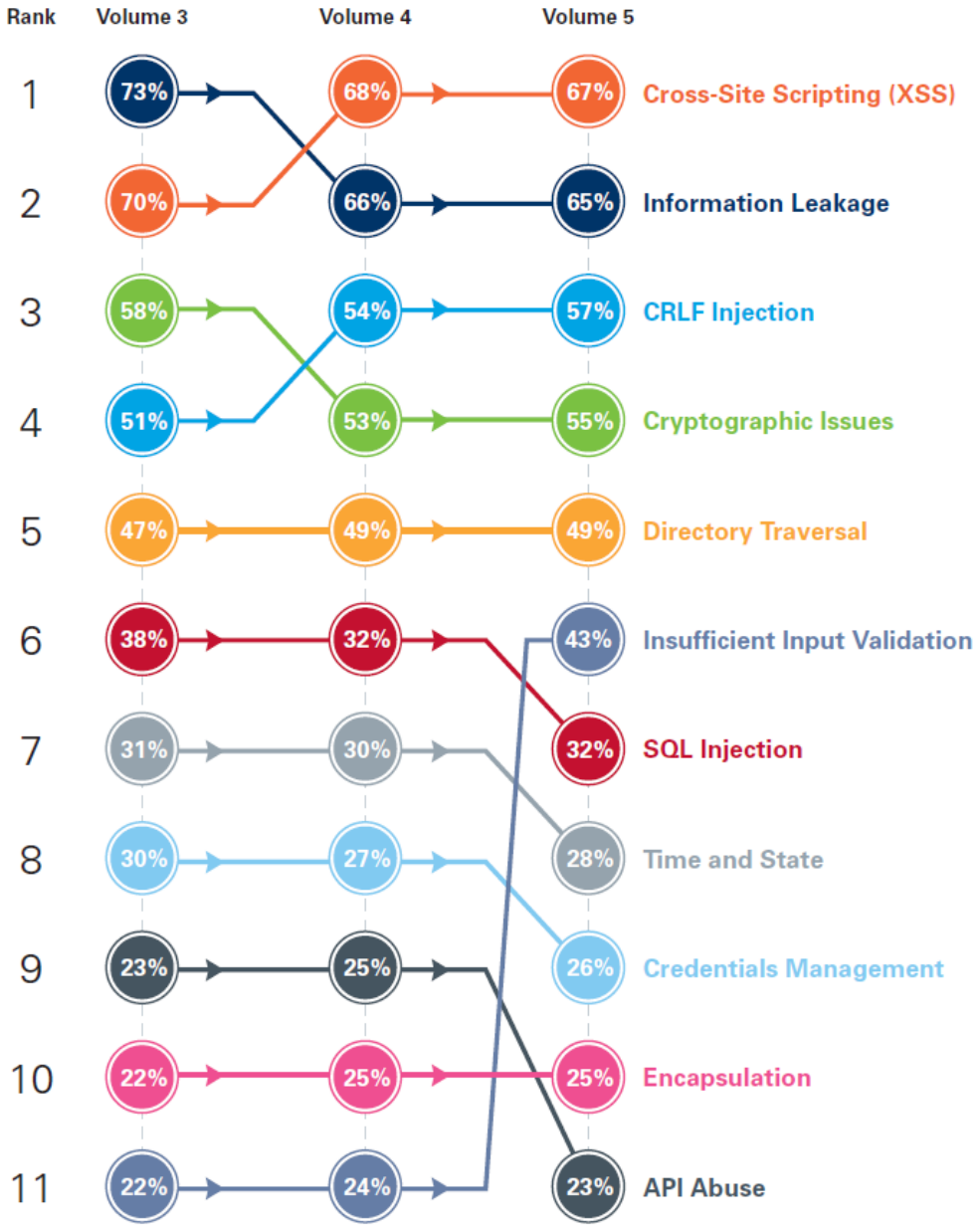

Figure 18. Top Vulnerability Categories (Percentage of Affected Web Application Builds) [Veracode, 2012]

Not much has changed. The top five categories remain the same as Volume 4. Cross-site scripting and information leakage are at the top with 67% and 65% respectively. Volume 5 reporting includes two additional CWE categories associated with the insufficient input validation category which vaulted the category to sixth place. API abuse dropped just out of the top ten.

- **State of Non-Web Application Security.** Figure 19 shows the trends in the top vulnerability categories for non-web applications over the last three Volumes. Cryptographic issues and directory traversal have remained the top two vulnerability categories for the last three Volumes, affecting 47% and 38% of all non-web applications in the current reporting period. Information leakage (26%) takes the third spot from error handling, which drops to fourth place. Buffer overflow dropped out of the top ten for the first time in this volume, and is replaced by SQL injection which is now affecting 16% of non-web applications. The good news is that the percentage of applications with buffer management errors is declining, from 20% in Volume 3 to 13% in Volume 5. However, the rise in the percentage of applications containing information leakage and SQL injection vulnerabilities is disturbing since applications are the conduit through which attackers gain access to confidential or proprietary information. It is noteworthy that the percentages reported in Figure 19 are generally lower than those reported in the software supply chain feature supplement published in November 2012. For example, cryptographic issues affected 62% of vendor supplied applications but only 47% of all applications (which include internally developed, outsourced, and open source applications in addition to vendor supplied applications). The relatively higher percentages reported in the supplement demonstrate the need for vendors to continue to work towards developing more secure software.

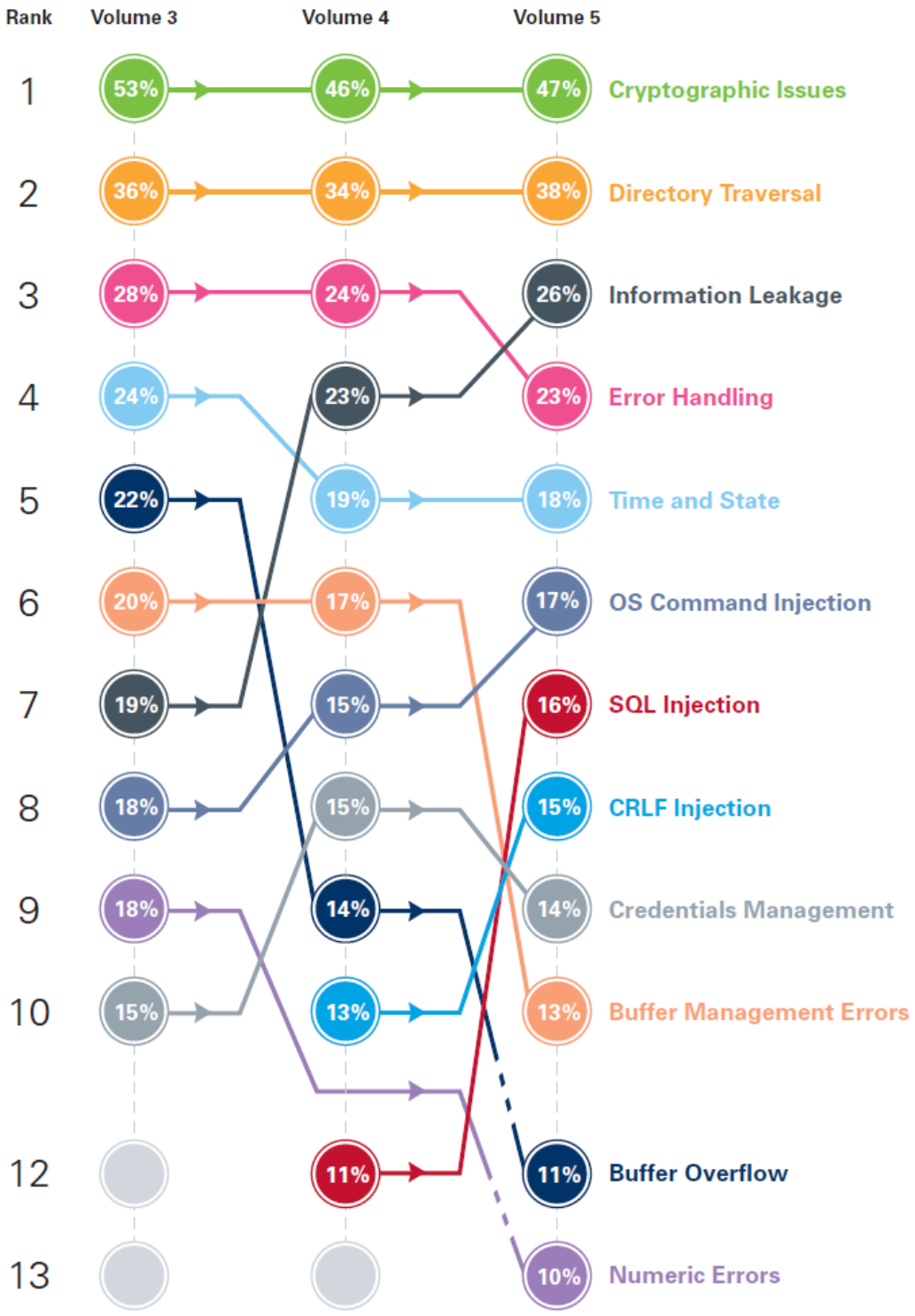

Figure 19. Top Vulnerability Categories (Percentage of Affected NON-Web Application Builds) [Veracode, 2012]

- **State of Mobile Application Security.** First they examine the vulnerability distribution in terms of share of total vulnerabilities discovered across all application builds associated with each mobile platform. In Volume 5, there is

enough data on all three platforms to provide a statistically sound basis for comparison. Table 12 shows that all three mobile platforms that they analyzed share cryptographic issues and information leakage in the Top 5 list of vulnerabilities, as measured by percent of total vulnerabilities found. As jailbreaking becomes more common practice and new features such as surviving reboots are supported, cryptographic issues significantly weaken data protection. Attackers with physical control of a mobile device for a small amount of time can jailbreak it and install a backdoor with keyloggers or other malware and/or copy the content. Both cryptographic issues and information leakage vulnerabilities increase the attack surface for mobile applications, and are two of Cloud Security Alliance's top five identified threats to mobile devices.

Table 12
Share of total vulnerabilities found in mobile applications [Veracode, 2012]

Mobile applications vulnerabilities					
ANDROID		IOS		JAVA ME	
CRLF injection	37%	Information leakage	62%	Crytographic issues	47%
Crytographic issues	33%	Error handling	20%	Information leakage	47%
Information leakage	10%	Cryptographic issues	7%	Directory traversal	3%
SQL injecgtion	9%	Directory traversal	6%	Poor input validation	2%
Time and state	4%	Buffer management	3%º	Credentials management	<1%

Veracode state of software security report volume 5 [Veracode, 2012**]** contains *java, .NET. C/C++, PHP and COLDFUSION* languages analysis trends from report volume 3 to

volume 5. By example the figure 20 shows trends of *java* vulnerabilities percentage of the total and figure 21 shows percentage of applications affected by vulnerabilities:

- Java vulnerabilities percentage:

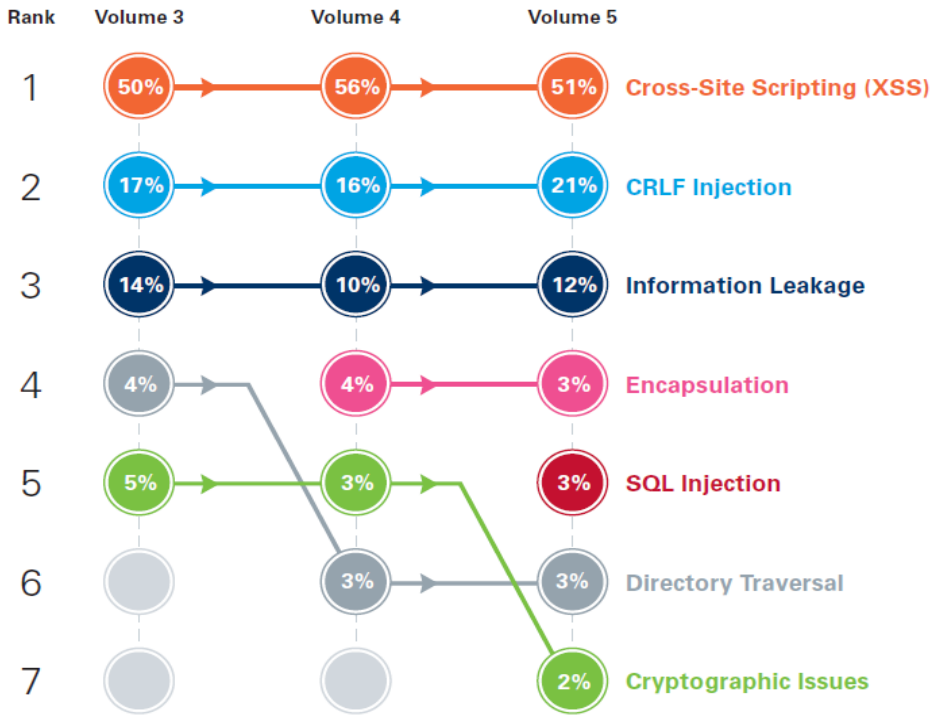

Figure 20. Share of total vulnerabilities found trends for Java Applications [Veracode, 2012]

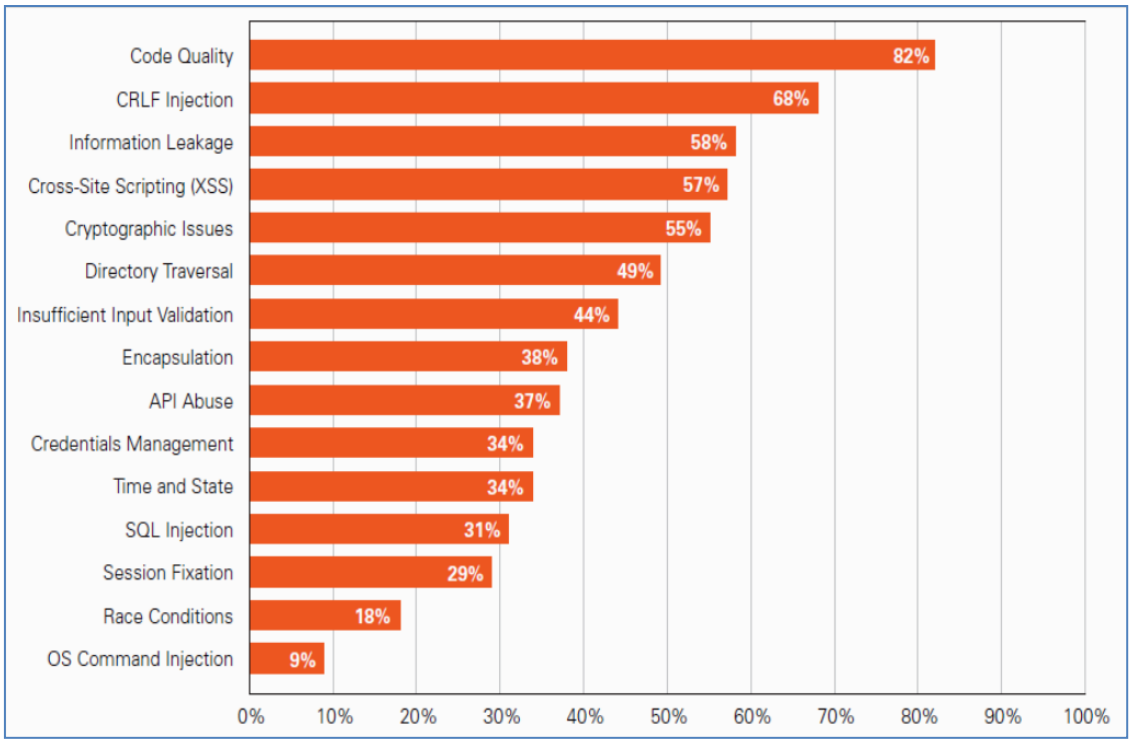

Figure 21. Percentage of Java Applications Affected [Veracode, 2012]

3.4.2. ATTACKS TENDENCIES.

This section shows how the attacks materialize by exploiting vulnerabilities, the methods used and the most used vulnerabilities. To clarify the last tendencies in recent attacks incidents, this section will analyze the results from 2013 Trustwave Global Security Report [Trustwave, 2013]. This report analyzes the results of hundreds of incident response investigations, thousands of penetration tests, millions of website and Web application attacks and tens of billions of events. It also includes detailed contributions from law enforcement agencies and experts from around the world. All in an effort to provide with perspectives on the latest threats and vulnerabilities facing organizations, along with actionable recommendations you can begin implementing immediately to strengthen your security program. The knowledge of how, when and the number of attacks is a valuable

resource for business, helping defend better, act faster and prepare for what's ahead in the upcoming future and beyond.

Specifically this section shows statistics about:

- Origin of attacks and the principal locations of the victims.
- The types of most targeted data.
- Attack Methods.
- Most frequently exploited vulnerabilities by attack methods.

Figure 22 shows a summary of the **origin of attacks and the principal locations of the victims.** The top victim locations are United States of America (73%) and Australia (7%) and the top attacker locations are Romania (33.4%) and United States of America (29%). Obviously United States of America is the most interesting place to study.

Figure 22. Attacks origin and victims locations [Trustwave, 2013]

Table 13 shows **the types of most targeted data**. The primary data type targeted by attackers in 2012, as in 2011, was cardholder data. There is a well-established underground marketplace for stolen payment card data; cards data are bought and sold quickly for use in fraudulent transactions. With such a vast number of merchants accepting payment cards (estimates from major credit card brands put the total in the United States of America between nine and 10 million merchants), and with so many available attack vectors, it is unlikely this market will change any time soon.

Criminals also sought personally identifiable information (PII), which has some monetary value, albeit not as much as cardholder data, since it requires additional work and risk (i.e., posing as someone else) without the same lucrative return on investment.

The primary targets of cybercriminals in 2012 were Retail (45%), Food & Beverage (24%) and Hospitality (9%). There are several contributing factors to this continuing trend:

- The sheer volume of payment cards used in these industries makes them obvious targets.
- The main focus of organizations operating in these spaces is customer service, not data security.
- There's a misconception that these organizations are not a target. In practically all of the 2012 investigations, this statement was made in just about every case: "Why me?" The answer can only be: "Because, you have something worth taking that is not protected."

Table 13
Types of targeted data by attacks [Trustwave, 2013]

Types of targeted data by attacks	
Customer Records (Payment Card Data, PII, Email Addresses)	96%
Confidential Information & Intellectual Property	2%
Electronic Protected Health Information (ePHI)	1%
Business Financial Account Numbers	1%

With respect to **attack methods**, remote access remained the most widely used method of infiltration in 2012. Unfortunately for victim organizations, the front door is still open. Organizations that use third-party support typically use remote access applications like

Terminal Services or *Remote Desktop Protocol, pcAnywhere, Virtual Network Client (VNC), LogMeIn or Remote Administrator* to access their customers' systems. If these utilities are left enabled, attackers can access them as though they are legitimate system administrators.

How do attackers find remote access systems? Would-be attackers simply scan blocks of IP addresses looking for hosts that respond to queries on one of these ports. Once they have a focused target list of IP addresses with open remote administration ports, they can move on to the next part of the attack, one of the most exploited vulnerabilities: default/weak credentials. Unfortunately, gaining access to systems is just as easy as it is for attackers to identify targets.

Most current Web pages are not made up of static content as they were years ago, but of fluid dynamic components and content. In addition, many pages ask for information, location, preferences, with the goal of improved efficiency and user interaction. This dynamic content is usually transferred to and from back-end databases that contain volumes of information anything from cardholder data to which type of running shoes is most purchased. Pages will make Structured Query Language (SQL) queries to databases to send and receive information critical to making a positive user experience. Poor coding practices have allowed the SQL injection attack vector to remain on the threat landscape for more than 15 years. Any application that fails to properly handle user-supplied input is at risk. The good news is that properly using parameterized statements (aka prepared statements) will prevent SQL injection. When programmers fail to validate input (either by incorrectly validating or not validating at all), attackers can send arbitrary SQL commands to the database server. The most common attack goal with SQL injection is bulk extraction of data. Attackers can dump database tables with hundreds of thousands of customer records that contain personally identifiable information, cardholder data and anything else stored by the victim organization. In the wrong environment, SQL injection can also be exploited to

modify or delete data, execute arbitrary operating system commands or launch denial of service (DoS) attacks.

The third most widely seen method of entry in Trustwave's investigations was "Unknown." However, an overwhelming number of these cases possessed a common indicator of compromise, specifically weak and/or default credentials.

In the majority of cases Trustwave investigated in 2012, username and password combinations were woefully simple. Combinations included *administrator:password*, *guest:guest* and *admin:admin*. In addition, many IT service providers had standard passwords that were used by administrators allowing them to access any customer at any time. This means that if one location is compromised, every customer with that same *username:password* combination could also be compromised. Table 14 shows the most used attacks methods during year 2012.

Table 14
Attack entry methods [Trustwave, 2013]

Attack ENTRY methods	
Remote Access	47%
SQL Injection	26%
Unknown	18%
Remote code execution	3%
Client-side-attack	2%
Remote file inclusion	2%
Authorization flaw	1%
Physical theft	1%

With respect to **web applications**, table 15 shows the top 10 Web Hacking Incidents Database during the 2012 year. Denial of service is on the top of classification; Attackers constantly create tools to facilitate DoS attacks, such as WHID 2012-372, or WHID 2012-

368, [Whid, 2013] in which *GoDaddy* (Enterprise mail server) was stopped by a massive DoS attack. SQL injection vulnerability is one of the most used attack method for web applications. Another classification can be consulted in *WhiteHat* official web site [WhiteHat, 2012]

Table 15
TOP 10 WHID attack methods [Trustwave, 2013]

TOP 10 WHID Attack methods	
Unknown	46%
Denial of service	29%
Sql injection	11%
Stolen credentials	3%
Brute force	3%
Banking trojan	2%
Cross site scripting	1%
Predictable resource location	1%
DNS hijacking	1%
Cross site request forgery	1%

3.5. CONCLUSIONS.

The security design of an application must be accomplished from the beginning of SSDLC. All software applications types can have vulnerabilities in the code of all architecture tiers and design vulnerabilities in their architectural platforms and components. To perform a security assessment of an application is obliged knowing the main security vulnerabilities tendencies in the code, application design and also operational security vulnerabilities. This section of the dissertation has shown code vulnerability assessment statistics of known organizations about the main security vulnerabilities that applications had in recent years. The security vulnerabilities are the door for materializing attacks. It is important to know the latest trends in application attacks. Therefore, this section has also shown application

attacks statistics in recent years, including mention to special applications cases as AJAX, SCADA, WEB or MOBILE applications. The number of these applications is increasing continuously and it is important to study its vulnerability tendencies in the most recent years.

4. STATE OF THE ART IN APPLICATIONS SECURITY ANALYSIS

This chapter is a state of the art analysis of tools and artifacts involved in software security analysis of applications, such as knowledge sources, security analysis tools, benchmarks and methodologies for security analysis tools evaluation during the phases of Secure Software Development Life Cycle (SSDLC). A good state of the art is an excellent start point for:

- Understanding the different types of existing security analysis tools.
- Building an effective methodology to perform an assessment of the tools to allow making the best choice of them.
- Understanding how the tools can collaborate together to reach better results.

Doing this type of state of the art analysis involves an exhaustive knowledge of security standards, Secure Software Development Life Cycle process, security analysis tools and assessment methodologies or benchmarks:

- Organizations and standards of software security.
- Secure Software Development Life Cycle (SSDLC)
- Security Analysis Tools:
    - White box security analysis: Static Application Security Testing (SAST)
    - White box security analysis: Real time or interactive Application Security Testing (RAST/IAST) for Web applications
    - Black box security analysis: Dynamic Application Security Testing (DAST) for web applications
    - Hybrid of some SAST-RAST-DAST type tools for web applications
- Methodologies for tools evaluation.

- Benchmarks for tools evaluation.

4.1. ORGANIZATIONS AND STANDARDS OF SOFTWARE SECURITY.

To address the security design of a system or application it is necessary to obtain and acquire a good overview of how to achieve a security analysis of an application. To test an application online is complicated and needs to gather adequate knowledge. This knowledge can be gathered through various sources, one of them is to address and investigate in organizations, open projects and standards that have been occupied for some time collecting information about methodologies, protocols, cryptography, paradigms, reference projects and studies on the characteristics of all regarding security, test tools and tools and other forms of real-time protection, etc.

A Security Standard reference can provide information and knowledge about aspects such as:

- Security vulnerabilities, nature, characteristics, importance, statistics, etc.
- Application development methodologies, security software development life cycles (SSDLC).
- Methodologies for testing and application security testing.
- Security analysis tools, types, features, etc.
- Online protection tools, lists, features, etc.
- Assessment Methodologies for testing security analysis tools.
- Benchmarks for assessment of security analysis tools and vulnerabilities.

The following organizations must be considered among the most important ones concerned with the security of applications:

- **NIST. National Institute of Standards and Technologies. U.S.A. [Nist, 2013]**
- **OWASP. Open Web Application Security Project. [Owasp, 2013]**
- **WASC. Web Application Security Consortium. [Wasc, 2013]**
- **MITRE CWE. Common Weakness Enumeration. [Mitre, 2013]**
- **SANS Institute for security training. [Sans, 2013]**
- NSA. National Security Agency. U.S.A. [Nsa, 2013]
- OASIS. Open Control Standards for the Information Society. [Oasis, 2013]
- OISSG. Open Information Systems Security Groups. [Oissg, 2013]
- CGISECURITY. Web application security services [Cgisecurity, 2013]
- SEI CERT. Software Engineering Institute. [Sei, 2013]
- HOMELAND SECURITY. Build Security In. [Homeland, 2013]
- CISECURITY. Center for Internet Security. [Cisecurity, 2013]

Being the most complete, general and important ones, the standards, organizations and open projects highlighted in bold, OWASP, WASC, SANS, NIST and CWE MITRE have been the most consulted and referenced by this thesis.

The **OWASP project,** [Owasp, 2013] is an independent organization dedicated to finding and fighting the causes of insecure software. Organized in chapters and projects all over the world, it develops documentation, tools and open source standards (GPL, GFDL, and GPL). It is open to anyone. On its website, they mentioned:

"*Everyone is free to participate in OWASP and all of our materials are available under a free and open software license. You'll find everything about OWASP here on or linked from our wiki and current information on our OWASP Blog. OWASP does not endorse or recommend commercial products or services, Allowing our vendor neutral community To Remain With The collective wisdom of the best minds in security software worldwide. We*

ask That the community look out for Inappropriate use of the OWASP brand Including use of our name, logos, project names and other trademark issues ".

The OWASP project list is extensive [Owasp, 2103], covering all aspects of web application security, security tools, methodologies, good safety practices, benchmarking, etc.

The purpose of WASC [Wasc, 2013] is, as quoted on their website:
"The Web Application Security Consortium (WASC) is 501c3 nonprofit made up of an international group of experts, industry practitioners, and organizational Representatives who produce open source and Widely Agreed upon best-practice security standards for the World Wide Web. As an active community, WASC Facilitates the exchange of thoughts and Organizes several industry projects. WASC consistently releases technical information, Contributed articles, security guidelines, and other useful documentation. Businesses, Educational Institutions, Governments, application developers, security professionals, and software vendors all over the world Use our materials to assist with the challenges presented by web application security. Volunteering to Participate in WASC related activities is free and open to all. "

WASC has many interesting projects related to web application security that can and should be taken into account when implementing an application. Those are the consulting projects:

- Distributed Open Proxy Honeypots
- Script Mapping
- Static Analysis Tool Evaluation Criteria
- The Web Security Glossary
- Web Application Firewall Evaluation Criteria
- Web Application Security Scanner Evaluation Criteria
- Web Application Security Statistics - Web Hacking Incidents Database

- WASC Threat Classification

MITRE CWE [Mitre, 2013]. It addresses areas such as security analysis of the code, applications, systems evaluation, training or risk management. All safety related systems and applications. But mainly provides a dictionary of international public use that provides a unified measure of a set of software weaknesses that can lead to security vulnerabilities.

The difference between the CVE (Common Vulnerabilities and Exposures) definition and CWE is that the CWE (Common Weakness Enumeration) list includes software errors that can lead to software vulnerabilities. The list of CVE vulnerabilities are specific software errors detected in a given system (e.g. CVE 1999-0005 Arbitrary command execution via IMAP buffer overflow in authenticate command) that can be directly used by an attacker to gain access to a system.

Other definitions for weakness exist as bug, flaw, vulnerability, defect, error, etc. **In this work, therefore all these definitions refer to a weakness actually CWE**.

SANS [Sans, 2013]. The SANS Institute was established in 1989 as a cooperative research and education organization. Its programs now reach more than 165,000 security professionals around the world. The main concern in the face of this work is its ranking of the top 25 security errors or problems (weaknesses), which are clearly identified in the list of MITRE CWE [Mitre, 2013]. They can occur in web applications and among them are the most important problems of Web applications, such as XSS, SQLI, CSRF, REDIRECT OPEN, CROSS PATH, etc. In reference SANS access can find multiple resources on how to remove these categorization errors SANS TOP 25.

NIST [Nist, 2013]. NIST is the U.S.A. National Institute for Standards and technology of Many guidelines can be found in NIST mainly on security platforms, environments and applications and services that can be a good starting point to address the secure configuration of many parts of a system or application. NIST covers many fields of science

and one of them is information technology that addresses the following areas as mentioned in its official website:

- Biometrics
- Computer Forensics
- Computer Security
- Conformance Testing
- Cybersecurity
- Data Mining
- Data and Informatics
- Health IT
- Imaging
- Information Delivery Systems
- Networking
- Scientific Computing
- Software Testing Metrics
- Telecommunications / Wireless

4.2. SECURE SOFTWARE DEVELOPMENT LIFE CYCLE (SSDLC)

A **Secure Software Development Life Cycle** (SSDLC) for applications is the basis for developing secure applications. It uses different **human processes** and **technological** artifacts, like derivation methods for security requirements, risk analysis methods, security checklists, security analysis tools, security functional test, penetration test, etc. All artifacts working together in its SDLC corresponding phase have as objective obtaining an application with the least possible number of security vulnerabilities. The purpose of this

section is to collect and present overview information about existing processes, standards, life cycle models, frameworks, and methodologies that support or could support secure software development. Where applicable and possible, some evaluation or judgment are provided for particular life cycle models, processes, frameworks, and methodologies. According to technical note of Carnegie Mellon University CMU-SEI-2005-TN-024 [Davis, 2005] on Secure Software Development Life Cycle Processes, a number of existing processes, process models, and standards identify the following four SDLC focus areas for secure software development:

1. **Security Engineering Activities** include those activities needed to engineer a secure solution. Examples include security requirements elicitation and definition; secure design based on design principles for security, use of static analysis tools, secure reviews and inspections, and secures testing methods.
2. **Security Assurance Activities** include verification, validation, expert review, artifact review, and evaluations.
3. **Security Organizational and Project Management Activities** include organizational policies, senior management sponsorship and oversight, establishing organizational roles, and other organizational activities that support security. Project management activities include project planning and tracking, resource allocation and usage to ensure that the security engineering, security assurance, and risk identification activities are planned, managed, and tracked.
4. **Security Risk Identification and Management Activities** There is broad consensus in the community that identifying and managing security risks is one of the most important activities in a secure SDLC, and, in fact, is the driver for subsequent activities. Security risks in turn drive the other security engineering activities, the project management activities, and the security assurance activities.

Also existing **Capability Maturity Models** (CMM) provide a reference model of mature practices for a specified engineering discipline. An organization can compare their practices to the model to identify potential areas for improvement. The CMMs provide goal-level definitions for and key attributes of specific processes (software engineering, systems engineering, security engineering), but do not generally provide operational guidance for performing the work. This work analyzed several approximations of SSDLC and CMMs:

- Microsoft SDL [Microsoft-SDL, 2013]
- OWASP CLASP [Owasp-CLASP, 2013]
- SDLC Touchpoints [McGraw, 2006]
- Building Security in Maturity Models (BSIMM) [Bsimm, 2013]
- Open Software Assurance Maturity Model [OpenSAMM, 2013]
- CMMI [Cmmi, 2013]
- FAA-iCMM [FAA-iCMM, 2013]
- SEE-CMM [SEE-CMM, 2013]
- T-CMM/TSM [Kara, 2012]
- CC (Common Criteria) [Kara, 2012]

and this section describes briefly some of them that focus on the use of security automatic analysis tools to discover the security vulnerabilities that a building application have. Also this section will mention several comparisons references between SSDLC and/or CMM.

4.2.1. Microsoft SDL

The Security Development Lifecycle (SDL) [Microsoft-SDL, 2013] is a security assurance process focused on software development. As a company-wide initiative and a mandatory policy since 2004, the SDL has played a critical role in embedding security and privacy in software and culture at Microsoft. Combining a holistic and practical approach, the SDL

aims to reduce the number and severity of vulnerabilities in software. The SDL introduces security and privacy throughout all phases of the development process (figure 23).

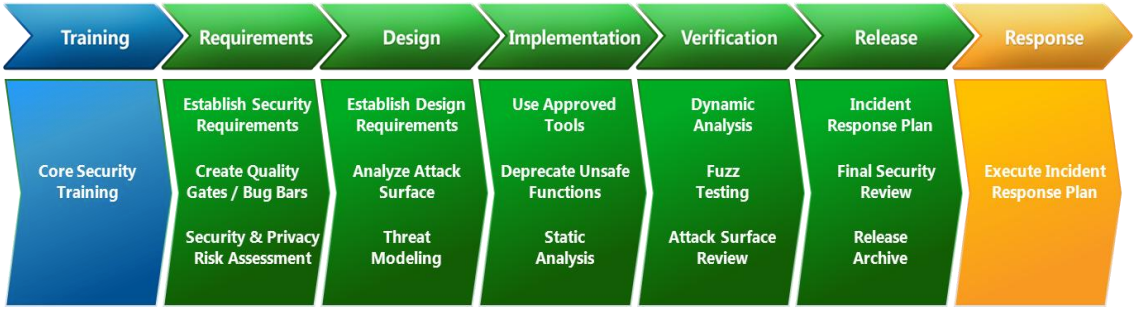

Figure 23. Microsoft SDL [Microsoft-SDL, 2013]

Regarding to the use of automatic security analysis tools in the implementation and verification phases, **SDL uses static analysis tools and black-box analysis tools** and white-box *appverifier* tool to examine the compatibility of the application with Windows OS platform. Therefore SDL do not use dynamic white-box security analysis tool in the verification phase. Neither SDL use Hybrid tools of static and dynamic tools.

4.2.2. OWASP CLASP

Comprehensive, Lightweight Application Security Process [Owasp-CLASP, 2103] is an activity-driven, role-based set of process components whose core contains formalized best practices for building security into an existing or new-start software development lifecycles in a structured, repeatable, and measurable way.

According to CLASP official site, "*is the outgrowth of years of extensive field work in which system resources of many development lifecycles were methodically decomposed in order to create a comprehensive set of security requirements. These resulting requirements form the basis of CLASP's best practices which allow organizations to systematically address vulnerabilities that, if exploited, can result in the failure of basic security services e.g., confidentiality, authentication, and access control*" [Owasp-CLASP, 2103].

This section provides an overview of CLASP's structure and of the dependencies between the CLASP process components and is organized as follows:

- CLASP Views
- CLASP Resources
- Vulnerability Use Cases

Figure 24 shows the CLASP Views and their interactions:

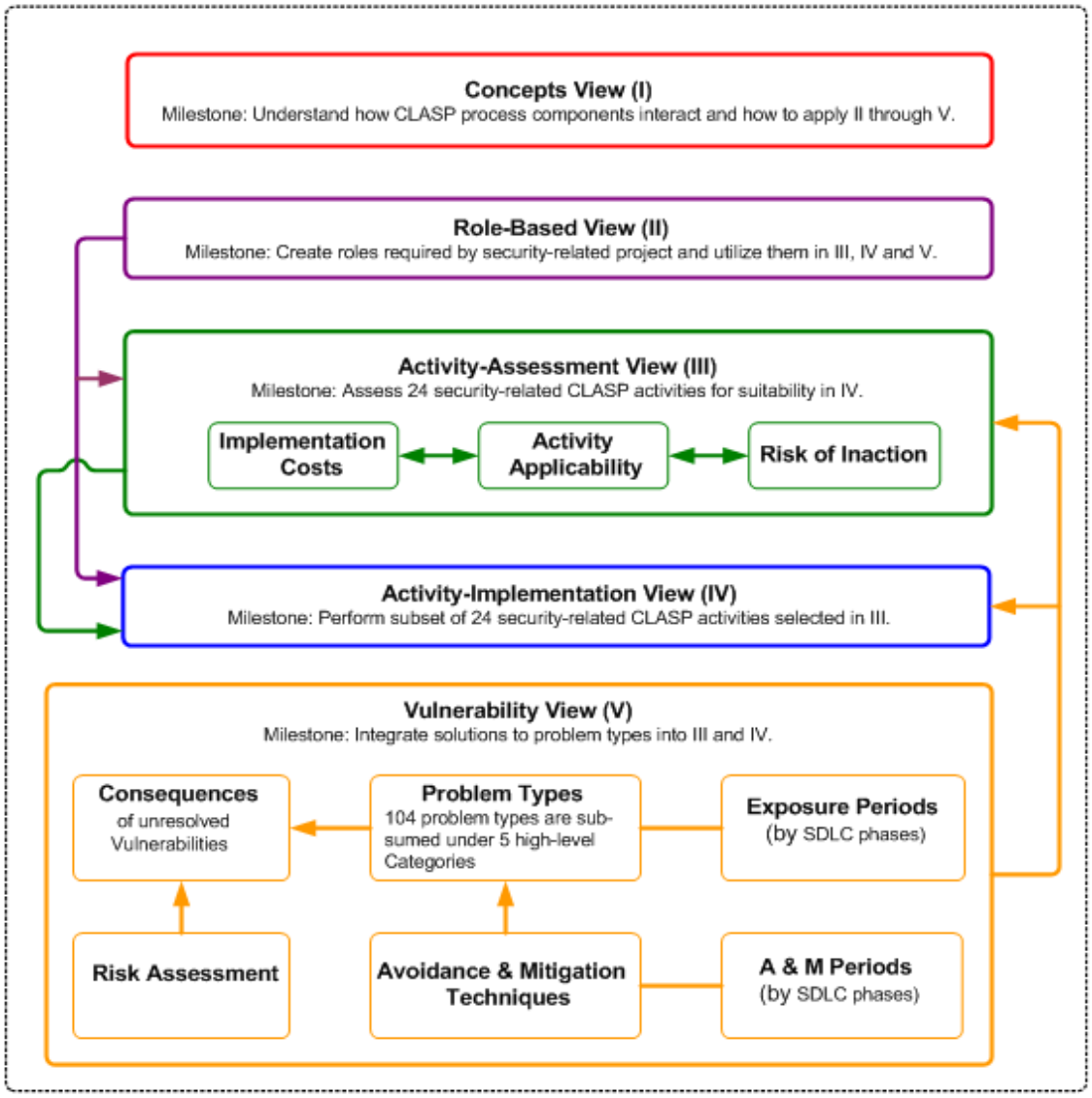

Figure 24. OWASP CLASP Views [Owasp-CLASP, 2013]

The CLASP process is presented through five high-level perspectives called CLASP Views. These views are broken down into activities which in turn contain process components.

This top-down organization by View > Activity > Process Component allows you to quickly understand the CLASP process, how CLASP pieces interact, and how to apply them to your specific software development lifecycle. These are the CLASP Views:

- Concepts View
- Role-Based View
- Activity-Assessment View
- Activity-Implementation View
- Vulnerability View

With respect to the use of automatic security analysis tools in the implementation and verification phases, **CLASP focuses more on white box testing** (Activities 7.1.2 and 7.1.3). CLASP also suggests the integration of security analysis into source management (Activity 6.1), in order to automate the implementation-level security analysis and metrics collection through the use of dynamic and/or static analysis tools.

4.2.3. SDLC Touchpoints

SDLC Touchpoints process [McGraw, 2006] provides a set of best practices that have been distilled over the years out of the extensive industrial experience of its proposer. Most of the best practices, named activities from here on, are grouped together in seven so-called touch points. The software security touchpoints are designed to be process agnostic. That is, the touchpoints can be applied no matter which software process used to build your software. As long as minimal set of software artifacts are being producing some, it can apply the touchpoints. Here are the touchpoints, in order of effectiveness (figure 25):

1. Code review
2. Architectural risk analysis
3. Penetration testing
4. Risk-based security tests

5. Abuse cases
6. Security requirements
7. Security operations

Regarding the use of automatic tools for testing applications Touchpoints emphasizes the importance of security testing, three out of seven touch points covered deal with security testing. A difference in focus does exist, as Touchpoints stresses the importance of risk based security testing (Activity 7.1.1). The main characteristic of Touchpoints is the **emphasis on code reviews**. In particular, the use of automated tools is suggested (and examples provided).

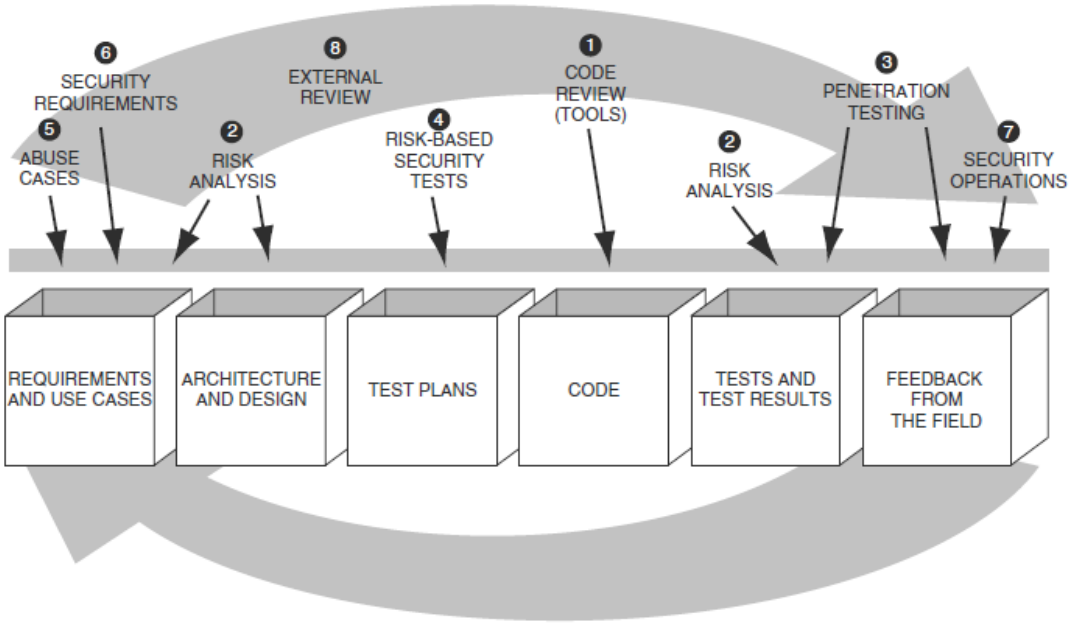

Figure 25. SDLC Touchpoints [McGraw, 2006]

4.2.4. SDL, CLASP, SDLC Touchpoints TESTING COMPARISON

Examining the paper "*On the Secure Software Development Process: CLASP, SDL and Touchpoints Compared"* [De win, 2009] not surprisingly, all analyzed processes stress the importance of security testing. A closer look to the documentation reveals that they all provide thorough, good-quality guidance in describing the testing-related activities. However, different flavors can be identified. SDL has a predominant black-hat approach to

testing, i.e., activities focus on fuzz testing and penetration testing. CLASP is mostly white-hat, as illustrated by activities like resource driven testing, testing of security attributes (e.g., privileges), integration and automation of security testing with the commit procedure of the software repository and with the build process. Touchpoints is in between: some activities are mainly white-hat oriented, e.g., security functionality testing, while the black-hat component is still very present, for instance, pentesting is a touch point in its own.

Concerning the stakeholders that are involved in this phase, the implementers and the testers are center stage in all process. However, we observe that SDL goes a bit further. Significant attention is devoted to provide the project managers with test results in order to track the project status (security-wise). As a final consideration, the use of both formal notations and the code generation techniques (e.g., MDA) do not find the proper space in any of the processes under study.

4.2.5. ADDITIONAL SSDLC,s and CMM,s SURVEY

The technical note of Carnegie Mellon University CMU-SEI-2005-TN-024 [Davis, 2005] on Secure Software Development Life Cycle Processes provides a briefly survey of following Capabilities Maturities models:

- Capability Maturity Model Integration (CMMI)
- Federal Aviation Administration integrated Capability Maturity Model (FAA-iCMM)
- Trusted CMM/Trusted Software Methodology (T-CMM/TSM)
- Systems Security Engineering Capability Maturity Model (SSE-CMM)

Another more recent work of Mehmet Kara [Kara, 2012] has the goal of evaluating and comparing Microsoft SDL, SSE-CMMI, OpenSAMM and CC secure software development approaches. CC is hardware/software security evaluation standard which is used for security testing, security requirements definition another secure system issues. In this paper

Common Criteria is used as secure software development guidance and compared with Microsoft SDL, SSE-CMMI, and OpenSAMM. Table 16 shows a brief comparison for security development processes.

Table 16
SSDLC,s and CMM,s comparison [Kara, 2012]

	Common Criteria	SSE-CMM	Microsoft-SDL	Open-SAMM
Security Training and Awareness	x	ok	ok	ok
Physical and Logical Security	ok	ok	x	x
Secure Configuration Management	ok	ok	x	x
Law, policy and procedure compliance	x	ok	x	ok
Threat Modeling	ok	ok	ok	ok
Risk Analysis	ok	ok	ok	ok
Security Requirements Definition	ok	ok	ok	ok
Security Architecture	ok	ok	ok	ok
Secure Design	ok	ok	ok	ok
Source Code Analysis	x	x	ok	ok
Vulnerability Analysis	x	ok	ok	ok
Security Verification	ok	ok	ok	ok
Vulnerability Management	ok	ok	ok	ok
Secure Development Techniques and Applications	x	ok	ok	ok
Operational Environment Security	ok	ok	ok	ok
Secure Integration with Peripheral	ok	ok	ok	ok
Secure Delivery	ok	ok	x	ok

4.3. WHITE BOX SECURITY ANALYSIS: STATIC ANALYSIS TOOLS

Static Applications Security analysis Tools (SAST) analyze both source code and executable, as appropriate according to its availability, for finding security vulnerabilities in

the code of applications. The no availability of source code for software of third parties make these tools can perform a white box analysis of entire application. Basically the difference between a tool for source code and for executable code is that the last one must first make a disassembly of the executable code to extract the source code and then act as the other source static tools, they have a pre-conversion executable code in source code. This section about SAST tools resumes:

- Characteristics.
- Categories of static tools.
- Survey of availability of Static tools

4.3.1. SAST TOOLS CHARACTERISTICS

This paragraph shows a brief description of concepts, architecture, using, problems, categories and availability of commercial and open source static tools.

Architecture. SAST tools take the source or executable code as input and transform it in an intermediate representation or model of source code, as appropriate, and then analyze it against a set of rules defined in the tools, to generate the corresponding security vulnerability report.

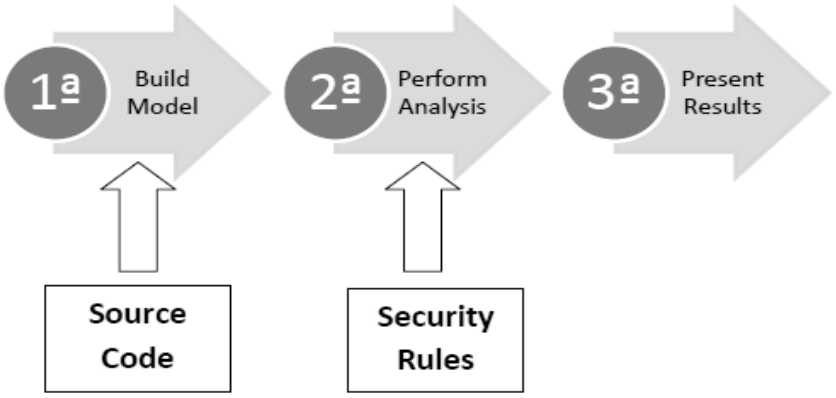

Figure 26. Static analysis tools process [Díaz, 2013]

As Figure 26 shows, according to Chess and West [Chess, 2007], the code is transformed using a combination of different techniques as lexical and semantic analysis, abstract syntax and parsing. The program model is analyzed using intraprocedural (local) analysis for examination of individual functions and interprocedural (global) analysis for the interaction between functions, using tracking control flow and data flow, pointer aliasing, etc. Depending on the selected tool these rules are fixed or can be extended. The set of rules can be increased in many cases by the user, who can define your own, to suit the particular application being analyzed. An example of rule specification language PQL (Program Query Language) for defining security flaws for java language can be found in [Livshits, 2005].

The majority of these tools work either by applying SAT solvers [Moskewicz, 2001], abstract interpretation [Cousot, 1996], by performing model checking [Aoki, 2010], or by performing Taint Analysis [Tripp, 2013], [Tripp, 2009] for local analysis and using function summaries along with SAT solvers, model checking or theorem provers [Detlefs. 2005] algorithms for global analysis. The global analysis can be:

- Context-sensitive. Determining the context of a function when called.
- Path sensitive. Explore the routes based on information flow control.
- Path insensitive. Explore all the routes. Very expensive.
- Based on Function summaries. Using the call context summaries of functions, more flexible than the previous one, can be more or less inaccurate.

SAST tools for executable code. According to the Veracode report volume 5 [Veracode, 2012], where initial security checks of third-party software are accomplished, the result, shown in Figure 7 (section 2.1), is that the security of third-party software is not acceptable by 75% .The same report showed the interesting observation, that almost a third of the development volume comes from third-party development, commercial and open source.

These approaches analyze machine code directly from a simplification of the same for constructing a flow control diagram and calls. In [Hanov 2005], addresses various problems that can present the machine code decompilation and describes the evolution of the various techniques used. Cifuentes [Cifuentes, 1997] developed a technique known as “program slicing”, for determining the set of statements of a program that potentially affect the value of a variable at some point in the program. This analysis is useful in the decoding phase machine code instructions from reverse engineering tools such as binary translators, disassemblers and debuggers. Relative to Java language, a technical analysis of java bytecode [O'Donoghue, 2002] can be consulted. Regarding to web applications, there are several commercial implementations for various languages and on-demand software service platforms SaaS (software as a service) companies as Veracode [Veracode, 2013]. Its services are available for J2EE, C / C + +. NET, C #, PHP, ColdFusion.

False Negatives & False Positives. Static tools exhibits a classical problem that the act of determining if a program reaches its final state, or not, is an undecidable problem, the halting problem [Turing, 1936]. In this context, **a false positive, is a problem discovered in a program when there is actually no problem**. The notion of using an algorithm to analyze other is part of the origins of computing. For further reading on the computational theory, Sipser's Introduction to the Theory of Computation, Second Edition [Sipser, 2005] is recommended. Therefore, SAST tools can suffer from false negatives and false positives. **A false negative is defined as a security vulnerability in the code which is not detected by the tool**. A false positive is a reported security vulnerability in an application that is not really a vulnerability. The best tool is that is capable of making the best balance between false negatives and false positives due to its algorithms efficiency. Usually a tool that has more false positives, it has a less number of false negatives depending obviously of the efficiency of its algorithms. The objective of having more or less false positives really determines distinct tools categories. Companies interested on acquiring a SAST tool should

take into account the time that security teams have to perform the audit of a tool that usually reports many false positives. But they must also take into account that maybe it is more important having the less possible number of false negatives. Conclusion: The penalty associated with a false negative is much greater because the vulnerability will remain in the code but also the task of revising a report with many false positives can be very arduous. Other discussion about the balance of false negatives and false positives can be found in [Chess, 2007] and [Díaz, 2013]. Also in *a Systematic Mapping Study of Static tools* [Lobo, 2013] the authors try identifying current state-of-the-art static analysis techniques and tools as well as the main approaches that have been developed to mitigate false positives. Among the retrieved studies, there was a lack of works on the types of false positive errors and the tools that generate them. This kind of research would help developers identify the tools that best serve their needs. The mapping also revealed studies that use hybrid approaches, which combine static and dynamic analyses techniques. Furthermore, a combination of different static analysis approaches proved more efficient than their isolated use.

One of the most important advantages of SAST tools is that they analyze the entire application and attack surface of the application covering all inputs. They are considered the most important safety activity within a SSDLC by [McGraw, 2006] and the same also appears from the statistics discussed above on WASC vulnerabilities [WASC-statistics, 2008]. The results in the security report, Veracode volume 3 [Veracode, 2011] are also relevant, showing the analysis accomplished with DAST and SAST techniques: 635 SAST detections vs. 29 DAST. SAST provides an extensive knowledge of application and DAST only test accessible parts from the outside. Static analysis tools check all code thoroughly and consistently, without any tendency Sometimes programmers could put more attention on revising some parts of the code that might be more "interesting" from a security perspective or parts of the code that may be easier to perform the dynamic tests. A valuable analysis should be as unbiased as possible. Examining the code completely and thoroughly

is a good feedback on the application knowledge digging deeper in the knowledge of the security.

Examining the code itself, static analysis tools can indicate the root cause of a security issue, not just one of its symptoms. This is particularly important to ensure that vulnerabilities are corrected properly. Static analysis can find errors early in the implementation phase of development, even before the program is executed for the first time. The early finding of an error does not only reduce the cost to fix the error, but also produces a rapid feedback cycle that can help directing the work of a programmer: a programmer has the opportunity to correct errors he was not aware of before. Attack scenarios and code information used by static analysis tools act as a means of knowledge transfer.

Audit. The existence of false positives and false negatives forces a subsequent audit of the tool reports, needed to eliminate the false positives and find the false negatives (much more complicated). This implies adequate training in the defects that can occur in the code for a particular programming language, which can be more or less "friendly" in terms of the error trace facilities a specific tool provide. Tools such as SCA, PREVENT or INSIGHT [Bermejo, 2009] are good examples of tools that provide a very good information for, above all, eliminate false positives.

There are available commercial and open source SAST tools for:

- Source and executable code.
- All types of applications: non-web, web (traditional web applications, web services or rich internet applications as Ajax) or mobile applications for blackberry, android and iphone platforms.
- Majority of languages and development technologies as J2EE or .NET.

4.3.2. SAST TOOLS CATEGORIES

Different schemes exist to categorize static security analysis tools according to [McGraw, 2006]. The classification can be made attending to:

- Languages and development technologies (web services, javascript, ajax, etc.)
- Kind of applications analyzed (non-web, web, and mobile).
- Algorithms and techniques of security vulnerabilities searching used as Model Checking [Aoki, 2010], Boolean Satisfiability [Moskewicz, 2001] or Taint Analysis ANDROMEDA tool [Tripp, 2013] and TAJ tool [Tripp, 2009].
- Different limits of false positives rates and false negatives as objective in the application analysis.

A relevant taxonomy for the purpose of this work is shown in [Chess, 2007]. This taxonomy refers to the general purpose of the tools and differentiates between:

- Style checking
- Program understanding
- Program verification and property checking
- Bug finding
- Security review

Style checking. These earlier tools as lint tool [Johnson, 1977] in Unix or PMD [Pmd, 2013] for java language usually enforce a more selective and more superficial set of rules than a type checker. They perform checks based on lexical and syntactic analysis (earlier generation of static analysis) to discover problems as inconsistencies in function calls, return values in some places and not in others, functions called with varying numbers of arguments, function calls that pass arguments of other types or detecting the use of certain dangerous functions. This analysis has limitations, when compared with other types mentioned afterwards, not making an analysis based on simulating what happens in

runtime. In comparison with any compiler, these tools perform only an additional lexical and syntactic check of the using of certain dangerous functions in the code.

Program understanding. These tools are designed for helping users to make sense of a project code. They are included in many Integrated Development Environments and are designed to help programmers to gain insight into the way a program works. They help the reviewer who performs security analysis to understand the code and discover vulnerabilities but, in any case, this is a manual review of all code and it is time consuming. An example is the Fujaba Tool [Fujaba, 2013].

Program verification and property checking. These tools accept a specification and a body of code and then attempt to prove that the code is a correct implementation of the specification. If the specification is a complete description of everything the program should do, the program verification tool can perform equivalence checking to make sure that the code and the specification exactly match. More commonly, verification tools check software against a *partial specification* that details only part of the behavior of a program. This endeavor sometimes goes by the name *property checking.* Many property checking tools focus on *temporal safety properties.* A temporal safety property specifies an ordered sequence of events that a program must *not* carry out. An example of a temporal safety property is "a memory location should not be read after it is freed." Most property checking tools enable programmers to write their own specifications to check program-specific properties. When a property checking tool discovers that the code might not match the specification, it traditionally explains its finding to the user by reporting a counterexample: a hypothetical event or sequence of events that takes place within the program that will lead to the property being violated. Some examples are CBMC [Clark. 2004], Polyspace [Mathworks, 2013], Codesonar [Grammatech, 2013], or Satabs [Clark, 2005].

Bug finding. These tools simply warn about places in the code where the program is going to act in a different way, not necessarily insecure, to the one desired by the programmer. These tools contain a predefined set of rules describing patterns in code that can indicate security vulnerabilities. This set can be extended in many tools by the user to adapt to the nature of a particular code.

Most "bug finding" tools are also designed to produce a low number of false positives for analyzing applications with a high number of lines of code, in the order of hundreds of thousands or millions of lines of code.

Vulnerabilities tracks are provided after the execution of the tools, showing a possible sequence of events in the code, once a suspicious vulnerability is identified allowing the possibility of checking the veracity of the discover vulnerability.

A number of these tools is available as, for example, FindBugs [Findbugs, 2103], a general tool for identifying vulnerabilities in Java code or Prevent, a commercial tool from Coverity [Coverity, 2013], available for C/C++, Java, J2EE and C#. Prefast [Prefast, 2013], which is able to check common coding error in C and C++ languages. Finally Klocwork offers Insight [Klocwork, 2013]; available also for C/C++, Java, J2EE and C #, a product suite that allows graphical exploration of programs with hundreds of thousands or millions of lines of code.

Security review. These tools combine many of the techniques of the previous tools with the goal of identifying security vulnerabilities applying these techniques differently. Their design implements in fact a combination of property checkers and "bug finding" class of tools techniques, because many security properties can be expressed briefly as program properties. The designers of these tools prefer the cautious side of the balance between false positives and negatives, the better a security tool is, the better job it will do at minimizing "dumb" false positives without allowing false negatives to creep in. These tools prefer to

show many points in the code that should be manually reviewed after the execution of the tool, producing more false positives when compared with “bug finding” tools.

Two of the most relevant “security review” tools are IBM Appscan source, [Appscan, 2013] and SCA (Source Code Analyzer) from HP Fortify Software [Hp-Fortify, 2013]. SCA is able to analyze code of different languages, as C/C++, C#, ASP NET, VB.NET, COBOL, CFML, HTML, Java, JavaScript, AJAX, JSP, PHP, PL/SQL, Python, Visual Basic, VBScript and XML.

4.3.3. SAST TOOLS AVAILABILITY SURVEY

This section is a review about available commercial and open source SAST tools showing their most important characteristics and skills. Chapter 5 will show several assessments processes for non-web and web applications SAST tools with objective results to allow selecting the tools according to the best performance, usability and the number and range of covered vulnerabilities for relevant commercial and noncommercial SAST tools.

There are available commercial and open source tools. As we will demonstrate (see chapter 5) commercial ones have support for more languages, have a larger vulnerabilities coverage, better usability and trace help for discarding false positives. They are also the best candidates for being included in a process of code security review in a company.

Open source tools offer their complete code and documentation. This is an advantage because we can understand better their limitations. Generally almost all open source tools are research projects from Universities, or in some cases from companies, and usually their vulnerabilities coverage is. Also their usability, human interfaces and warning trace capabilities are much more reduced than commercial tools. Some tools as MAGIC [Chaki, 2004], Blast [Beyer. 2007], Splint [Evans, 2002] or CQUAL [Foster, 1999], can require code annotations to enhance the results, making them not useful for analysis of projects

with several hundreds of thousands or millions of lines of code. Other tools as ITS4 [Viega, 2000], UNO [Holzmann, 2002], Lint [Jhonson, 1977], RATS [Nazario, 2002] or Flawfinder [Nazario, 2002] belong to an earlier generation of tools based only in lexical and syntactic analysis. Table 17 resumes the reviewed open source tools and the considerations observed. SAST tools for other languages, platforms and technologies:

- Non-Web applications
    - C/C++: MAGIC, Blast, Splint, CQUAL, SATURN [Aiken, 2006], BOOP [Boop, 2013], ITS4, UNO, Lint, RATS, Clang, Smatch, CppCheck [CppCheck, 2013] or Flawfinder
    - Java: PMD [Pmd, 2013], ESC [Esc, 2013], Java PathFinder [Pathfinder, 2103]
- Web applications,
    - PHP: Rips [Rips, 2013]
    - J2EE; LAPSE+ [Lapse+, 2013], Findbugs [Findbgs, 2103]
    - .NET: FxCop [FxCop, 2013] CAT.NET [Cat, 2013]
    - C/C++, C#, VB, Java and PL/SQL: VisualCodeGrepper [Grepper, 2013]
    - Multilanguage: Yasca [Yasca, 2013]

Table 17
Open source SAST tools

TOOLS	CHARACTERISTICS AND CONSIDERATIONS
UNO, RATS, FLAWFINDER, ITS4, LINT	C/C++. Earlier tools limited to lexical-syntactic analysis and only for a reduced subset of vulnerabilities. All of them preprocess and tokenize source files (the same first steps a compiler take) and then match the resulting token stream against a library of vulnerable constructs.
BOON	C/C++. Applies integer range analysis. It can't model interprocedural dependencies, and it ignores pointer aliasing
CQUAL	C/C++. Type-based analysis, requires annotations in the code
BLAST	C/C++. Model checking tool, with the option of adding assertions in the

	code
SPLINT	C/C++. Enhanced version of Lint. Requires annotations in the code
SATURN	C/C++. Boolean satisfiability and summary based tool. Only limited to memory leaks, lock problems and null dereferences vulnerabilities.
BOOP	C/C++. Abstraction and model checking tool. Not maintained anymore. The formalization of C expressions is incomplete and not all C constructs are covered.
SATABS	C/C++. Program verification tool with Model checking, that implements a predicate abstraction refinement loop using a SAT-solver. This allows the model checker to handle the semantics of the ANSI-C standard accurately.
CBMC	C/C++. Program verification tool with Bounded Model Checking new tool research. In CBMC, the transition relation for a complex state machine and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using a SAT procedure
MAGIC	C/C++. Bounded Model Checking tool that require specifications in the code to accomplish an analysis
SMATCH	C/C++. Simple scripts look for problems in simplified representation of code. primarily for Linux kernel code
CLANG	C/C++, objective C. Resports dead stores, memory leaks, null pointer deref, and more. Uses source annotations like "nonnull".
CPPCHECK	C/C++. pointer to a variable that goes out of scope, bounds, classes (missing constructors, unused private functions, etc.), exception safety, memory leaks, invalid STL usage, overlapping data in sprintf, division by zero, null pointer dereference, unused struct member, passing parameter by value, etc. Aims for no false positives.
PMD	Java. Questionable constructs, dead code, duplicate code.
JAVA PATHFINDER	Java. Its primary application has been Model checking of concurrent programs, to find defects such as data races and deadlocks
ESC	Java. Provides programmers with a simple annotation language with which programmer design decisions can be expressed formally. Check code for nulls, race conditions, non init vars, exceptions and other.
JLINT	Java. inconsistencies, and synchronization problems
FINDBUGS	WEB APPLICATIONS. J2EE. ANALYZES BYTECODE. Null pointer deferences, synchronization errors, SQLI, XSS, etc.

FINDSECURITYBUGS	WEB APPLICATIONS. J2EE. ANALYZES BYTECODE. Extends FindBugs with more security detectors (Command Injection, XPath Injection, SQL/HQL Injection, Cryptography weakness and more).
LAPSE+	WEB APPLICATIONS. J2EE. is a security scanner for detecting vulnerabilities of untrusted data injection in Java EE Applications. It has been developed as a plugin for Eclipse
FXCOP	WEB APPLICATIONS. .NET. ANALYZES BYTECODE. Microsoft free tool. FxCop analyzes the compiled object code
RIPS	WEB APPLICATIONS. PHP.
YASCA	WEB APPLICATIONS. Java, C/C++, JavaScript, ASP, ColdFusion, PHP, COBOL, .NET. aggregator of other tools, including: FindBugs, PMD, JLint, JavaScript Lint, PHPLint, CppCheck, ClamAV, RATS, and Pixy.
VISUALCODEGREPPER	WEB APPLICATIONS. C/C++, C#, VB, Java and PL/ SQL.
CAT.NET	WEB APPLICATIONS. .NET (MICROSOFT) ANALYZES BINARY CODE

Commercial tools have a much more limited support for researchers. For example, all of them give a list of the vulnerabilities they claim to detect, but this list doesn't follow a specific standard. Each tool provides its own list with its own format, although all of them present also its vulnerabilities list in CWE format. Also all of them provide a limited set of documentation, explaining the internal design (algorithms, heuristics, etc.) they use, but without details and, of course, without access to the code.

Table 18 resumes the properties of commercial SAST tools relative to languages, vulnerabilities coverage and availability for web, non-web or mobile applications.

- WEB/NON-WEB/MOBILE applications.
    - SCA(HP-FORTIFY) [HP-Fortify, 2013]
    - SECURITY APPSCAN SOURCE (IBM) [Appscan, 2013]
    - INSIGHT (KLOCWORK) [Klocwork, 2013]

- VERACODE SaaS (VERACODE) [Veracode, 2013]
- CXSUITE (CHECKMARX) [Checkmarx, 2013]

- WEB APPLICATIONS
  - CODESECURE (ARMORIZE) [Armorize, 2013]
  - BUGSCOUT (BUGUROO) [Buguroo, 2013]
- NON-WEB APPLICATIONS
  - SAVE (COVERITY) [Coverity, 2013]
  - GOANNA (RED LIZARD) [RedLizard, 2013]
  - PC-LINT (GIMPEL) [Gimpel, 2013]
  - CODESONAR (GRAMMATECH) [Grammatech, 2013]
  - POLYSPACE (MATHWORKS) [Mathworks, 2013]
  - .TEST / jTEST / DOTTEST [Parasoft, 2013]

Table 18
Commercial SAST tools

TOOLS	CHARACTERISTICS AND CONSIDERATIONS
SCA (HP FORTIFY)	WEB/NON-WEB/MOBILE APPLICATIONS. Leader security review tool. 100% coverage of table 1 vulnerabilities categories. Covers 18 different languages.
SECURITY APPSCAN SOURCE (IBM)	WEB/NON-WEB/MOBILE APPLICATIONS. Leader security review tool. Large coverage of languages and vulnerabilities categories.
K8-INSIGHT (KLOCWORK)	WEB/NON-WEB/MOBILE APPLICATIONS. Java, J2EE, C and C#. Bug finding tool.
VERACODE SaaS (VERACODE)	WEB/NON-WEB/MOBILE APPLICATIONS. C, C++, .NET (C#, C++/CLI, VB.NET, ASP.NET), Java, JSP, ColdFusion, PHP, Ruby on Rails, and Objective-C, including mobile applications on the Windows Mobile, BlackBerry, Android, and iOS platforms.
CHECKMARX CX-ENTERPRISE (CHECKMARX)	WEB/NON-WEB/MOBILE APPLICATIONS Bug finding tool. It covers 15 different languages.
SAVE (COVERITY)	NON-WEB APPLICATIONS. Java, C and C#. Bug finding tool.
GOANNA (RED LIZARD)	NON-WEB APPLICATIONS. Bug finding tool for C and C++. Without injection vulnerabilities coverage

PC-LINT (GIMPEL)	NON-WEB APPLICATIONS. Tool for C and C++. Without injection vulnerabilities coverage.
CODESONAR (GRAMMATECH)	NON-WEB APPLICATIONS. Program verification tool for C/C++ and Java, It does not check for the most severe vulnerabilities, such as SQL injection and cross-site scripting.
POLYSPACE (MATHWORKS)	NON-WEB APPLICATIONS. Program verification tool for ADA, C/ C++. Proves the absence of overflow, divide-by-zero, out-of-bounds array access, and run-time errors. It was not possible getting it for evaluation, no response received.
.TEST / jTEST / DOTTEST (PARASOFT)	NON-WEB APPLICATIONS. Security and Quality analysis tool C, C++, Java, C#, and .NET. It focuses more in quality than security.
CODEPEER (ADACORE)	NON-WEB APPLICATIONS. ADA. Detects uninitialized data, pointer misuse, buffer overflow, numeric overflow, division by zero, dead code, concurrency faults (race conditions), unused variables, etc.
CODESECURE (ARMORIZE)	WEB APPLICATIONS. ASP.NET, VB.NET, C#, Java/J2EE, JSP, EJB, PHP, Classic ASP and VBScript. Powerfull tool for web applications with wide coverage of vulnerabilities.
BUGSCOUT (BUGUROO)	WEB APPLICATIONS. Java, PHP, ASP and C#

Several other SAST tools lists for commercial and non-commercial tools and diversity of languages and platforms can be consulted from diverse sites as:

- WASC [SAST-wasc, 2013]
- SAMATE NIST [SAST-samate, 2103]
- WIKIPEDIA [SAST-wiki, 2013]
- OWASP [SAST-owasp, 2013]

4.4. BLACK BOX SECURITY ANALYSIS: DYNAMIC ANALYSIS TOOLS

Dynamic Application Security Testing (DAST) tools act as black box type tools, executing the tool against the running application performing a penetration test and trying to cover the

entire surface of attack (all possible inputs to the application) to find vulnerabilities that may exist. Several examples of this type of tools for web applications are WebInspect [HP-Fortify, 2013], PAROS [Paros, 2013] or CENCIZ [Cenciz, 2013]. Such tools relative to web applications are more commonly called Web Application Automatic Vulnerabilities Scanners. Bau et al. [Bau, 2010] assess the current state of the art analyzing eight leading tools and carried out a study of: the class of vulnerabilities tested by these scanners, their effectiveness against target vulnerabilities, and the relevance of the target vulnerabilities to vulnerabilities found in the wild.

This section about DAST tools resumes:

- Characteristics and architecture.
- Survey of availability of DAST tools

4.4.1. DAST TOOLS CHARACTERISTICS

According to Elizabeth Fong, in two interesting articles about DAST tools [Fong, 2007] [Fong, 2008], they should:

- Be able to identify a subset of acceptable security vulnerabilities of web applications.
- Generate a report for each vulnerability detected, indicating an action or set of actions that suggest the aforementioned vulnerability.
- Have an acceptable false positive rate, which of course can also have these tools

An automated vulnerability scanner acts as shown in Figure 27: the scanner is between the administrator and the web application tool to launch attacks against the application, performing a penetration test, injecting malicious data and code to detect vulnerabilities.

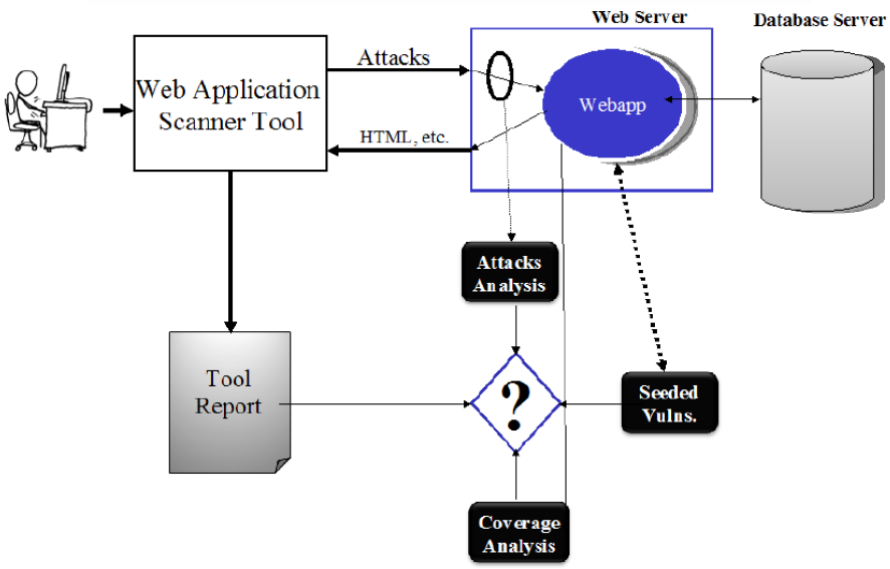

Figura 27. Web application vulnerability scanner schema [Samate, 2013]

Automatic scanners have the following **advantages**:

- They can detect vulnerabilities in the final version of the product in the phase prior to its distribution.
- They simulate the behavior of a malicious user, carrying out attacks and analyzing test results in a very near as would a real attacker.
- They are independent of the coding language. A web scanner itself can deal with applications coded in different languages.

However, DAST tools also have a number of **weaknesses and limitations** that must be taken into account:

- Being based on trial and error techniques, they cannot cover 100% of web application code, and parts of it may be untested.
- There are problems that cannot be found, especially related to the logic of the application, as they are oblivious to it. For example, scanners cannot determine if

somewhere there is information that should not be, or if an user really has permission to view the received item.

- To cover all possible attack vectors of a vulnerability is difficult. The scanners have certain predefined attack patterns, but they will never be as imaginative as a real attacker.
- The knowledge we have about the behavior of the different dynamic elements like JavaScript or Flash, is limited. Therefore scanners cannot determine whether its behavior is correct.
- Black box scanners should be extended to handle AJAX requests. In fact, any interaction with the web application always contains a request and response, however the content of the response is no longer an HTML page. Thus, DAST tools could extend our notion of a "page" to typical response content, end of AJAX calls, such as JSON or XML. A way to handle AJAX would be to follow a Crawljax approach [Doupé, 2012] and convert the dynamic AJAX calls into static pages.

Vulnerabilities coverage. Covering the entire surface application of attack is difficult; the degree to which this is achieved also determines the effectiveness of the tool. It's hard, because the person that performs a manual penetration test or the automatic scanner tool must try all entrances to the application and all user roles, each parameter of each request and each response pattern to find a vulnerability. The possibilities and weaknesses of a scanner must be well understood to make the best possible interpreting their results [SAMATE, 2013]. The automatic scanners have limitations and can detect only a set of vulnerabilities due to their nature. For example, a scanner can detect vulnerabilities as [Stuttard, 2008]:

- **Reflected cross-site scripting** vulnerabilities arise when user-supplied input is echoed back in the application's responses without appropriate sanitization.

Automated scanners typically send test strings containing HTML markup, and search the responses for these strings, enabling them to detect many of these flaws.

- Some **SQL injection** vulnerabilities can be detected via a signature. For example, submitting a single quotation mark may result in an ODBC error message, or submitting the string '; waitfor delay '0:0:30'-- may result in a time delay.
- Some **path traversal** vulnerabilities can be detected by submitting a traversal sequence targeting a known file such as boot.ini or /etc/passwd and searching the response for the appearance of this file.
- Some **command injection** vulnerabilities can be detected by injecting a command that will cause a time delay, or will echo a specific string into the application's response, and others as file inclusion, xpath injection or http response spliting.
- Straightforward **directory listings** can be identified by requesting the directory path and looking for a response containing text that looks like a directory listing.
- Vulnerabilities like **frame injection**, liberally scoped cookies, and forms with autocomplete enabled can be reliably detected by reviewing the contents of client-side code.
- Items not linked from the main published content, such as backup files and source files, can often be discovered by requesting each enumerated resource with a different file extension.

Because of these scanners perform syntactic parsing of the web application, they cannot understand the semantics of various parameters as a whole, that can hide an attempted attack. Therefore it is difficult the detection of other vulnerabilities as:

- **Broken access controls**, which enable a user to access other users' data, or a low-privileged user to access administrative functionality. A scanner does not understand the access control requirements relevant to the application, nor is it able

to assess the significance of the different functions and data that it discovers using any particular user account.

- Attacks involving the modification of a parameter's value in a way that has meaning within the application, for example, a hidden field representing the price of a purchased item, or the status of an order. A scanner does not understand the meaning that any parameter has within the application's functionality.
- Other logic flaws, such as beating a transaction limit using a negative value, or bypassing a stage of an account recovery process by omitting a key request parameter.
- **Vulnerabilities in the design** of application functionality, such as weak password quality rules, the ability to enumerate usernames from login failure messages, and easily guessable forgotten password hints.
- **Session hijacking** attacks in which a sequence can be detected in the application's session tokens, enabling an attacker to masquerade as other users. Even if a scanner can recognize that a particular parameter has a predictable value across successive logins, it will not understand the significance of the different content that results from modifying that parameter.
- **Leakage of sensitive information** such as listings of usernames, and logs containing session tokens.

False positives. As with SAST tools, it is very important to check the false positives produced by these tools. As described below, a good tactic can be to correlate the results of static analysis and automated scanners of web applications to assist in discarding false positives. Another approach is to use static analysis results to generate test cases for automatic scanners, improving the accuracy of the existence of the vulnerability reported by static analysis.

In the WASC website [Wasc, 2103], the Web application scanners project evaluation criteria can be consulted. It provides a document on criteria to be considered for the evaluation of these tools and many other resources and interesting information.

4.4.2. AVAILABILITY SURVEY OF DAST TOOLS

Table 19 shows some of the most common vulnerability DAST tools. Commercial and also open source tools can be found.

Table 19
DAST tools [Owasp, 2013]

Name	Owner	Licence	Platforms
Acunetix WVS	Acunetix	Commercial / Free (Limited Capability)	Windows
AppScan	IBM	Commercial	Windows
Burp Suite	PortSwiger	Commercial / Free (Limited Capability)	Most platforms supported
GamaScan	GamaSec	Commercial	Windows
Grabber	Romain Gaucher	Open Source	Python 2.4, BeautifulSoup and PyXML
Grendel-Scan	David Byrne	Open Source	Windows, Linux and Macintosh
Hailstorm	Cenzic	Commercial	Windows
IKare	ITrust	Commercial	N/A
N-Stealth	N-Stalker	Commercial	Windows
Netsparker	MavitunaSecurity	Commercial	Windows
NeXpose	Rapid7	Commercial / Free (Limited Capability)	Windows/Linux
Nikto	CIRT	Open Source	Unix/Linux
NTOSpider	NT OBJECTives	Commercial	Windows
ParosPro	MileSCAN	Commercial	Windows
QualysGuard	Qualys	Commercial	N/A
Retina	eEye Digital Security	Commercial	Windows
ScanDo	KaVaDo Inc	Commercial	Windows
SecurityQA Toolbar	iSec Partners	Commercial	Windows

Securus	Orvant, Inc	Commercial	N/A
SecPoint Penetrator	SecPoint	Commercial	Windows, Unix/Linux and Macintosh
Sentinel	WhiteHat Security	Commercial	N/A
Vega	Subgraph	Open Source	Windows, Linux and Macintosh
Wapiti	Informática Gesfor	Open Source	Windows, Unix/Linux and Macintosh
WebApp360	nCircle	Commercial	Windows
WebInspect	HP	Commercial	Windows
OpenVAS	OpenVAS	Open Source	Windows / Linux
WebKing	Parasoft	Commercial	Windows / Linux / Solaris
Trustkeeper Scanner	Trustwave SpiderLabs	Commercial	SaaS
WebScanService	German Web Security	Commercial	N/A
Websecurify	GNUCITIZEN / Websecurify	Commercial / Free	Windows, Mac OS, Linux and others
Wikto	Sensepost	Open Source	Windows
Zap	OWASP	Open Source	Windows, Mac OS, Linux
Ironwasp	Ironwasp	Open Source	Windows, Mac OS, Linux

4.5. WHITE BOX SECURITY ANALYSIS: REAL-TIME ANALYSIS TOOLS

Real-Time/Interactive Application Security Testing (RAST/IAST) tools operate in much the same way as a profiler or a debugger. Because they have the ability to see inside the process space of the running application, RAST tools can observe and record information about requests made to the application, the code the application executes as a result, and the values of variables inside the running program. Furthermore they can make this information available to the analyzer while an attack is taking place. RAST is similar to SAST in that it employs a collection of rules that define vulnerable behavior in terms of code-level

interfaces, and yet it also has DAST's ability to observe a concrete execution of the program.

4.5.1. ARCHITECTURE AND CHARACTERISTICS

RAST tools act directly on the executable code, observing the execution environment of the processes, how they work, their content in memory variables and application state in general. They also note the requests that are made to the web application and receive responses. This allows detecting vulnerabilities in the input fields to a concrete application in real time because it follows the operations of the application. Once detected the vulnerability some tools can take one of three actions:

- **Generate a report** after detecting no more vulnerabilities. HP FORTIFY SECURITYSCOPE [HP-Fortify, 2013] is an example of this type. AcuSensor also [Acunetix, 2013] as added functionality to Acunetix, which, as noted in the previous section, is an automated vulnerability scanner web application.
- **Block the attempted attack**, as does HP FORTIFY RTA [HP-Fortify, 2013]. RTA is the previous version of HP FORTIFY SECURITYSCOPE, so it has many similarities in architecture. They differ in the concept of what to do when a vulnerability is detected: blocking or reporting
- **Clean up the malignant request** to the web application, correcting the input to the application. SANER is an example of this type [Balzarotti, 2008].

The thesis of Benjamin Livshits "*Improving software security with static and runtime analysis required*" [Livshits 2006], can be consulted to understand better the functionality of RAST tools. This work concluded that the real-time analysis advantage is that these tools can detect all attacks (for attacks categories the tool is designed) because the tools are able of keeping track of how data flows through the application in run (real) time, analyzing if

the data can be an attack source. The thesis mentions the following characteristics about RAST tools:

- RAST tools do not have false positives, because of the complete historical information of each data type they manage.
- In addition RAST tools can recover from an attack by a vulnerability that can be exploited, by sanitizing the data input to the application when it is necessary.
- RAST tools detect vulnerabilities using a vulnerability specification or rule, by example, written in a language called PQL (program query language), which is translated into a finite state automaton non-deterministic (NFAs). When application is running, the generated NFAs running along with the application it is collecting information on relevant events. The tool try to find the automata reproduced during real-time analysis. When an NFA reaches an accepting state several actions can be taken. If there is a clause *replaces*, the unsafe action is replaced by another to recovery. If instead there is a clause e*xecutes*, RAST will run the code in this clause to generate a vulnerability report. Briefly, finding consultations PQL contained in the application involves 3 steps:
    1. PQL translation to NFAs.
    2. Code is inserted to monitor the application to record events related to NFAs which is being investigated at a given time in the application.
    3. Use a comparator to interpret query all states through which passes a NFAs to find all instances in the application.

In the same thesis [Livshits 2006] several other interesting considerations are discussed, in terms of the increasing **overload** that may involve applying a real-time analysis in an application. Adding code to monitor events provokes an overload of total application execution. One way to reduce this overload it is proposed: to use the results of static

analysis to reduce the monitoring code, eliminating sentences that cannot refer to objects involved in any "match" of a query. Reduction monitoring code using this technique can be 97% in the case of the application *roller* used as a benchmark. In other application benchmark such a *webgoat* overload is reduced by half with the optimized version. Figure 28 shows a comparison of overload of both versions, optimized with static analysis and non optimized, for 5 different applications.

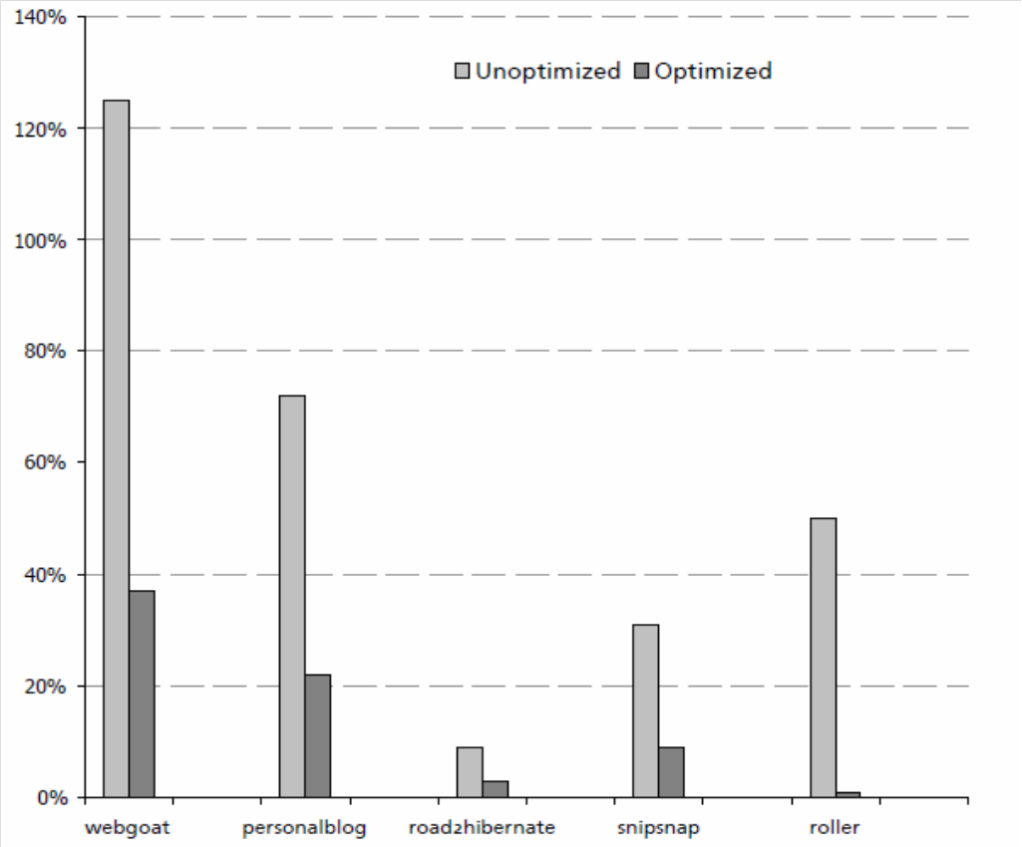

Figura 28. Run-time Overload comparison. [Livshits 2006]

In the same line as the mentioned thesis there are some other interesting works: [Lam. 2008], [Monga, 2009] or [Saxena, 2011**]**. Other different approach is that of [Wassermann, 2008] , in which they propose an algorithm for automated test generation, that uses data input and runtime values to analyze the code dynamically, semantic models of string operations, and handles operations whose argument and return values may not share a common type.

The joint use of source code static analysis and real-time analysis introduces the following section on hybrid security analysis tools that combine two or more of the types of static analysis, automated vulnerability scanners and real-time analysis.

Classification of RAST tools. It is difficult to establish a classification of these tools, taking into account that commercial tools do not give information about the techniques used in its building and architecture. RAST tools can be classified attending to:

- Purpose of the tool:
    1. **Runtime security protection** (report, block or sanitize attacks). HP WEBINSPECT REALTIME (SECURITYSCOPE) and HP FORTIFY RUNTIME (RTA) [HP-Fortify, 2013]**.**
    2. **Runtime security analysis** on testing penetration phase (report attacks). [Quotium, 2013]

- Technique of building:
    1. **Shadow memory for taint analysis** [Nagarajan, 2009]. Tools as TainTrace [Cheng, 2006] and Flextaint [Venkataramani, 2008] for C/C++ languages. The objective is to perform a taint analysis of input data to find out if data can make up an attack form [Newsome, 2005], [Cheng, 2006], [Tripp, 2009] or [Haldar, 20]. This technique keeps track of the propagation of untrusted (tainted) data during program execution. Tainted data may represent sources such as user input, packets from the network, or data read from specific files and devices. Taint tracing is based on a program's dynamic behavior. Unlike virus scanners that require known attack signatures, dynamic taint tracing can defend against future attacks. Another example of dynamic taint analysis for PHP is ARDILLA [Kiezun, 2009], that presents an automatic technique for creating inputs that expose SQLI and XSS vulnerabilities. The

technique generates sample inputs, symbolically tracks taints through execution (including through database accesses), and mutates the inputs to produce concrete exploits. Ours is the first analysis of which we are aware that precisely addresses second-order XSS attacks.

2. **Compiler techniques** [Wilander, 2003] pushes a tag into the stack and checks the stack to see if the tag is still there unchanged; if so, it continues - the normal execution flow, and if not, it aborts the program and gives an error message. StackGuard [Cowan, 1998] for C/C++ languages.
3. **Hardware defense mechanism**. Hardware modules can be included in the system to provide defense, with no modification of the program and with no destruction of the integrity of the pipeline. Heapdefender [Li, 2012], Heapbound [Devietti, 2008] for C/C++ languages.

4.5.2. RAST (IAST) TOOLS AVAILABILITY SURVEY.

There are RAST (IAST) tools for web and non-web applications, available commercially and also as open source.

Non-web applications. C/C++ languages:

- VALGRIND [Valgrind, 2013] is an open source instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. Valgrind is Open Source / Free Software, and is freely available under the GNU General Public License, version 2.
- INSURE++ [Insure, 2013] is a runtime memory analysis and error detection commercial tool for C and C++ that automatically identifies a variety of difficult-to-track programming and memory-access errors, along with potential defects and inefficiencies in memory usage. Errors such as memory corruption, memory leaks,

access outside of array bounds, invalid pointers, and the like often go undetected during normal testing, only to result in application crashes in the field. Insure++ will help you find and eliminate such defects in your applications to ensure the integrity of their memory usage.

- PURIFY [Purify, 2013] When a program is linked with Purify, corrected verification code is automatically inserted into the executable by parsing and adding to the object code, including libraries. That way, if a memory error occurs, the program will print out the exact location of the error, the memory address involved, and other relevant information. Purify also detects memory leaks. By default, a leak report is generated at program exit but can also be generated by calling the Purify leak-detection API from within an instrumented application.

Web application tools:

- HP FORTIFY RUNTIME (REAL-TIME ANALYZER) [HP-Fortify, 2013]. It identifies root security causes in deployed software, it provides accuracy defence without tuning, it enables customized attack responses and it delivers sophisticated protection out of the box.

    - It automatically blocks attacks for common vulnerabilities from inside applications and monitor Java and .NET applications and get data on actual attacks.

- IBM SECURITY APPSCAN STANDARD with GLASS BOX agent [IBM-Appscan, 2013]. It scans and tests for the latest threats with a desktop solution that offers:

    - A broad coverage of emerging threats, including Web 2.0 application vulnerabilities.

    - Advanced dynamic application security testing, also known as black-box analysis (DAST).
    - Glass-box testing, also known as runtime analysis or integrated application security testing (IAST).
    - Cross-Site Scripting (XSS) Analyzer for cutting-edge XSS detection and exploitation and JavaScript Security Analyzer for static taint analysis of client-side security issues

- SEEKER [Quotioum, 2013]. For testing processes, Seeker analyzes the application code and data as it runs, in response to simulated attacks. Seeker monitors application behavior and data flow across modules, components, tiers and servers to accurately identify application threats.

- ACUNETIX+ACUSENSOR [Acunetix, 2013]. It is a security technology with feedback from sensors placed inside the source code, while the source code is executed. Black box scanning does not know how the application reacts and source code analyzers do not understand how the application will behave while it is being attacked, see figure 29. When AcuSensor Technology is used, it communicates with the web server to find out about the web application configuration and the web application platform (for PHP and .NET) configuration. Once triggered from the ACUNETIX WVS scanner, the sensor gets a listing of all the files present in the web application directory, even of those which are not linked to through the website. It also gathers a list of all the web application inputs. Since it knows which kind of inputs the application expects, it can launch a broader range of tests against the application.

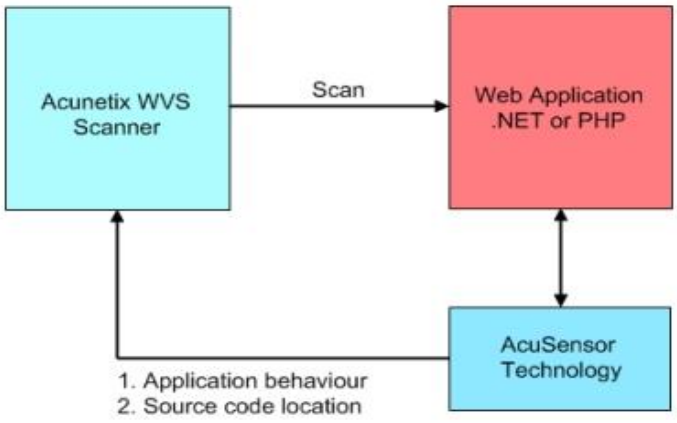

Figure 29 Acunetix+Acusensor. [Acunetix, 2013]

- ASPECT SECURITY [Aspect, 2013]. For *java* applications, this company develop the SaaS (Software as a service) CONTRAST tool that works by inserting passive sensors across the entire application stack. Events are fed into a powerful engine which detects vulnerabilities automatically and accurately.

- WHITEHAT SENTINEL [WhiteHat2, 2103] offers basic IAST capabilities through a vulnerability discovered by static analysis is correlated with DAST results, and also uses IAST within its mobile testing capabilities. WhiteHat Security provides only cloud-based testing as a service; no product option is available.

4.6. HYBRID ANALYSIS TOOLS

4.6.1. INTRODUCTION

A number of efforts of building hybrid security analysis tools or combinations of SAST, DAST or IAST tools are appearing, to leverage the individual characteristics of each type of tool and getting a hybrid tool which enhances the individual capabilities to increase the number of vulnerabilities detection and to decrease the false positive rate. As a result a hybrid tool can be more efficient as a whole.

As explained in detail before, two of the most effective automated vulnerability detection techniques available today are Dynamic Application Security Testing (DAST) and Static Application Security Testing (SAST). However, each method also has its weaknesses. DAST must explore the attack surface to launch a successful attack, but its knowledge of potential attack pathways is sometimes incomplete, inhibiting its ability to fully test an application. Additionally, DAST is able to detect only the symptoms of vulnerability, not its underlying cause within the code. DAST also cannot observe an application's internal behavior. For example, if a DAST tool launched a successful SQL injection attack that destroyed a database, the only symptom DAST might detect would be the appearance in HTTP of a "404 – Page Not Found" error message, with no insight into the error's cause. In this scenario, and others like it, DAST might register the attack as meaningless or even unsuccessful, and hence the underlying vulnerability would slip through undiagnosed. And while SAST offers greater coverage and is extremely proficient at finding potential vulnerabilities in source code, it does not produce concrete test cases to demonstrate the exploitability of the vulnerabilities it finds. Hybrid tools can be classified in earlier tools that correlated DAST and SAST tools results and the last hybrid tools that incorporate IAST tools.

First-generation hybrid analysis: A vital first step that doesn't go far enough the allure of hybrid analysis is obvious: Combining the results from DAST and SAST holds the potential to maximize the advantages of each the vulnerability substantiation of dynamic testing with the application coverage, root-cause analysis, and line-of-code specifics of static testing.

The first hybrid analysis tools were introduced just a few years ago. They help enterprises conduct more complete security testing, validate results through enhanced correlation, and reduce the time and expense of resolving application security issues. And yet they do not go far enough when it comes to realizing the full potential of hybrid analysis. One key reason is that first generation hybrid tools work by correlating results *only after* testing is complete.

However a real hybrid tool combining black box and static analysis exists, as mentioned in [Tripp, 2011], where they present a commercial-grade hybrid-analysis solution for automated security assessment of client-side JavaScript code. This approach brings together the advantages of the white-box and black-box methodologies while overcoming their weaknesses. A black-box component interacts with the subject web application and collects pages that contain client-side JavaScript code. The pages are then analyzed using static taint analysis to detect security vulnerabilities. The black-box component provides URLs and other pieces of dynamic information that contribute toward specializing the static analysis, making it much more precise and effective than its baseline version, as the authors demonstrate empirically in their work.

One of the limitations of first-generation hybrid tools is that, because vulnerability correlation happens after attacks have occurred and testing is concluded, important opportunities for more thorough analysis can be missed. Another issue is that it can be difficult to readily align the results of DAST and SAST analysis because the two technologies process two very different types of information under very different circumstances. DAST examines web traffic while applications are under attack; its output is oriented around HTTP traffic. In contrast, static analysis scrutinizes source code and configuration files. Therefore, in order to match up results, the correlation algorithm must track down how a given vulnerability described within the relevant HTTP traffic by DAST links to a specific line of code or configuration file identified through SAST. This correlation can be difficult to perform accurately, which undercuts the ability of hybrid to make more rapid remediation possible. Additional concerns with first-generation hybrid involve questions of accuracy and application coverage. By focusing on vulnerabilities detected by DAST *and* SAST, and hence with a high degree of correlation between the two techniques, first-generation systems may inadvertently downplay the potential risk from vulnerabilities detectable by only one method or the other, but not by both. Moreover, as

mentioned previously, because a DAST tool lacks key information about the interior landscape of an application, the attack surface it targets may be incomplete.

.

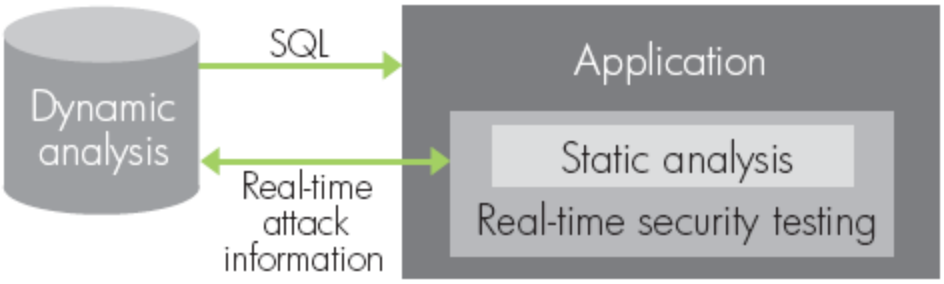

Figure 30. Hybrid analysis information flow [HP-fortify, 2013]

Let consider the introduction of RAST to the SQL injection scenario mentioned previously (Figure 30). In this instance, RAST is able to detect that an input parameter contains SQL metacharacters. It then observes the SQL statements the application assembles, and it can recognize when a malicious query is about to be delivered to the database. It communicates all of this information in real time to DAST, which is then able to capture and report the incident as a vulnerability.

Second generation of hybrid analysis. RAST (IAST) technology provides the foundation for the next generation of hybrid analysis, real-time hybrid analysis. Real-time hybrid analysis significantly enhances code coverage and accuracy, while fully automating the process of identifying, locating, organizing, and ranking the severity of vulnerabilities in code. Using **real-time hybrid analysis** [Livshist 2006], [Artho, 2005], [Lam, 200], [Monga, 2009], organizations can resolve their most critical software security issues faster and more cost-effectively, than any other available analysis technology. The degree of synergy between static and dynamic analysis is discussed in [Mock 2003], [Ernst 2003]. Key benefits include:

- **Identification of more vulnerabilities**: RAST technology enables analysis tools to investigate more of an application's attack surface because it is capable of observing application details statically and at runtime. For example, RAST conveys critical details about file systems and the contents of configuration files to enable it to target areas of code it otherwise would not have known to attack.
- **More accurate diagnosis**: RAST also enhances vulnerability diagnosis by observing code execution in response to an attack, enabling DAST to know whether an attack has succeeded and therefore represents a vulnerability. SAST can reduce the amount of instrumentation code to reduce overhead during runtime analysis [Livshist 2006]. IAST could check the results of SAST de [Artho, 2005].
- **Faster remediation of critical issues**: By offering an unprecedented view of application behavior, made possible through RAST, real-time hybrid analysis does not only provide details of an attack and their relative level of impact, it also exposes a vulnerability's root cause in code. With this explicit guidance, security and development teams can rapidly address security issues.
- **Better understanding of vulnerabilities by distilling common causes**: One root cause is often responsible for generating thousands of vulnerability symptoms. Real-time hybrid analysis is able to group all symptoms (vulnerabilities) that share a common root cause, enabling teams to quickly eliminate multiple reported vulnerabilities by resolving a single underlying problem.
- **Simplified software security management**: Leveraging RAST, real-time hybrid analysis generates a single unified report combining DAST and SAST analysis [HP-Fortify, 2013] that greatly simplifies management and oversight of remediation efforts, enabling teams to quickly determine which vulnerabilities to address first for their particular circumstances. The report lists all discovered vulnerabilities, organized by such traits as:

- Impact of an exploit
- Degree of correlation
- Common root causes
- Location in code

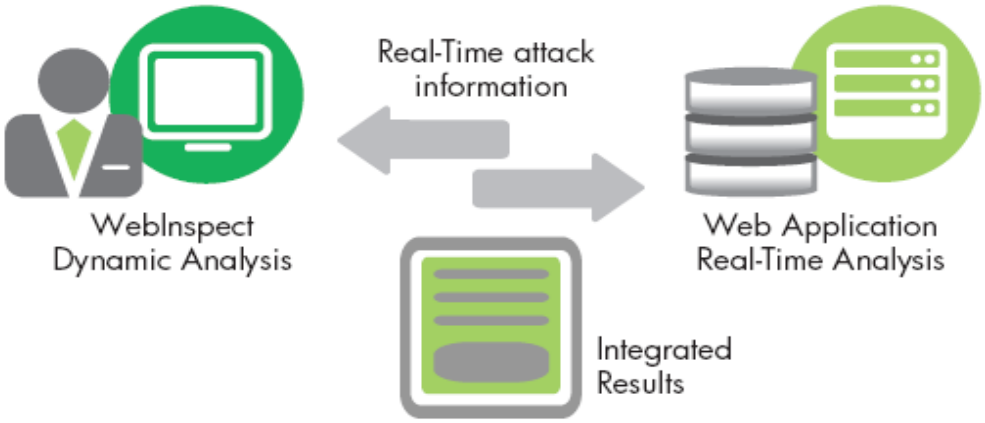

Figure 31. HP FORTIFY HIBRID ANALYSIS [HP-Fortify, 2013]

By example, figure 31 shows the HP FORTIFY HIBRID ANALYSIS composed by a IAST tool (SecurityScope) that interchanges real-time attack information with a DAST tool (Webinspect). After this, the results can be correlated with SAST analysis (SCA) to decrease false positives and increase the number of detections or true positives.

4.6.2. HYBRID TOOLS TYPES.

Hybrid tools are appearing in the last recent years. This section is a survey of hybrid tools types available as academic research, commercial tools implementations or open source tools. Examining the academic research is very important to know the last tendencies on techniques used in hybrid security analysis tools.

As commented in the previous section, the main objective of building hybrid security analysis tools or combinations of SAST, DAST or IAST tools is to leverage the individual

characteristics each type of tool and getting a hybrid tool with enhance the individual capabilities to increase the number of vulnerability detection, decrease the false positive rate and to increase the total vulnerability coverage. As a result the hybrid tool can be more efficient as a whole. The following types of hybrid tools can be found.

- SAST- DAST.
- SAST-RAST
- DAST-RAST
- SAST, DAST y RAST.

SAST- DAST. As mentioned in previous section the first generation of hybrid tools was limited to correlate individual results obtained with each type of tool. There have no many implementations of collaborative tools composed of SAST and DAST tools, for example [Csallner, 2005], [Csallner, 2006] describe several implementation of this type of tools. Despite its simplicity they can find bugs that would require complex static analysis efforts. Check 'n' Crash [Csallner, 2005] uses JCrasher as a post-processing step to the powerful static analysis tool ESC/Java. JCrasher [Csallner, 2004] is a simple, mostly dynamic analysis tool that generates JUnit test cases. As a result, Check 'n' Crash is more precise than ESC/Java alone and generates better targeted test cases than JCrasher alone. DSD-Crasher [Csallner, 2006] adds a reverse engineering step to Check 'n' Crash to rediscover the program's intended behavior. This enables DSD-Crasher to suppress false positives with respect to the program's informal specification.

Babic et al [Babic, 2011] present a new technique for exploiting static analysis to guide dynamic automated test generation for binary programs, prioritizing the paths to be explored. The technique is a three-stage process, which alternates dynamic and static analysis. Preliminary experiments on a suite of benchmarks extracted from real applications

show that static analysis allows exploration to reach vulnerabilities it otherwise would not, and the generated test inputs prove that the static warnings indicate true positives.

Omer Tripp et al [Tripp, 2011] present a commercial-grade hybrid-analysis solution for automated security assessment of client-side JavaScript code. This approach brings together the advantages of the white-box and black-box methodologies while overcoming their weaknesses. A black-box component interacts with the subject web application and collects pages that contain client-side JavaScript code. The pages are then analyzed using static taint analysis to detect security vulnerabilities. The black-box component provides URLs and other pieces of dynamic information that contribute toward specializing the static analysis, making it much more precise and effective than its baseline version.

Veracode [Veracode, 2013] offers an online SaaS service to analyze the security of applications using SAST, DAST and manual penetration testing correlating their results.

Open source tools as IRONWASP [Ironwasp, 2013] for the client side *javascript* code (AJAX engines of RIA applications or javascript code generated on server side but executed on client side) give the possibility of performing static analysis besides penetration testing. Also GRABBER [Grabber, 2013] performs Hybrid analysis testing for PHP application using PHP-SAT (PHP source code analyzer) and JavaScript source code analyzer with JavaScript Lint.

SAST-RAST. These hybrid tools perform collaborative static white box and runtime white box security analysis. The architectures and purposes of these tools can be focused from different points of view.

SECURFLY is a first example of academic research SAST-RAST hybrid tool exposed in the thesis of Benjamin Livshits [Livshits, 2006], where the advantage of real time analysis is discussed. It can detect all attacks in a particular category because they follow the trail of how data flows through the application. It has no false positives, because it has perfect

historical information of each data type. It can also recovers from an attack against vulnerabilities before it can be exploited by cleaning up the data entry application when necessary in a production time, by adding code to monitoring events increasing the total web application overhead. SECURFLY minimizes the total overhead using the results of previous static analysis to reduce the monitoring code, eliminating statements that cannot refer to objects involved in any "match" of a given query. The monitoring code reduction by this technique may be 97% when the application roller is used as a benchmark. Others examples of SAST-RAST tools can be examined in [Halfond, 2006] [Artho, 2005]. In these cases, the tools are used in test phase to discover security vulnerabilities while, in Livshits solution real-time analysis, SAST is optimized for real-time protection, blocking the attacks, with the application in production.

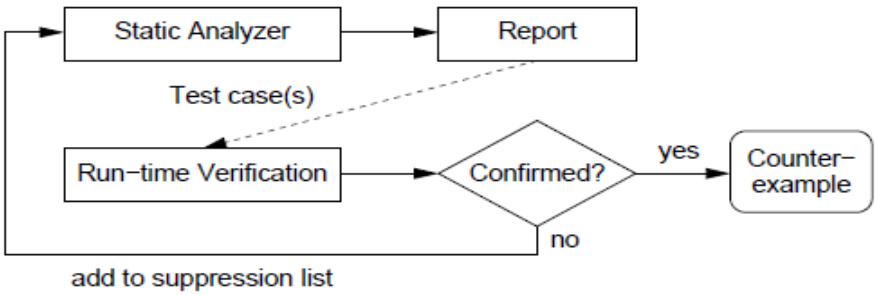

Figure 32. JNUKE architecture [Artho, 2005]

JNUKE [Artho, 2005] (figure 32) uses SAST to check for vulnerabilities detected, generating test cases for each one, that are verified by RAST in a test analysis phase.

AMNESIA (Analysis and Monitoring for Neutralizing SQL-Injection Attacks), [Halfond, 2006] (http://www-bcf.usc.edu/~halfond/amnesia.html) is a tool that implements a technique for detecting and preventing SQL INJECTIONS. AMNESIA uses a model-based approach that is specifically designed to target SQLIAs and combines static analysis and runtime monitoring. It uses static analysis to analyze the Web-application code and automatically builds a model of the legitimate queries that the application can generate. At

runtime, the technique monitors all dynamically-generated queries and checks them for compliance with the statically-generated model. When the technique detects a query that violates the model, it classifies the query as an attack, prevents it from accessing the database, and logs the attack information.

PHP VULNERABILITY HUNTER [Hunter, 2013] is a whitebox PHP web application fuzzer, that scans for several different classes of vulnerabilities via static and dynamic analysis. By instrumenting application code, PHP Vulnerability Hunter is able to achieve greater code coverage and uncover more bugs.

WEBSSARI (Web application Security by Static Analysis and Runtime Inspection) [Huang, 2004] acts as an extension to a language's existing type system. It is implemented as a framework for extending existing script languages with our system. Currently, WebSSARI supports PHP, one the most widely used Web application programming language. Given the corresponding grammar, WebSSARI can also support other languages used for Web application programming. WebSSARI automatically inserts runtime guards in potentially insecure sections of code, meaning that a piece of PHP code will be secured immediately after WebSSARI processing even in the absence of programmer intervention. Induced overhead is low because the number of insertions is reduced to a minimum when information gathered from static analysis is utilized. Users can add annotations to further reduce this number, possibly to zero.

F4F [Sridharan , 2011] is a novel solution that augments taint analysis engines with precise framework support and allows for handling new frameworks without modifying the core analysis engine. In F4F, a framework analyzer first generates a specification of an application's framework-related behavior in a simple language called WAFL (for Web Application Framework Language). The WAFL specification is generated based on both lightweight code analyses and information found in other relevant artifacts such as

configuration files. The taint analysis then uses the WAFL specification to enhance its analysis of the application. The code analyses are implemented using the Watson Libraries for Analysis WALA (http://wala. sourceforge.net).

JPREDICTOR [Chen, 2006] (http://fsl.cs.uiuc.edu/jPredictor/) is a tool for detecting concurrency vulnerabilities in java programs. It is composed of two major components: the program instrumentor and the trace predictor (Figure 33). The program instrumentor instruments the program under testing with instructions that log the execution. To reduce the runtime overhead caused by monitoring, only partial information is logged during execution. The trace predictor analyzes the logged execution trace to predict potential bugs using sliced causality.

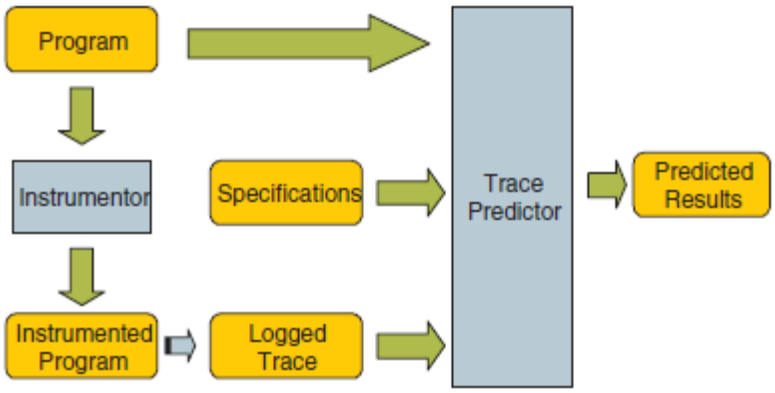

Figure 33. JPREDICTOR [Cheng, 2006]

If a possible bug is detected, JPREDICTOR generates an abstract execution trace leading to it, which explains how the bug can be hit in a real execution. As shown in Figure 33, the trace predictor consists of four stages: the pre-processor, the trace slicer, the *VC* calculator, and the property checker. The role of the pre-processor is two-fold. First, it constructs a more informative trace from the partially logged trace using static analysis on the original program, providing a foundation for the subsequent analysis. Second, it identifies all the shared locations in the observed execution, which are critical for a precise predictive analysis. The slicer scans the re-constructed trace, producing a trace slice for every property

to check. The generated slices are fed into the VC calculator, which computes the sliced causality. In the last stage, the property checker verifies the execution against the desired property using the computed sliced causal with lock-atomicity.

PHAN [Monga, 2009] presents a hybrid analysis framework that blends together the strengths of static and dynamic approaches for the detection of vulnerabilities in web applications: a static analysis, performed just once, is used to reduce the run-time overhead of the dynamic monitoring phase. PHAN is able to statically analyze PHP bytecode searching for dangerous code statements; then, only these statements are monitored during the dynamic analysis phase.

SANER [Balzarotti, 2008] analyzes the use of *custom sanitization routines* to identify possible XSS and SQL injection vulnerabilities in web applications. In the context of this work, any function that takes as input a (string) value and that can output a modified version of this input is considered a possible sanitization routine. In particular, this includes functions that replace or remove certain characters or substrings from their input (such as the PHP functions *str* replace). This requires the system to model the ways in which these functions can modify the application's input. To this end, SANER uses a combination of static and dynamic program analysis techniques. The core of the approach consists of a static analysis component that uses data flow techniques to identify the flows of input values from sources to sensitive sinks. This component is based on the open-source web vulnerability scanner called Pixy. In its current form, Pixy only provides information about the presence of data flows between sources and sinks. In addition, it can determine whether built-in sanitization operations (such as html entities) are applied on all paths between a source (input data) and a sink (instruction where exploit is executed). To achieve this, it is sufficient to assign one of two types (or labels) to each program variable: tainted or untainted. Whenever input is read from a user and stored in a variable, the variable initially receives the label tainted. Once a variable is sanitized, its label is set to untainted.

Because the number of false positives can be large (depending on the application), authors augment the static analysis with an additional dynamic analysis phase. The goal of the dynamic phase is to examine all those program paths from input sources to sensitive sinks that the static analysis has identified as suspicious. More precisely, using dynamic analysis, we attempt to confirm the existence of a potential security vulnerability (reported by the static analysis phase) by finding program inputs that can bypass the sanitization routines and reach the sensitive sink. To this end, the dynamic analysis is used to simulate the effect of the program operations on the input while it is propagated to the sensitive sink (in particular, sanitization operations are of interest). Of course, the analysis is performed by exercising the code with a large set of different input values, which contain many different ways of encoding and hiding malicious characters. In some sense, the dynamic analysis phase automates the actions of a programmer when a static analysis tool reports a warning. See figure 34 with execution results of SANER against several applications.

Application	Sinks		Static Analysis		Dynamic Analysis		
	Total	With Sanitization	Sinks Eliminated (Basic)	Sinks Eliminated (Advanced)	Sinks Analyzed	Sinks Vulnerable	Not Vulnerable
Jetbox 2.1	311	7	5	0	2	1	1
MyEasyMarket 4.1	737	50	0	45	5	5	0
PBLGuestbook 1.32	41	5	2	0	3	3	0
PHP-Fusion 6.01	1,015	67	19	0	48	4	44
Sendcard 3.4.1	84	10	2	0	8	1	7
Totals	2,188	139	28	45	66	14	52

Figure 34. SANER results with application benchmarks. [Balzarotti, 2008]

Pranith Kumar D. [Pranith, 2009] presents another hybrid approach, which utilizes the strength of both dynamic and static analysis to efficiently detect security vulnerabilities like buffer overflow, dangling pointers and memory leaks. Executable is first instrumented using PIN library, a instruction trace tool and memory profiling, [Keugh, 2005], to extract the exact control flow and register bounds. Executable is disassembled then to get the assembly

code. Control flow and register bounds are then used in static analysis in which constraint bound check is performed on the slice generated. Finally memory errors obtained as discussed earlier are reported.

SANTE [Chevaro, 2012] (Static ANalysis and TEsting) combines value analysis, program slicing and structural dynamic testing for the verification of C programs. SANTE is implemented using Frama-C, an open-source framework for static analysis of C code, and PathCrawler, a structural test generation tool.

Ruoyu Zhang et al. [Zhang, 2011] proposes a novel approach, and realizes it as a framework. Moreover, we verify the practicality of the framework by building a vulnerability discovery tool on it. Their contributions are summarized as follows:

- They propose SDCF (Static and Dynamic Combined Framework).
- They implement a tool to detect latent software vulnerabilities. They present and evaluate LSVD (Low-overhead Software Vulnerability Detector), an SDCF based tool to discover software vulnerabilities. LSVD cannot only detect software vulnerabilities being exploited at runtime, but also find the unexecuted code containing weak spots.

DAST-RAST. An example of this hybrid tool implementation is the approach of Andrey Petukhov, Dmitry Kozlov [Petukhov, 2008] that incorporates advantages of penetration testing and dynamic analysis. This approach effectively utilizes the extended Tainted Mode model. The prototype implementation of our approach consists of three main components: the dynamic analysis module, which is an extension of the Python interpreter that collects traces of the executed application, the analyzer, which builds DDGs for the collected traces and performs analysis thereof, and the penetration testing (Owasp Webscarab tool) module that submits input data (both normal and malicious) to the web application. This proposal

combines dynamic analysis approach with penetration testing. Arguments lying behind it are as follows:

- During penetration testing real attacking patterns are submitted to an application. By combining penetration testing with dynamic analysis the scope of the web application view is widened, so error suppression and custom error pages are not an issue.
- By submitting real attacking patterns it is possible to test data validation routines for correctness, not just trust them blindly.
- The dynamic analysis implementation knows the web application from inside, so more accurate penetration test cases can be generated.

These kind of hybrid tools are mainly commercial solutions of leader software companies:

- IBM SECURITY APPSCAN STANDARD [IBM-Appscan, 2013], scans and tests for the latest threats with a desktop solution that offers:
    - Broad coverage of emerging threats, including Web 2.0 application vulnerabilities.
    - DAST Advanced dynamic application security testing, also known as black-box analysis.
    - IAST Glass-box testing, also known as runtime analysis or integrated application security testing.
    - Cross-Site Scripting (XSS) Analyzer for cutting-edge XSS detection and exploitation.
    - JavaScript Security Analyzer for static taint analysis of client-side security issues.
- HP FORTIFY WEBINSPECT REAL TIME [HP-Fortify, 2013] is the combination of HP WebInspect working in concert with HP Fortify SecurityScope. HP

WebInspect delivers core platform-independent dynamic security analysis, broad security assessment and accurate web application security scanning results. HP Fortify SecurityScope is an agent that is installed on a target application server and is designed to detect when HP WebInspect scans the target, providing application information that WebInspect otherwise could not obtain. When used together, HP WebInspect Real-Time stimulates the application through automated, external security attacks, and then gathers internal, code-level vulnerability information by observing the attacks in the code as they happen. HP WebInspect Real-Time improves the accuracy of scan results, improves application coverage, reduces the time required to validate vulnerabilities and offers developers key information that allows them to find and fix the vulnerabilities more easily.

- ACUNETIX+ACUSENSOR [Acunetix, 2013]. Is a security technology with feedback from sensors placed inside the source code while the source code is executed combining IAST and DAST analysis. Black box scanning does not know how the application reacts and source code analyzers do not understand how the application will behave while it is being attacked. See figure 25. When AcuSensor Technology is used, it communicates with the web server to find out about the web application configuration and the web application platform (for PHP and .NET) configuration.

SAST-DAST-RAST(IAST). An example of this type of hybrid tool is the academic research prototype SDAPT [Halfond, 2011].The authors propose a new approach to penetration testing based on previous information gathering and the response analysis phases. One of the key insights of this approach is that many of the limitations of the previous approaches can be addressed by assuming that penetration testers have access to the source code or executable of the web application. This assumption is realistic in the context of in-house penetration testing and it is consistent with the best practices defined by

both OWASP and OSSTMM, which assume that potential adversaries have access to one or more versions of an application's source code. The proposed penetration testing approach leverages several newly developed analyses that make use of the web application source code. To improve the information gathering phase, the approach builds on a static analysis technique for discovering inputs vectors that was developed by two of the authors in previous work. The proposed approach also improves the response analysis phase by incorporating the use of precise dynamic analyses to determine when an attack has been successful. The dynamic analysis allows the approach to perform fully automated detection of successful attacks. SDAPT is used in an extensive empirical evaluation of the proposed approach. In this evaluation, the authors used SDAPT to perform penetration testing on nine web applications, and SDAPT's performance was compared with that of two state-of-the-art penetration testing tools. The empirical results show that the approach was able to (i) exercise the subject applications more thoroughly and (ii) discover a considerably higher number of vulnerabilities than the traditional penetration testing approaches.

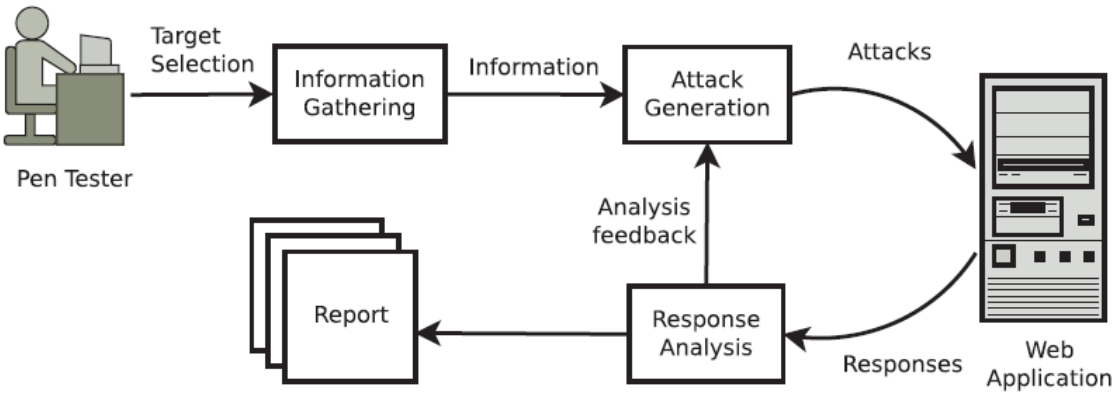

Figure 35. SDAPT tool. [Halfond, 2011]

As is shown in figure 35, SDAPT consists of:

- An approach for penetration testing based on improved input vector identification and automated response analysis.

- An implementation of the approach in a prototype tool that targets SQL Injection and Cross Site Scripting vulnerabilities.

SAST-DAST-RAST (IAST) hybrid tools are mainly commercial solutions of leaders software companies:

- IBM SECURITY APPSCAN ENTERPRISE [IBM-Appscan, 2013] performs Correlation and triage security testing results from dynamic black box testing and IAST glass box solution along with static (white box) scans.

- HP FORTIFY HYBRID ANALYSIS [HP-Fortify, 2013] composed by a IAST tool (SecurityScope) that interchange real-time attack information to a DAST tool (Webinspect). After this the results can be correlated with SAST analysis (Fortify SCA) to decrease false positives and increase the number of detections or true positives.

- WHITEHAT SENTINEL [WhiteHat2, 2103] offers basic IAST capabilities through a vulnerability discovered by static analysis is correlated with DAST results, and also uses IAST within its mobile testing capabilities. WhiteHat Security provides only cloud-based testing as a service; no product option is available.

4.7. METHODOLOGIES FOR TOOLS EVALUATION.

An adequate methodology is a necessary instrument to perform an assessment of any of the different security tools categories. A well-defined and repeatable methodology allows an evaluation of the performance of vulnerabilities detection capacity of security tools. The methodology adopted should use a selected benchmark with a well-known set of security vulnerabilities. A security tool has the best performance against a benchmark if it has the best balance between detecting the highest number of *true positives* and having few *false positives.* ". In accordance with Gray [Gray, 1993] a benchmark must be "repeatable, portable, scalable, representative, require minimum changes in the target tools and simple to use. The main goal is to derive a particularized methodology for comparing the performance of each categories of security tool described in previous sections as SAST, DAST, IAST or HYBRID tools. The methodology searches their effectiveness mainly in terms of the number of detected security vulnerabilities and uses a selected and widely accepted set of metrics necessary to analyze the results and extract adequate conclusions

Several mayor methodologies initiatives can be considered first, in order to develop a particularized methodology:

- The NIST SAMATE project [Samate, 2013]
- WASC Static Analysis Technologies Evaluation Criteria project
- WASC Web Application Security Scanner Evaluation Criteria project

NIST SAMATE project [Samate, 2013]. SAMATE includes a methodology to perform an assessment of SAST tools. The methodology for using SAMATE is detailed in two different documents. The first one, NIST SP 500-268 [NIST268, 2007], is the minimum functional specification for source code security analysis tools and includes:

- **Functional requirements** that must have every source code analysis tool: identifying a minimum set of software vulnerabilities in source code, reporting of

the vulnerabilities found, their type and location, a low number of false positives, and producing a findings report.

- **Tables with source code vulnerabilities**, for web and non-web applications based on Common Weakness Enumeration (CWE) from MITRE Corporation [Mitre, 2013]. Its CWE identification and a small description, organized by classes of vulnerabilities. Each of them must be considered also for a different number of cases, depending on *code complexity* a concept referred to the way of storing a memory variable. A general list of these types of structures, adapted from Kratkiewicz thesis [Kratkiewicz, 2005] is also provided. Each different *code complexity* type, such as fixed or variable loops, memory indexing nested within indexing, local vs. global scope, and others, may require additional analytical capabilities.

The second document (NIST SP 500-270) [NIST260, 2009] specifies the test plan to determine how well a particular source code security analysis tool conforms to the requirements specified in the first document. It includes:

1. The tests implementation: installation, test selection, execution of the tool for every case and how to interpret the obtained results.
2. The different test suites available for C, C++ and Java, including complete test-suites for testing all the source code vulnerabilities of Table 1.

As SAMATE defines (NIST SP 500-270) [NIST270, 2009], "*a test suite is a collection of test cases explicitly selected for a special purpose*". Each test case contains an atomic program that ensures that a specific functionality required by NIST SP 500-268 [NIST268, 2007] can be performed by the tool under testing. Test suites and their test cases are stored in SAMATE Reference Dataset [Samate, 2013].

Also SAMATE has a publication about Software Assurance Tools: Web Application Security Scanner Functional Specification Version 1.0. [NIST269, 2008]. It specifies the functional behavior of one class of software assurance tool: the web application security scanner tool. Due to the widespread use of the World Wide Web and proliferation of web application vulnerabilities, application level web security and assurance requires major attention. This specification defines a minimum capability to help software professionals understand how a tool will meet their software assurance needs.

WASC Static Analysis Technologies Evaluation Criteria project (SATEC). [Wasc, 2103]. The goal of the SATEC project is to create a vendor-neutral set of criteria to help guide application security professionals during the process of acquiring a static code analysis technology that is intended to be used during source-code driven security programs. This document provides a comprehensive list of criteria that should be considered during the evaluation process. Different users will place varying levels of importance on each feature, and the SATEC project provides the user with the flexibility to take this comprehensive list of potential criteria, narrow it down to a shorter list which contains the most important or most relevant set of criteria, assign weights to each criterion, and conduct a formal evaluation to determine which scanning solution best meets the user's needs.

The aim of this document is not to define a list of requirements that all static code analysis vendors must provide in order to be considered a "complete" solution. In addition, evaluating specific products and providing the results of such an evaluation is outside the scope of the SATEC project. Instead, this project provides criteria and documentation to enable anyone to evaluate static code analysis tools and services and choose the product that best fits their needs. The purpose of this document is to develop a set of criteria that should be taken into consideration while evaluating static code analysis tools or services for

security testing. The vendor-neutral criteria defined in this document are selected using a consensus-driven review process comprised of volunteer subject matter experts. Every organization is unique and has a unique software development environment, this document aims to help organizations achieve their application security goals through acquiring the most suitable tool for their own unique environment. The document will strictly stay away from evaluating or rating vendors. However, it will focus on the most important aspects of static code analysis technologies that would help the target audience identify the best technology for their environment and development needs.

Taking a decision regarding the best static code analysis tool or service to acquire could be a daunting task. However, preparation for such a task could be very helpful. Every technology is unique so as your corporate environment. The following is a set of information you need to gather which could make the decision much easier to take. In its appendix A, SATEC recommends to prepare a cheat sheet:

- A list of the programming languages used in the organization.
- A list of the frameworks and libraries used in the organization.
- Who will be tasked to perform the scan
- How the tool or service will be integrated into the Software Development Lifecycle
- How will the developers see the scan results
- Budget allocated to the technology purchase including the hardware to run the machine (if any)
- A decision on whether the code (or the binaries) is allowed to be scanned outside the organization.

The project covers many aspects relational to SAST tools as:

- Deployment:

- Technology Configuration Support:
- Technology Support
- Scan, Command and Control Support
- Testing Capabilities
- Industry Standards Aided Analysis
- Product Signature Update
- Triage and Remediation Support
- Reporting Capabilities
- Enterprise Level Support

WASC Web Application Security Scanner Evaluation Criteria project (WASSEC) **[**Wasc, 2103]. This methodology is a set of guidelines to evaluate web application scanners on their ability to effectively test web applications and identify vulnerabilities. It covers areas such as crawling, parsing, session handling, testing, and reporting.

The goal of WASSEC is to create a vendor-neutral document to help guide web application security professionals during web application scanner evaluations. This document provides a comprehensive list of features that should be considered when conducting a web application security scanner evaluation. Different users will place varying levels of importance on each feature, and the WASSEC provides the user with the flexibility to take this comprehensive list of potential scanner features, narrow it down to a shorter list of features that are important to the user, assign weights to each feature, and conduct a formal evaluation to determine which scanning solution best meets the user's needs.

The aim of this document is not to define a list of *requirements* that all web application security scanners must provide in order to be considered a "complete" scanner, and evaluating specific products and providing the results of such an evaluation is outside the scope of the WASSEC project. Instead, this project provides the tools and documentation

to enable anyone to evaluate web application security scanners and choose the product that best fits their needs. The project covers the major aspects a web application scanner tools should be meet (NIST Special Publication 500-269, "Software Assurance Tools: Web Application Security Scanner Functional Specification Version 1.0" [NIST269, 2008], contains minimal requirements for mandatory and optional web application scanner features): *protocol support, authentication, session management, crawling, parsing, testing command and control, reporting and advice* for conducting a scanner evaluation in each phase of the evaluation process.

This work uses a derived, particularized and enhanced methodology based on commented previous projects. It compiles a suite of synthetic benchmarks with support for multiplatform and for the C/C++, java, J2EE and PHP languages, with coverage for most of vulnerabilities categories. The methodology used will be detailed in the next section.

4.8. BENCHMARKS FOR TOOLS SECURITY EVALUATION.

The methodology used to perform a security tools assessment should select a benchmark with a well-known set of security vulnerabilities. A tool has the best performance against a benchmark if it has the best balance between detecting the highest number of true positives and having few false positives. As said before, the benchmark must be ‘‘repeatable, portable, scalable, representative, require minimum changes in the target tools and simple to use’’, in accordance with Gray [Gray, 1993].

Elisabeth Fong et al., established a procedure for evaluating web applications [Fong, 2008] and **described the design of a test suite** for thorough evaluation of web application scanners (DAST). This approach allows us to develop an extensive test suite that can be easily configured to switch on and off vulnerability types and select a level of defense. The experiments suggest that the test suite is effective at distinguishing the tools based on their vulnerability detection rate; in addition, its use can suggest areas for tool improvement.

As in many other disciplines, a benchmark is needed for comparing tools. A benchmark should serve to agree in the way to compare the results, trying to reach a consensus for a trade-off between false positives and false negatives. According to Martin and Barnum in [Martin, 2008] maybe this benchmark could also motivate its use as a referential standard by community players as, for example, OWASP (Open Web Application Security Project), the SANS (SysAdmin, Audit, Networking, and Security) Institute, CERIAS (Center for Education and Research in Information Assurance and Security) and many others.

As mentioned and following Gray [Gray, 1993], a good benchmark must have a number of characteristics:

- Its cost should be comparable to the value of the results.
- To be credible, a benchmark for vulnerability detection tools must report similar results when run more than once over the same tool. It must be easily portable, as

must allow the comparison of different tools in a given domain. In practice, the workload is the component that has more influence on portability, as it must be able to exercise vulnerability detection capabilities of a set of tools in the domain.

- For reporting relevant results, a benchmark must represent real world. The representativeness must be based on realistic code and must include a realistic set of vulnerabilities and it should be scalable to increase the representativeness.
- A benchmark must require minimum changes in the target tools evaluated.
- Finally a benchmark must be as easy to implement and run as possible. Ideally, the benchmark should be provided as a computer program ready to be used or, if that is not possible, as a document specifying in detail how the benchmark should be implemented and executed. In addition, the benchmark execution should take the smallest time possible.

Taking into account that a false positive is a reported vulnerability in a program that is not really a security problem and a false negative is a vulnerability in the code which is not detected by the tool. When analyzing the results for test suite 46 (SAMATE REFERENCE DATASET (SRD), [Samate, 2013]) we must remind that each test case is usually a fixed case, corresponding to a test in test suite 45. If the tool detects a vulnerability, this is a false positive. For example, the following code in figure 36, shows the test case 1898 (part of test suite 46), related with a resource injection CWE, but corrected by using a function (*allowed*), that uses a white listing of file names, to validate the inserted filename. If the function *allowed()* does not exist, the argument *argv*[1] could be any file with any valid path. If one of the tools detects a "resource injection" vulnerability at the corresponding line in the code, this is a false positive (figure 36).

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
const char *whitelist[5] = {
```

```
		“users_site.dat”,
		“users_reg.dat”,
		“users_info.dat”,
		“admin.dat”,
		“services.dat.cxx”
};
int allowed(const char *_str) {
		for (unsigned i = 0; i < 5; i++)
		{
			if (!strcmp(whitelist[i], _str))
				return 1;
		}
		return 0;
}
void printLine(const char *fileName)
{
		FILE *fp = (FILE *)NULL;
		if ((fp = fopen(fileName, “r”)))
		{
			char buff[512];
			if (fgets(buff, 512, fp))
			{
				printf (“%s\n”, buff);
			}
			fclose(fp);
		}
}
int main(int argc, char *argv[])
{
		if (argc > 1)
		{
			if (allowed(argv[1]))

				printLine(argv[1]);
		}
		return 0;
}
```

Figure 36. SAMATE test case 1898 of test suite 46.

The test case 1897 is the test case associated for the previous 1898 test case. The test case 1897 is designed with resource injection vulnerability (figure 37). The argument *argv*[1] could be any file with any valid path.

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>


void printLine(const char *fileName)
{
		FILE *fp = (FILE *)NULL;
		if ((fp = fopen(fileName, "r")))
		{
			char buff[512];
			if (fgets(buff, 512, fp))
			{
				printf ("%s\n", buff);
```

```
                }
                fclose(fp);
        }
}


int main(int argc, char *argv[])
{
        if (argc > 1)
        {
                printLine(argv[1]);
        }
        return 0;
}
```

Figure 37. SAMATE test case 1897 of test suite 45.

Figure 38 shows how Fortify SCA find a path manipulation vulnerability in the test case 1897 (resource_injection_basic.c) in line 21. The tool shows the analysis evidence in the left frame and other information about examples and how an auditor can remediates the vulnerability.

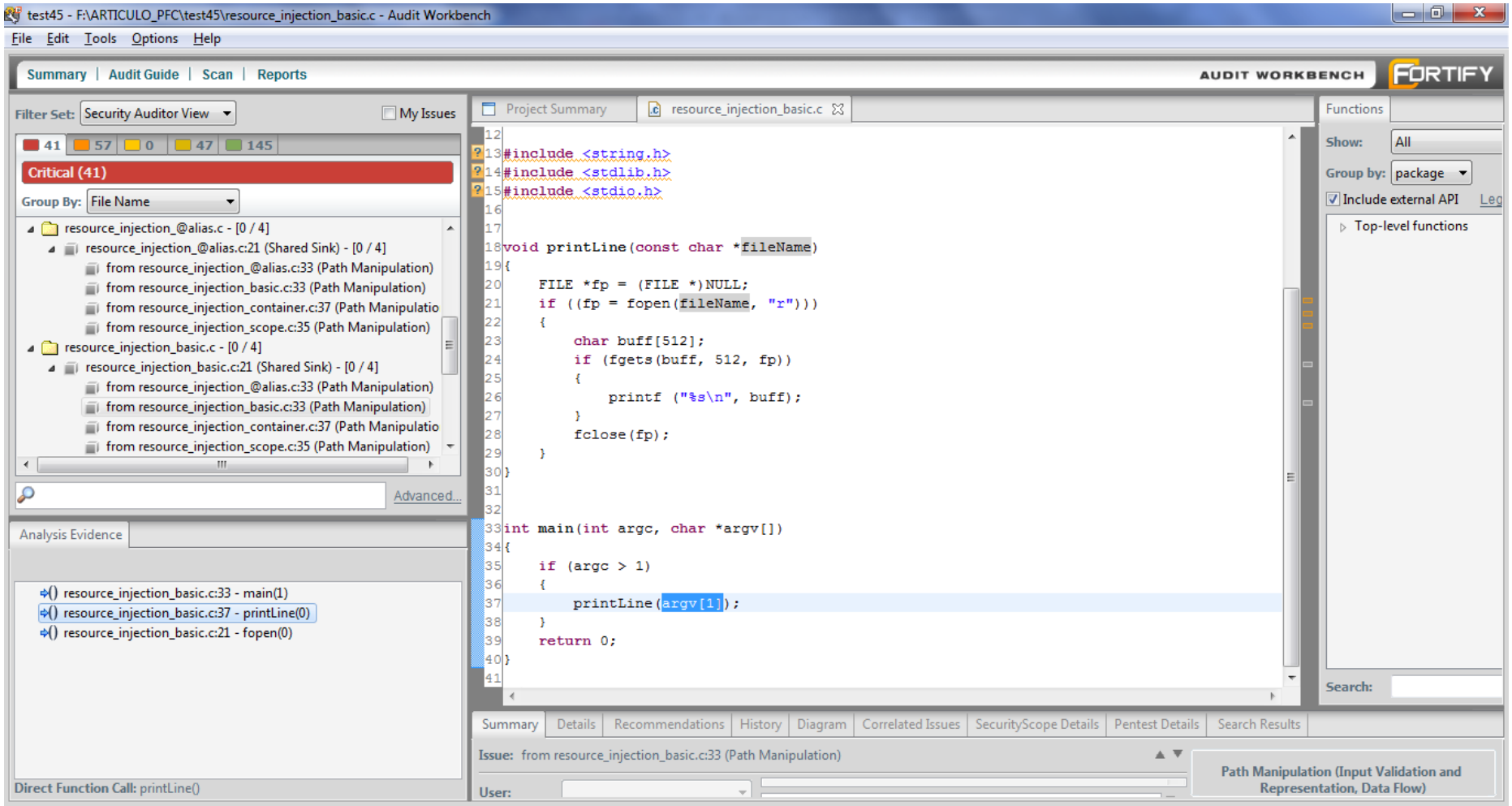

Figure 38. Analysis of SAMATE test case 1897 of test suite 45 with Fortify SCA.

For security tools evaluation process two distinct benchmarks approachs can be used mainly:

1. **Test suites collections** composed of test cases for specific languages. There are test suites for detect false positives and test suite for vulnerability detection (true positive).
2. **Application benchmarks** with specific vulnerability design to perform a security analysis for vulnerability detection.

Next we try to reflect the state of art of benchmarking taking into account the described previous characteristics .

4.8.1. TEST SUITES COLLECTIONS BENCHMARKS.

SAMATE REFERENCE DATASET (SRD). [Samate, 2013]. It provides a wide number of tests suites for different languages, designed to check if a tool detect some (or all) of these vulnerabilities. If, for example, the objective is to know if a static tool detects a particular software vulnerability we must select the corresponding test. Then, the static tool is executed against the code in the test and we can see if the tool really detects it or not. SAMATE SRD provides also some ‘‘test-suites’’ that cover many (or all) of the selected vulnerabilities. SAMATE SRD makes all possible efforts to become a benchmark reference and its use is recommendable for sharing the information obtained when applied to static analysis tools. Although not so detailed as our study, these tests have been used before as a reference. Cifuentes and Scholz in [Cifuentes, 2008] used a subset of the SAMATE tests, only those that are for C code and that relate to buffer overflow and ‘‘read outside the bounds of an array’’, for evaluating the design of the Parfait tool. Some SAMATE tests have been used also to perform a study to understand the coverage of the security rules in the Motorola coding standards [Krishnan, 2008]. The NIST SAMATE project conducted

also the third Static Analysis Tool Exposition NIST SP 500-297 [NIST297, 2012] in 2012 to advance research in static analysis tools that find vulnerabilities in source code. The main goals of SATE were to enable empirical research based on test sets, encourage improvements to tools, and promote broader and more rapidly adoption of tools by objectively demonstrating their use on production software. The master thesis of Jayesh Shrestha [Shrestha, 2013] used SAMATE SRD to evaluate several static analyzers security tools.

SECUREBENCH MICRO. [Securebench, 2103] Securebench Micro is a series of small test cases designed to exercise different parts of a static security analyzer (SAST). Each test case in Securebench Micro comes with an answer, which simplifies the comparison process. All test cases included in this release can be installed on a standard application server such as Apache Tomcat and be used to compare the effectiveness of runtime techniques such as penetration testing tools (DAST) or IAST. This test suite has been used by Benjamin Livshits and Monica S. in their work "*Finding Security Vulnerabilities in Java Applications with Static Analysis"* [Livshits, 2005] and Benjamin Livshits in "*Improving software security with precise static and runtime analysis"* [Livshits, 2006].

U.S. DEPARMENT of HOMELAND SECURITY [Homeland, 2014]. It includes 23 test samples designed specifically for SCA tools testing. These example programs demonstrate flaws that may be detected by security scanners for C/C++ software. The examples are small, simple C/C++ programs, each of which is meant to evaluate some specific aspect of a security scanner's performance.

WAVSEP PROJECT [Wavsep, 2104].This evaluation platform contains a collection of unique vulnerable web pages in J2EE technology that can be used to test the various properties of web application scanners (DAST) or static analyzers (SAST). WAVSEP has

been used in DAST evaluation performed by SECTOOLMARKET benchmarking tool evaluation project [SEC, 2012]

4.8.2. APPLICATION BENCHMARKS.

Apart from test suites benchmarks there are insecure application benchmarks designed with known vulnerabilities to perform security tool assessments. Assessment of false positives and true positives can be performed with these types of applications. This section presents deliberately insecure applications benchmarks for using in web application assessments:

MOTH (http://www.bonsai-sec.com/en/research/moth.php) is a VMware image with a set of vulnerable Web Applications and scripts, that people may use for:

1. Testing Web Application Security Scanners
2. Testing Static Code Analysis tools (SCA)
3. Giving an introductory course to Web Application Security

The main objective of this tool is to give the community a testbed for web application security tools. It is possible to find a test script available in MOTH for almost every web application vulnerability that exists in the wild.

NOWASP (MULTIDAE) (http://www.irongeek.com/i.php?page=mutillidae/mutillidae-deliberately-vulnerable-php-owasp-top-10) is a free, open source web application provided to allow security enthusiast to pen-test and hack a web application. NOWASP (Mutillidae) can be installed on Linux, Windows XP, and Windows 7 using XAMMP making it easy for users who do not want to install or administrate their own webserver. It is already installed on Samurai WTF. Simply replace existing version with latest on Samurai. NOWASP contains dozens of vulnerabilities and hints to help the user exploit them; providing an easy-to-use web hacking environment deliberately designed to be used as a hack-lab for security

enthusiast, classroom labs, and vulnerability assessment tool targets. NOWASP has been used in graduate security courses, in corporate web sec training courses, and as an "assess the assessor" target for vulnerability software.

NOWASP has been tested/attacked with Cenzic Hailstorm ARC, W3AF, SQLMAP, Samurai WTF, Backtrack, HP Web Inspect, Burp-Suite, NetSparker Community Edition, and other tools.

HACME SERIES FROM FOUNDSTONE (http://www.mcafee.com/us/downloads/free-tools/index.aspx). **Foundstone** has put out a whole series of venerable web applications practitioners can learn from and test your skills against:

- Hacme Travel (http://www.foundstone.com/us/resources/proddesc/hacmetravel.htm)
  Platform: Windows XP, Microsoft .NET Framework v1.1, C++
- Hacme Bank (http://www.foundstone.com/us/resources/proddesc/hacmebank.htm)
  Platform: Windows, IIS, .Net 1.1:
- Hacme Shipping
  (http://www.foundstone.com/us/resources/proddesc/hacmeshipping.htm)
  Platform: Windows XP, Microsoft IIS, Adobe ColdFusion MX Server 7.0 for Windows, MySQL (4.x or 5.x with strict mode disabled)
- Hacme Casino (http://www.foundstone.com/us/resources/proddesc/hacmecasino.htm)
  Platform: Ruby on Rails
- Hacme Books (http://www.foundstone.com/us/resources/proddesc/hacmebooks.htm)
  Platform: J2EE application, Java Development Kit

WIVET (https://code.google.com/p/wivet/) is a benchmarking project that aims to statistically analyze web link extractors. In general, web application vulnerability scanners fall into this category. These web application vulnerability scanners, given a URL(s), try to

extract as many input vectors as possibly they can to increase the coverage of the attack surface. WIVET provides a good sum of input vectors to any extractor and presents the results. In order an input extractor to run meaningfully, it has to provide some kind of session handling, which nearly all of the decent crawlers do. It has been used in DAST evaluation performed by SECTOOLMARKET benchmarking tool evaluation project [SEC, 2011].

4.9. CONCLUSIONS.

This chapter has reviewed the main Secure Software Development Life Cycle implementations to find out how and when (phase) they use automatic security tools. It also has described the main projects about security standards, about security tool assessments, benchmarks and evaluation methodologies. The state of the art knowledge of security tools and benchmarks is necessary to select the most adequate to accomplish a security tool assessment. Also, the state of the art of assessment methodologies, as for example SAMATE, is the starting point to derive a new and repeatable methodology (see chapter 5), one of the original contributions of this thesis, used in the security tool evaluations.

The main characteristics of all types of automatic or semi-automatic security tools have been reviewed:

- SAST (Static Analysis Security Tools)
- DAST (Dynamic Analysis Security Tools)
- IAST(RAST) (Interactive-real Analysis Security Tools)
- HYBRID tools of the some previous types.

5. ASSESSMENT OF SECURITY ANALYSIS TOOLS

5.1. INTRODUCTION.

This chapter is dedicated to evaluate the type of semi-automated security tools revised in previous sections to study their relative performance in terms of detection rates, false positive rates and vulnerability coverage degree metrics. The evaluation process follows a specific and repeatable methodology to allow getting the objectives proposed with a defined procedure. This procedure consists in using selected benchmarks to run the selected tools against them and analyze the results with a set of widely acceptable metrics. Each evaluation process will permit to compare the performance of the analyzed security tools. The following sections address the evaluation process of the security tools categories considered in this thesis to perform security analysis of applications:

- Section 5.2: Methodology used to accomplish the assessments of security tools categories considered in this work.
- Section 5.3: Set of metrics used for the analysis of the results of the assessments.
- Section 5.4: SAST assessment for C/C++ applications.
- Section 5.5: SAST assessment for web applications.
- Section 5.6: DAST-IAST-HYBRID assessment for web applications.

5.2. ASSESSMENT METHODOLOGY.

Section 4.7 was a survey of main methodologies available to accomplish assessment processes or security tools as SAST, IAST or DAST. Based on that previous work, we derived the proposed methodology.

The methodology process, shown in figure 39, consists on:

1. Benchmark selection. The benchmark state of art is examined (see section 4.8) to select the most adequate to execute the tools against it. The benchmarks can be test suites or application benchmarks. A benchmark is selected for each tool category assessment (details below in this chapter).
2. Metrics selection. With the metrics selected we analyze the results to rank the tools strictly. The metrics selected are widely accepted and explained in the following section 5.3.
3. Tools selection. SAST, DAST, IAST and HYBRID tools state of art is revised (see sections 4.3, 4.4, 4.5, 4.6) to select the most adequate for each type of assessment performed.
4. Test execution: run tools against benchmarks. This execution provided a first set of results that must be analyzed for each case and for each tool, using its trace help for warnings (vulnerability detected), vulnerability documentation and background references.
5. Computing the evaluation metrics for each tool executed against benchmark test suites.
6. Analyzing the results for each for each tool using the computed metrics.

The analysis of these results allows getting the performance of the tools related with the different accuracy for finding vulnerabilities and, as a consequence, also the false negative and false positive ratios associated with each tool. The performance degree of tools allows ranking them strictly.

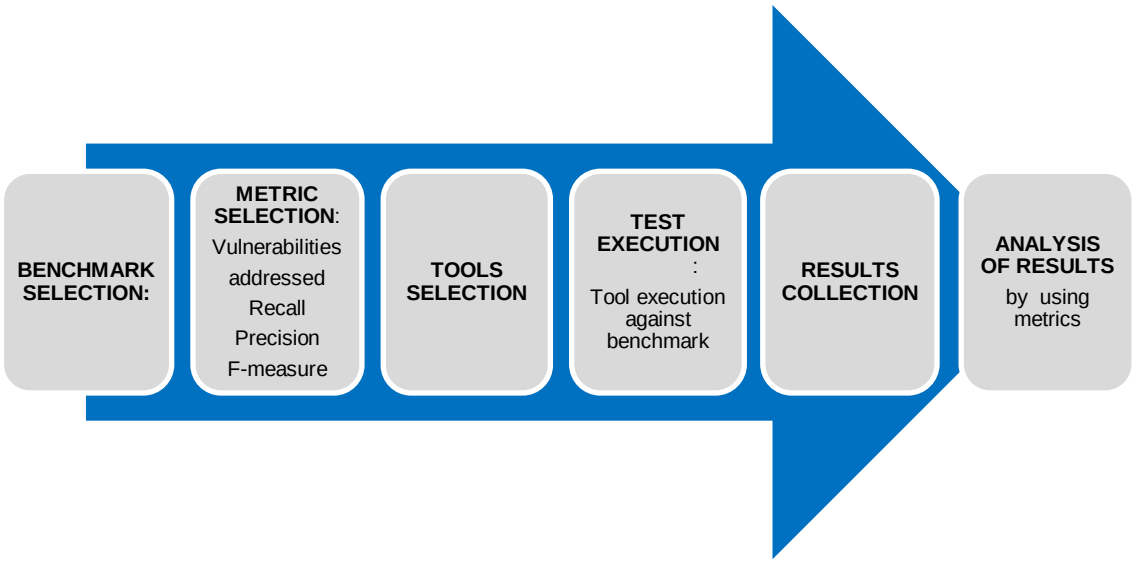

Figure 39. Methodology process.

5.3. EVALUATION METRICS.

The evaluation metrics considered in this section will be used through all assessment processes accomplished in the following sections.

The first metric to compute in relation to SAMATE test suites is the **vulnerabilities coverage percentage** for each tool. A tool should be able of detecting at least the vulnerabilities defined for each assessment process, according to the most dangerous vulnerabilities published in standards as OWASP TOP TEN, SANS TOP 25 or SAMATE project publications.

In previous section 4.1 the problems that static analysis tools suffer in relation to false positives and negatives were exposed. The measurement of false positives and negatives must be computed from the information collected after execution of each tool against the benchmark test suites. The goal is to characterize vulnerability detection tools using the **F-Measure** as shown in *Information Retrieval* [Rijsbergen, 1979], which is largely independent of the way vulnerabilities are counted. In fact, it represents the harmonic mean

of two measures (precision and recall), which, in the context of vulnerability detection, can be defined as:

- **Precision**: the ratio of correctly detected vulnerabilities to the number of all detected vulnerabilities. Precision is also referred to as Positive predictive value (PPV):

$$\frac{TP}{TP+FP}$$

- **Recall**: a ratio of correctly detected vulnerabilities to the number of known vulnerabilities. Recall in this context is also referred to as the True Positive Rate or Sensitivity:

$$\frac{TP}{TP+FN}$$

Where:

- TP (true positives) is the number of true vulnerabilities detected (i.e., vulnerabilities that, in fact, exist in the code);
- FP (false positives) is the number of vulnerabilities detected that, in fact, do not exist.
- FN (false negatives) is the total number of vulnerabilities not detected in the code.

- **Harmonic mean** is:

$$\frac{n}{1/X1+\cdots+1/Xn}$$

Where:

- *n*: number of variables.
- $x_{n:}$ value of variable n.

The formula for **F-Measure** is harmonic mean of *precision* and *recall*:

$$\frac{2\ x\ precision\ x\ recall}{precision+\ recall}$$

Two other commonly used F-measures are the F_2-measure, which weights recall higher than precision and the $F_{0,5}$-Measure, which puts more emphasis on precision than recall. The formula for F_β-Measure is:

$$(1+\beta^2)x\frac{precision\ x\ recall}{\beta^2\ x\ Precision+Recall}$$

F-Measures will have ranges between 0 and 1 for a tool. A tool with 65% precision means that a given warning has a 65% chance of being correct. A recall of 0.8 expresses that 80% of all the known vulnerabilities are detected and that 20% are missed. In this case the F-Measure is approximately 0.717. The three measures can be used to establish a ranking of the performance of several tools depending on the purposes of the benchmark user.

5.4. SAST ASSESSMENT IN C-C++ APPLICATIONS.

Following the methodology exposed in section 5.2, we executed the selected tools (described in following subsection 5.4.2) against a representative repeatable, portable and scalable benchmark, described in subsection 5.4.1. The results are analyzed with widely accepted metrics of section 5.3, to extract conclusions of their performance and recommendations to their future using and improving.

Table 20, extracted from NIST SP 500-268 [NIST268, 2007] of SAMATE project, shows the minimal set of vulnerabilities that a static security tool should be able to detect in C/C++ code.

Table 20

Set of source code vulnerabilities [NIST268, 2007]

NAME	CWE ID	Description
Input validation		
Basic XSS (Cross-Site Scripting)	80	Unfiltered input is passed to a web application that in turn passes that data back to another client in the form of a malicious script

Resource Injection	99	Unfiltered input is used in an argument to a resource operation function.
OS Command injection	78	Unfiltered input is used in an argument to a system operation execution function.
SQL Injection	89	Unfiltered input is used in an argument to a SQL command calling function.
Range Errors		
Stack overflow	121	Input is used in an argument to the creation or copying of blocks of data beyond the fixed memory boundary of a buffer on the stack.
Heap overflow	122	Input is used in an argument to the creation or copying of blocks of data beyond the fixed memory boundary of a buffer in the heap portion of memory.
Format string vulnerability	134	Unfiltered input is used in a string used to format data in the printf() style of C/C++ functions.
Improper null termination	170	The software does not properly terminate a string or array with a null character or equivalent terminator
API Abuse		
Heap Inspection	244	Using realloc() to resize buffers that store sensitive information can leave the sensitive information exposed to attack because it is not removed from memory.
String management	251	Some string manipulation functions can be exploited through their input to produce buffer overflows.
Security features		
Hard-coded password	259	Hard-coded data is passed as an argument to a login function.
Time and state		
Time-of-check Time-of-use race condition (TOC_TOU)	367	Between the time in which a given resource (or its reference) is checked, and the time that resource is used, a change occurs in the resource to invalidate the results of the check.
Unchecked Error Condition	391	No action is taken after an error or exception condition occurs.
Code quality		
Memory leak	401	Memory is allocated, but is not released after it has been used.
Unrestricted Critical Resource Lock	412	A resource is “deadlocked” by obtaining an exclusive lock or mutex, or modifying the permissions of a shared resource.
Double Free	415	An attempt is made to free memory using an address that has previously been used in a free () function call.

Use After Free	416	An attempt is made to access the same memory address previously released by a call to the free() function.
Uninitialized variable	457	A variable is created without assigning it a value. It is subsequently referenced in the program, causing potential undefined behavior or denial or service.
Unintentional pointer scaling	468	Improper mixing of pointer types in an expression may result in references to memory beyond that intended by the program.
Null Dereference	476	A pointer with a value of NULL is used as though it pointed to a valid memory area.
Encapsulation		
Leftover Debug Code	489	Debug code can create unintended entry points in an application.

5.4.1. BENCHMARK SELECTION.

The static analysis tools selected were executed against SAMATE Reference Dataset [Samate, 2013] test suites 45 and 46 for C language. Test suite 45 includes test cases with known vulnerabilities and test suite 46 is designed with specific vulnerabilities fixed. Each test case is relative to a specific *complexity*, a concept referred to the way of storing a memory variable [Kratkiewicz, 2005]. Each different *code complexity* type, such as fixed or variable loops, memory indexing nested within indexing, local vs. global scope, and others, may require additional analytical capabilities. This benchmark meets all the requirements that a benchmark must have according to the section 4.8.

We did a detailed review process of the validity of all tests in SAMATE test suites 45 and 46. Thanks to this process, we have found eight non valid test cases in test suite 46. Four of them correspond to “os command injection” test cases and the other four correspond to “resource injection” test cases.

The study of test case 1931 is particularly relevant. This case is related with a command injection CWE, but corrected by using a function (*purify()*) that, theoretically, validates the variable storing the command. The truth is that the function in this test is insufficient to prevent command injection, not checking important characters like pipes (|), backquotes (‘),

the dollar sign ($), i/o redirection (<, >), conditional shell operators (&&, ||) and others. However, after executing all the tools against the modified test (including all these characters) the results are almost identical. The only difference is with *SCA*, but it is a minor change. With the corrected *purify()* function, *SCA* marks a "WARNING" (detecting a vulnerability), instead of "HOT", the old result, when executed against the SAMATE test without corrections. So the results of the study remain the same, although this test case and the three related ones must be changed.

The four "resource injection" test cases exhibit an error in the return of the *allowed()* function: the two "return" sentences must be interchanged, see test cases 1896, 1898, 1900, 1902. However, the results of tools execution against tests, after fixing the four errors, were the same for all tools.

5.4.2. SAST SELECTION.

The tools selected should allow comparing their performance, their usability and the number and range of covered vulnerabilities for relevant commercial and open source tools. From all the analyzed tools, two open source solutions and seven leading commercial solutions were selected.

Commercial tools have advantages as:

- Support for more languages,
- Larger vulnerabilities coverage when compared with table 20,
- Better usability and trace help for discarding false positives.
- They are the best candidates for being included in a process of code security review in a company.

These are the main reasons for the authors to select seven of most important commercial tools to be included in the assessment.

Table 21 shows the reviewed commercial tools and the considerations observed to select the tools for the assessment.

Table 21.
Analyzed commercial static analysis tools. [Díaz 2013]

TOOLS	CHARACTERISTICS AND CONSIDERATIONS
SCA (HP FORTIFY)	Leader security review tool. 100% coverage of table 1 vulnerabilities categories. It covers 18 different languages
APPSCAN SOURCE EDITION (IBM)	Leader security review tool. Large coverage of languages and vulnerabilities categories. It was not possible getting it for evaluation, no response received.
K8-INSIGHT (KLOCWORK)	Bug finding tool for Java, J2EE, C and C#. 82.8% coverage of table 20 vulnerabilities categories
PREVENT (COVERITY)	Bug finding tool for Java, C and C#. 92.4% coverage of table 20 vulnerabilities
GOANNA (RED LIZARD)	Bug finding tool for C and C++. Without injection vulnerabilities coverage
PC-LINT (GIMPEL)	Tool for C and C++. Without injection vulnerabilities coverage.
CHECKMARX CX-ENTERPRISE (CHECKMARX)	Bug finding tool. 91.4% coverage of table 20 vulnerabilities. It covers 15 different languages.
CODESONAR (GRAMMATECH)	Program verification tool for C/C++ and Java, It does not check for the most severe vulnerabilities, such as SQL injection and cross-site scripting.
POLYSPACE (MATHWORKS)	Program verification tool for ADA, C/ C++. It proves the absence of overflow, divide-by-zero, out-of-bounds array access, and run-time errors. It was not possible getting it for evaluation, no response received.
C++TEST (PARASOFT)	Security and Quality analysis tool C, C++, Java, C#, and VB.NET. It focuses more in quality than security

Finally, the selected commercial tools were:

SCA version 4 is a product of Fortify Software [HP-Fortify, 2013], now a Hewlett-Packard Company. *SCA* is a tool of the "security review" type. It uses lexical, syntactic and semantic analysis, control flow and data flow analysis. It builds an intermediate model of the code on which several specialized analyzers run, using many different security rules. *SCA* covers C/C++, C#, ASP NET, VB.NET, COBOL, CFML, HTML, Java, JavaScript,

AJAX, JSP, PHP, PL/SQL, Python, Visual Basic, VBScript and XML. Fortify claims that *SCA* provides details for more than 450 categories of vulnerability.

Prevent version 3.8 is a product of Coverity [Coverity, 2013]. *Prevent* is a tool of the "bug finding" type. *Prevent* is able to analyze C/C++, Java and C# code.

K8-Insight version 8 is a product of Klocwork [Klocwork, 2013]. *K8-Insight* is another leading "bug finding" tool that covers C/C++, Java, J2EE and C #code.

PC-lint version 8.00n is a Gimpel Software product [Gimpel, 2103]. *Pc-lint* is a style checking tool. Their web site states the tool detects "vulnerabilities, glitches, inconsistencies, non-portable constructs, redundant code for C/C++ programs".

Goanna release 2.9.0-11916, a Red Lizard Software product [RedLizard, 2013]. *Goanna* is a "bug finding" tool. It uses model checking techniques to keep false positives as low as possible in C code. Its engine uses abstract interpretation algorithms.

Cx-enterprise version 6.20 is a product of Checkmarx [Checkmarx, 2013]. *Cx-enterprise* is another bug finding tool for Java, C# / .NET, PHP, C, C++, Visual Basic 6.0, VB.NET, Flash, APEX, Ruby, Javascript, ASP, Android and Perl languages with 92,4% of table 1 vulnerabilities coverage.

Codesonar *version* 3.7 is a Grammatech product [Grammatech, 2013]. *Codesonar* is a program verification tool for C/C++ and Java (announced for next 3.8 version) languages, which performs a unified dataflow and symbolic execution.

The other two tools we included in the assessment are open source tools. These are a set of common characteristics about open source tools:

- Generally almost all open source tools are research projects from Universities, or in some cases from companies,

- Usually their vulnerabilities coverage is limited to a short subset of those in table 20.
- Their usability, human interfaces and warning trace capabilities are much more reduced than commercial tools.
- Some tools can require code annotations to enhance the results, making them not useful for analysis of projects with several hundreds of thousands or millions of lines of code.

Table 22 shows the reviewed open source tools and the considerations observed to select the tools for the assessment.

Table 22
Analyzed open source static analysis tools [Díaz, 2013]

TOOLS	CHARACTERISTICS AND CONSIDERATIONS
UNO, RATS, FLAWFINDER, ITS4, LINT	Earlier tools limited to lexical-syntactic analysis and only for a reduced subset of vulnerabilities. All of them preprocess and tokenize source files (the same first steps a compiler take) and then match the resulting token stream against a library of vulnerable constructs.
BOON	Applies integer range analysis. It can't model interprocedural dependencies, and it ignores pointer aliasing
CQUAL	Type-based analysis, requires annotations in the code
BLAST	Model checking tool, with the option of adding assertions in the code
SPLINT	Enhanced version of Lint. Requires annotations in the code
SATURN	Boolean satisfiability and summary based tool. Only limited to memory leaks, lock problems and null dereferences vulnerabilities.
BOOP	Abstraction and model checking tool. Not maintained anymore. The formalization of C expressions is incomplete and not all C constructs are covered.
SATABS	Program verification tool with Model checking, that implements a predicate abstraction refinement loop using a SAT-solver. This allows the model checker to handle the semantics of the ANSI-C standard accurately.
CBMC	Program verification tool with Bounded Model Checking new tool research. In CBMC, the transition relation for a complex state machine and its specification are jointly unwound to obtain a Boolean formula, which is then checked for satisfiability by using a SAT procedure
MAGIC	Bounded Model Checking tool that require specifications in the code to accomplish an analysis

Finally, the selected open source tools were:

Satabs [Clarke, 2005] is a program verification tool for ANSI-C programs. It allows verifying array bounds (buffer overflows), pointer safety, exceptions and control-flow oriented user-specified assertions.

CBMC (*C Bounded Model Checking*) [Clarke, 2004] is a program verification tool designed for ANSI C, it allows verifying array bounds (buffer overflows), pointer safety, exceptions and user-specified assertions.

5.4.3. EXECUTION RESULTS.

The detailed execution results for test suites 45 y 46 are included, in tables 42 and 43 respectively, in Appendix B. Table 42 shows the percentage of detections for each vulnerability category and tool in test suite 45. In last row detection percentage mean for each tool is calculated. Table 43 shows the percentage of false positives for each vulnerability category and tool in test suite 46. In last row false positive percentage mean for each tool is calculated.

Table 23 summarizes the execution results of the security static tools against the 78 tests in SAMATE test suite 45. "Fails" (false negatives) means that the tool should have detected the specific vulnerability, because the tool was designed to detect it, but the tool did not do so. "Good" (true positives) means that the tool has detected the vulnerability. "Vulnerabilities not covered" means that the tool has not been designed to detect the specific vulnerability. The absolute detection percentages, for each tool, are calculated by excluding these cases each tool cannot detect ("vulnerabilities not covered"), and then dividing the number of detected vulnerabilities by the number of vulnerabilities that the tool is theoretically designed to detect. To normalize the result of detections, we computed the percentage of detections for each type of vulnerability (recall) that each tool is designed for

detecting. Last column of table 23 shows also the *arithmetic mean* of detection percentage for all types of vulnerabilities that each tool is able of detecting.

Table 23

Executions results for SAMATE test suite 45 [Díaz, 2013]

Test suite 45 results 78 cases	FAILS (FN)	GOOD (TP)	Vulnerabilities not covered by a tool	ABSOLUTE DETECTION % OF VULNERABLITIES COVERED	DETECTION PERCENTAGE MEAN
SCA	18	60	0	76.9	**66.5**
PREVENT	15	50	13	76.9	**65.6**
CX-ENTERPRISE	26	46	7	64.7	**57.4**
CODESONAR	21	34	23	61.8	**55.6**
K8-INSIGHT	23	42	13	64.6	**52.7**
GOANNA	24	29	25	54.7	**49.4**
CBMC	19	29	30	60.4	**46**
SATABS	20	28	30	58.3	**43.6**
PC-LINT	25	28	25	52.8	**37.6**

None of the tools detect all the SAMATE vulnerabilities in test suite 45. Indeed only *SCA* is theoretically designed for detecting all of them.

Seven important vulnerabilities are not detected by any tool: the "unrestricted critical resource lock" vulnerability, a dangerous vulnerability related with many kinds of DOS (*Denial of Service*) attacks and well documented (CWE ID 412), the "incorrect pointer scaling" (CWE ID 468) vulnerability and the five vulnerabilities of "basic XSS" cross-site scripting (CWE ID 80). Only *SCA* and *Codesonar* detects one of three "Time-of-check, Time-of-use race condition (TOCTOU)" (CWE ID 367), but only at the basic complexity level. Only *Cx-enterprise* detects one of five possible vulnerabilities of "hard-coded password" (CWE ID 259) at the basic complexity level. This behavior is an example of how the detection depends on the level of code complexity of the test case.

Table 24 shows the summary of results for the execution of the nine tools against the 74 specific cases of SAMATE test-suite 46. The term "false positives" indicates the number of false positives found and "good" (true negatives) means that the tool detected nothing, being this the expected behavior of a good tool. The absolute percentages of false positives,

for each tool, are calculated by excluding again the cases that each tool cannot detect ("vulnerabilities not covered") and then dividing the number of false positives by the number of vulnerabilities the tool can detect.

Table 24
Executions results for SAMATE test suite 46 [Díaz, 2013]

Test suite 46 results 74 cases	FALSE POSITIVES (FP)	GOOD (TN)	Vulnerabilities not covered by a tool	% ABSOLUTE FALSE POSITIVES	FP PERCENTAGE MEAN
PREVENT	4	57	13	6.5	**5.3**
K8-INSIGHT	7	54	13	11.4	**9,5**
CX-ENTERPRISE	15	53	6	22	**21.9**
GOANNA	8	43	23	15.6	**22.6**
CODESONAR	8	44	22	15.3	**24.7**
PC-LINT	18	33	23	35.2	**29.2**
SCA	27	47	0	36.4	**32.1**
SATABS	24	22	28	52.1	**36.6**
CBMC	29	17	28	63	**43.7**

To normalize the results authors calculate the percentage of false positives for each type of vulnerability that each tool is designed for detecting. Last column of table 24 shows the *arithmetic mean* of false positives percentage of all vulnerabilities types that each tool is able to detect.

5.4.4. ASSESSMENT RESULTS.

Figure 40 shows the summary of the types and numbers of not covered vulnerabilities by each tool for test suite 45 (summary for test suite 46 is similar) in comparison with vulnerabilities of table 20. SCA is the only tool design to detect all vulnerabilities categories of table 20.

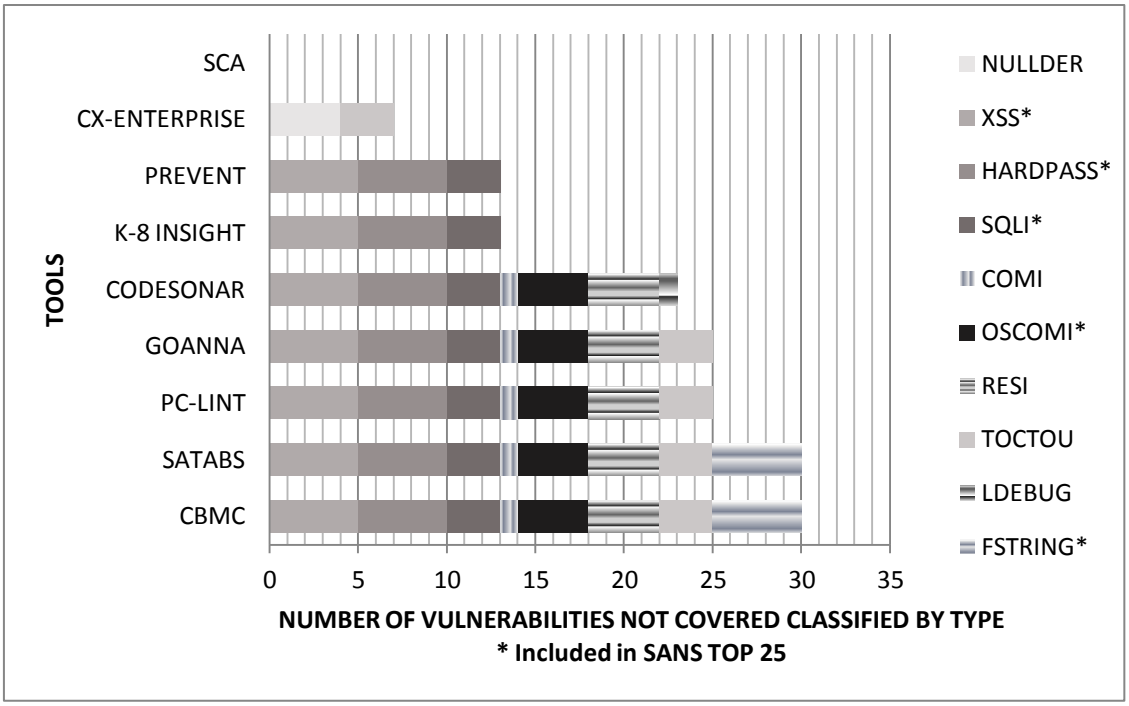

Figure 40. Vulnerabilities types not covered by tools for test suite 45. [Díaz, 2013]

Table 25 shows the results for the different F-measure metrics, applied to absolute precision and recall metrics:

Table 25

Metrics applied to test suites absolute results [Díaz, 2013]

METRICS APPLIED TO RESULTS	% Vul. COVERED	TP	FP	PRECISION	RECALL	F0.5-MEASURE	F2-MEASURE	F-MEASURE
PREVENT	**82.8**	50	4	0.925	0.769	0.888	0.795	**0.839**
K8-INSIGHT	**82.8**	42	7	0.857	0.646	0.804	0.679	**0.736**
SCA	**100**	60	27	0.689	0.769	0.703	0.751	**0.726**
CODESONAR	**70.3**	34	8	0.809	0.618	0.761	0.648	**0.700**
CHECKMARX	**91.4**	46	15	0.754	0.633	0.726	0.653	**0.686**
GOANNA	**68.4**	29	8	0.783	0.547	0.720	0.582	**0.644**
PC-LINT	**68.4**	28	18	0.608	0.528	0.590	0.542	**0.565**
SATABS	**61.8**	28	24	0.538	0.583	0.546	0.573	**0.559**
CBMC	**61.8**	29	29	0.500	0.604	0.517	0.579	**0.547**

Table 26 shows the results for the different F-measure metrics applied to weighted precision and recall metrics:

- *Recall mean* is calculated as the arithmetic mean of all vulnerabilities types recall got by each tool.
- *TP mean* is a value obtained from recall mean. For example, for *Prevent*, *TP mean* is:

  recall mean X vulnerabilities covered = 0.656 x 65
- *FP mean* is a value obtained from % FP mean. For example, for *Prevent*, *FP mean* is:

  % FP mean X vulnerabilities covered / 100 = 5.3 x 61 / 100

Table 26
Metrics applied to test suites weighted results [Díaz, 2013]

METRICS APPLIED TO RESULTS	% Vul. COVERED	TP (mean)	FP (mean)	PRECISION	RECALL (mean)	F0.5-MEASURE	F2-MEASURE	F-MEASURE
PREVENT	**82.8**	42.6	3.2	0,930	0.656	0,858	0,697	**0,769**
SCA	**100**	51.8	23.7	0.685	0.665	0.680	0.668	**0.674**
K8-INSIGHT	**82.8**	34.2	5.7	0,857	0.527	0,761	0,570	**0,652**
CHECKMARX	**91.4**	40.7	14.8	0.732	0.574	0.693	0.599	**0.643**
CODESONAR	**70.3**	30.5	12.8	0,704	0.556	0,668	0,580	**0,621**
GOANNA	**68.4**	26.1	11.5	0.694	0.494	0.642	0.524	**0.579**
SATABS	**61.8**	20.9	14.8	0.554	0.436	0.525	0.455	**0.487**
CBMC	**61.8**	22	20.1	0.523	0.460	0.509	0.471	**0.481**
PC-LINT	**68.4**	19.9	11.5	0.621	0.376	0.549	0.408	**0.469**

F-measure metric allows obtaining a strict ranking of the analyzed tools' performance.

Prevent obtained the best result for F-measure, indicating a very good balance between false and true positives (65.6% of detections and 5.3% of false positives). *SCA*, *K8-insight*, *Cx-enterprise* and *Codesonar* obtained a similar result of F-measure. *Pc-lint* obtained the worst result score of 0.469. F-measure metric allows obtaining a strict ranking of the analyzed tools' performance.

5.4.5. CONCLUSIONS.

The main conclusions of this section are:

1. The methodology applies widely known metrics based on rates of true and false positives and vulnerabilities coverage degree of tools, producing a strict scale for the performance of static analysis tools. Then, a company can choose a tool by analyzing the precision, recall, F-measure and vulnerabilities coverage metrics obtained against SAMATE test suites.
2. Commercial tools (as *Prevent, SCA, K8-Insight, Cx-enterprise and Codesonar*) show a better performance, usability and vulnerabilities coverage than the other analyzed tools. However, all the analyzed tools obtain different results for different types of vulnerabilities and cover different subsets of them.
3. Only one tool (SCA) covers all vulnerability categories in test suites 45 and 46 (see figure 40). The other tools do not cover important vulnerability categories as XSS, SQLI, CMDI, OSCMDI or Hardcoded password. Besides none tool detects some vulnerability categories though they are designed to detect it. *Unintentional pointer scaling* and *unrestricted critical resource lock* (see table 42 in appendix B) are two vulnerabilities not detected by any tool.
4. A simple execution of many of these tools against a piece of code is not enough to get reliable results, and raw results (results from the first execution) must be reviewed. Of course the automatic execution of tools gives a formidable first step, especially for analyzing lengthy code, but this is not enough. A careful analysis of the results by an experienced user or team (with security skills and experience in the language used in the target code) is always necessary.
5. The use of tools for static source code analysis to search security vulnerabilities must be integrated as a part of the security policy of any development organization. But current state of these tools does not allow indistinctly using them. The tools' internal designs and reporting output formats are different, so they produce substantially different results.

Some recommendations easy to infer are:

- The need of standard output formats.
- Improving true and false positive rates and their balance is yet important.
- Take into account the time needed for performing the report audit of a static tool.
- Using several tools with different designs and with different detection algorithms/heuristics to improve the analysis results when making a real analysis of a big project
- Promote the use of SAMATE tests as a benchmark for objectively evaluating and comparing the performance of static source code security analyzers

5.5. SAST ASSESSMENT IN WEB APPLICATIONS.

As commented in section 2.2.3, the volume of web applications developed exceeds 55% of total developed applications. Web applications account for 75% of analyzed software. Therefore, the number of threats that web applications may suffer is quite high and about 47% of all disclosed vulnerabilities were in web applications [IBM, 2012], according to IBM X-Force 2012 Trend and Risk Report. This forces to make a security analysis of any web application to avoid as many threats as possible.

Regarding to SAST, the range of solutions in this category must be analyzed to find out which solutions are most appropriate to test the capacity of detecting the most important and frequent security vulnerabilities. The tools' performance includes:

- Calculating the average of true positive and false negatives and other metrics detailed in section 5.3.
- Examining SAST aid to eliminate false positives and the possibility for the user of making new detectors or rules for detecting other vulnerabilities.
- Studying how the tools can be combined to detect more vulnerabilities showing that it is possible reaching rates of 95% of detections.

In this section we present the results of a comparative assessment between six SAST commercial and open source tools for source and executable code (see section 5.2.2). SAST tools are selected for J2EE technology, the most used technology in web developing [Veracode. 2012]. Four *source code static analysis tools* are selected according to SAST state of art for web applications (see section 4.4.3) and related work (see section 7.2) *: K8 Insight, Lapse+, Fortify Sca, Cx-enterprise* and two *binary static analysis tools: Veracode and Findbugs*.

The methodology used in the assessment is the one exposed in section 5.2. We executed the selected tools against a representative repeatable, portable and scalable benchmark. The results are analyzed with the metrics of section 5.3, to extract conclusions of their performance and recommendations to their future using and improving.

Table 27, shows the most dangerous security vulnerabilities in web applications, according to SAMATE-NIST *webapp scanner specification* SP 500-269 document [NIST269, 2008], their specification document to evaluate SAST, DAST, IAST or HYBRID tools.

Table 27
Most dangerous security vulnerabilities in web applications [NIST269, 2008]

Name	Description	Related terms	CWE
Cross Site Scripting (XSS)	A web application accepts user input (such as client-side scripts and hyperlinks to an attacker's site) and displays it within its generated web pages without proper validation.	Reflected XSS, persistent (stored) XSS, DOM-based XSS	79
SQL Injection	Unvalidated input is used in construction of an SQL statement.	Blind SQL injection	89
OS Command Injection	Unvalidated input is used in an argument to a system operation execution function.		
XML Injection	Unvalidated input is inserted into an XML document.	XPath injection, XQuery injection	91
HTTP Response Splitting	Unvalidated input is used in construction of HTTP response headers.	CRLF injection	113, 93
Malicious File Inclusion	Unvalidated input is used in an argument to file or stream functions.	File inclusion, Remote code execution, Directory traversal	98
Insecure Direct Object Reference	Unvalidated input is used as a reference to an internal implementation object, such as a file, directory, or database key.	Parameter tampering, Cookie poisoning, Path manipulation	233, 73, 472
Cross Site Request Forgery (CSRF)	An application authorizes requests based only on credentials that are automatically submitted by the browser. A CSRF attack forces a logged-in victim's browser to send a request to a vulnerable application, which then performs the chosen action on behalf of	Session riding, One-click attacks, Hostile Linking	352

	the victim, to the benefit of the attacker.		
Information Leakage	Disclosure of sensitive information or the internal details of the application.	File and directory information leaks, System information leak.	538, 200, 497
Improper Error Handling	Error message may display too much information that is useful in exploring a vulnerability.	Error message information leaks, Detailed error handling	388, 209, 390
Weak Authentication and Session Management	Lack of proper protection of account credentials and session tokens through their lifecycle.		287
Session Fixation	Authenticating a user without invalidating any existing session identifier. This gives an attacker the opportunity to steal authenticated sessions.		384
Insecure Communication	Transmitting sensitive information (e.g., session tokens, credit card numbers or health records) without proper encryption (e.g., SSL).		
Unrestricted URL Access	Missing or insufficient access control for sensitive URLs and functions.	Predictable resource location, security by obscurity	425

5.5.1. BENCHMARK SELECTION.

After analyzing all benchmark initiatives summarized in section 4.8, SAMATE-NIST *test suite Juliet 2010* (SAMATE Juliet, 2010) has been considered the most representative and adequate to accomplish this assessment. It has 13782 test cases and covers all weaknesses categories in OWASP top ten 2010 and SANS *25* and therefore satisfies the objectives of this comparative with regard to weakness categories coverage. In SAMATE-NIST *test suite Juliet 2010,* each test case contains a bad function with a particular vulnerability and one (1), two (2) or four (4) good versions of the **bad function**, depending on the case, with different ways of correcting the vulnerability directly validating the input source to the

application (**goodsource**) or validating where vulnerability specifically occurs (**goodsink**). Also, for each vulnerability versions of test cases with different complexities of code (flow) [Kratkiewicz, 2005] are provided and, for each type of code complexity, there are different versions of test cases with different input source type, such as *tcpip connections, console input, database, file, cookies, requests input parameters*, etc. For example in each test case for vulnerability *relative_path_transversal* CWE 23, the following description is mentioned:

** @description*

** CWE: 23 Relative Path Traversal*

** BadSource: connect_tcp Read data using an outbound tcp connection*

** GoodSource: A hardcoded string*

** BadSink: readFile no validation*

** Flow Variant: 09 Control flow: if(IO.static_final_t) and if(IO.static_final_f)*

***

** */*

Each test case has a function called bad() with an input source that is not validated, **badsource**, and a point in the code not validated where the vulnerability materializes, **badsink**. The variation of code complexity, **flow variant** [Kratkiewicz, 2005] is also indicated**.** Each test case can have versions of good functions with good source input, **goodsource,** or good sink, **goodsink.**

The wide range of vulnerability categories with a great number of test cases with different code complexities available in the benchmark SAMATE Juliet 2010 makes necessary to select the most frequent and dangerous ones, according to statistics from vulnerabilities shown in chapter 3. They are distributed in two groups of categories of vulnerabilities:

Vulnerabilities Group 1. This group contains twelve weakness categories that almost all the selected tools are able to detect by design, with some exceptions in two tools that cannot detect any of the categories. This group contains the most dangerous categories given in web applications according OWASP top ten 2010. For each vulnerability some variants in complexity and source are selected, at least one variant of each input source to the application. Table 28 shows the vulnerabilities in group 1.

Table 28
Group 1. Most dangerous vulnerabilities of SAMATE Juliet 2010 test suite. [Bermejo, 2011]

CWE	DESCRIPTION	Nº TEST CASES
23	Relative_Path_Traversal	11
36	Absolute_Path_Traversal	9
78	Command_Injection	10
80	XSS	11
81	XSS_Error_Message	13
83	XSS_Attribute	8
89	SQL_Injection	19
90	LDAP_Injection	11
113	HTTP_Response_Splitting	32
352	Cross_Site_Request_Forgery	7
566	Access_Through_SQL Primary	10
601	Open_Redirect_Servlet	11
TOTAL TEST CASES		154

Vulnerabilities Group 2. This group includes other dangerous categories in MITRE CWE and SANS TOP 25. These ones include disclosure vulnerabilities, cryptographic procedures vulnerabilities, unsynchronized shared data or weak random numbers using. Group 2 is a complement to group 1 for assessment of tools vulnerabilities coverage degree. Table 29 shows the vulnerabilities categories for group 2. There are 32 vulnerabilities categories, each one with two different test cases except for two vulnerabilities categories that has one test case.

Table 29
Group 2. Complement vulnerabilities of SAMATE Juliet 2010 test suite. [Bermejo, 2011]

CWE	DESCRIPTION	Nº TEST CASES
209	Information_Leak_Error	2
256	Plaintext_Storage_of Password	2
257	Storing_Password Rec._Format	2
259	Hard_Coded_Password	2
293	Using_Referer_Field_for Auth.	2
315	Plaintext_Storage_in_a Cookie	2
319	Plaintext_Tx_Sensitive_Info	2
321	Hard_Coded_Cryptographic Key	2
327	Use_Broken_Crypto	2
328	Reversible_One_Way_Hash	2
330	Insufficiently_Random Values	2
336	Same_Seed_in_PRNG	2
338	Weak_PRNG	2
367	TOC_TOU	2
378	Creation_of_File_with Insec_Per	2
413	Insufficient_Resource Locking	1
476	NULL_Pointer_Dereference	2
489	Leftover_Debug_Code	2
497	Information_Leak_SystemData	2
523	Unprotected_Cred_Transport	2
547	Hardcoded_Security Constants	2
549	Missing_Password_Masking	2
567	Unsynchronized_Shared_Data	1
572	Call_Thread_run_Instead start	2
598	Information_Leak QueryString	2
603	Client_Side_Authentication	2
613	Insufficient_Session Exp.	2
614	Sensitive Cookie Without Secure	2
615	Info_Leak_By_Comment	2
643	Unsafe_Treatment_XPath Input	2
759	Unsalted_One_Way_Hash	2
760	Predictable_Salt_One_Way Hash	2
TOTAL TEST CASES		62

5.5.2. SAST SELECTION.

The next step was the selection of commercial and open source security static analysis tools for source or executable code. They must detect vulnerabilities in web applications

developed using the J2EE specification, that it is the most used developing technology, according to section 2.3.2.

According to previous comparatives in related work section (see section 7.2) and analyzing the available commercial tools (see section 4.3.3., table 18), we selected six tools. Coverity Prevent (actually Coverity SAVE) [Coverity, 2103] is not considered because it is designed for no web java applications, not for J2EE applications. It was not possible to obtain CodeSecure from Armorize [Armorize, 2103]. Parasoft mainly focuses on application quality, with a lesser focus on security.

Finally, the selected tools were three commercial tools for source code (Checkmarx CxEnterprise, Fortify SCA and Klocwork INSIGHT), VERACODE SaaS for executable code and two open source (LAPSE+ for source code and FINDBUGS for executable code).

Their main characteristics are:

1. Fortify SCA. v. 5.10.0.0102 [HP-Fortify, 2013]. It supports 18 distinct languages, the most extended OS platforms and it also offers SaaS (Software as a service). IHP claims that the tool detects more than 479 weaknesses (FORTIFY weakness). It presents very complete reports, classifying detections according to four severity levels (HIGH-CRITICAL-MEDIUM-LOW) and industry vulnerability classifications, as OWASP top 10, SANS 20, MITRE CWE and others. It allows the addition of new custom rules defined by the user, to adapt the tool to peculiarities that may require a particular Web application. Regarding to the track information of a detected vulnerability, SCA is very complete. It allows integrating and correlating the results with those of another open source tool like FINDBUGS.
2. Checkmarx CxEnterprise v. 5.5.0 [Checkmarx, 2103]. It supports JAVA, JSP, C#, ASP, VB.NET, VB6, C++, PHP, APEX, JAVASCRIPT and VBSCRIPT languages. It supports the Windows OS platforms, eclipse plugin, and it offers SaaS and a wide

set of vulnerabilities. It presents very complete reports classifying detections according to three severity levels HIGH--MEDIUM-LOW and industry vulnerability classification as OWASP top 10, SANS 25 or MITRE CWE. The track information of a vulnerability detected by the tool is very complete. It doesn´t allow the definition of new rules by the user to detect additional vulnerabilities.

3. Klocwork INSIGHT (v. SOLO JAVA 8.1.2v011) [KLOCWORK, 2103]. It support JAVA-J2EE, C#, C/C++ languages and WINDOWS, UNIX, MAC, ANDROID OS platforms and eclipse plug-in. It detects a wide set of vulnerabilities (http://www.klocwork.com/products/documentation/current/CWE_IDs_mapped_to_Klocwork_Java_issue_types). It allows the addition of new custom rules defined by the user to adapt the tool to peculiarities a particular Web application requires. Regarding to the track information of a vulnerability detected it is a very complete tool.
4. VERACODE SaaS. [VERACODE, 2103]. It offers only SasS (software as a service) static analysis for executable code service. Veracode no allow downloading the tool. It support languages as Java, J2EE-J2ME, CC++, C#, ASP.NET, VB.NET, PHP, ColdFusion (compiled as Java), BLACKBERRY and it covers a wide set of vulnerabilities.
5. LAPSE + v. [Lapse+, 2103]. (open source). Lapse was a tool developed by Benjamin Livshits as part of the Griffin Software Security Project (Securebench Micro). Lapse + is a new version developed by the laboratory eValues the University Carlos III of Madrid. It supports any platform if Java Runtime Environment is available and integrated in Eclipse plugin. It detects a reduced set of vulnerabilities but they are the most important according to OWASP top ten 2010. It not classifies the report of vulnerabilities by severity and its information trace of a

given vulnerability is not adequate and it not has possibility of adding new detecting vulnerabilities by the user.

6. FindBugs. 1.3.9 [Findbugs, 2013] (open source). It supports only java-J2EE language and it can be integrated in eclipse. It supports a reduced set of vulnerabilities, therefore it is not enough for an exhaustive and complete analysis. Its reports of vulnerabilities give few trace information and not classifies them by severity degrees. It permits to the users the addition of new detectors.

5.5.3. EXECUTION RESULTS.

In this section we show the results obtained when the selected tools are executed against the two groups of test cases defined in section 5.2.We only took into account, for each test execution, the vulnerabilities detected for which each test case is designed. Next, the metrics selected in section 5.4 are applied to obtain the most appropriate measures to promote good interpretation of the results and to draw the best conclusions.

Vulnerabilities Group 1 has the main objective of testing the most dangerous categories given in web applications as XSS, SQLI, CSRF or HTTP response splitting. Table 30 accounts for the number of vulnerabilities detected (true positives), with exception of Checkmarx that is not designed for *path traversal* and FINDBUGS not designed for *command injection.* Veracode SaaS send the analysis of vulnerabilities groups requested but XSS test cases were no analyzed. The total of test cases analyzed was 154.

To normalize the result of detections we calculate the percentage of detections for each type of vulnerability (recall) that each tool is designed for detecting. Last file of table 30 shows also the *arithmetic mean* of detection percentage for all types of vulnerabilities that each tool is able of detecting.

Table 30
Vulnerabilities detection for Group 1. True positive ratio [Bermejo, 2011]
ND: Tool not designed for a vulnerability
NA: Vulnerability not analyzed

CWE	VULN.	Nª TC	Checkmarx	SCA	Klocwork	Lapse+	Veracode	Findbugs
23	**Relative Path Traversal**	11	ND	11 (100%)	9 (81,8%)	11 (100%)	7 (63,6%)	1 (0.9%)
36	**Absolute Path Traversal**	9	ND	9 (100%)	7 (77,7%)	9 (100%)	4 (36,3)	1 (0,11%)
78	**Command Injection**	10	4 (40%)	10 (100%)	6 (60 %)	10 (100%)	4 (40%)	ND
80	**XSS**	11	1 (0,9%)	6 (54,5%)	7 (63,6%)	11 (100%)	NA	1 (0,9%)
81	**XSS Error Message**	13	10 (76,9%)	5 (38,4%)	6 (46,1%)	6 (46,1%)	NA	0 (0%)
83	**XSS Attribute**	8	0 (0%)	4 (50%)	6 (75%)	7 (87,5%)	NA	0 (0%)
89	**SQL Injection**	19	19 (100%)	19 (100%)	14 (73,6%)	19 (100%)	12 (63,1%)	5 (23,6%)
90	**Ldap Injection**	11	8 (72,7%)	11 (100%)	5 (45,4%)	5 (45,4%)	0 (0%)	ND
113	**http Response Splitting**	32	15 (46,8%)	16 (50%)	10 (31,2%)	18 (56,2%)	12 (37,5%)	3 (0,09%)
352	**CSRF**	7	7 (100%)	7 (100%)	5 (71,4%)	7 (100%)	NA	0 (0%)
566	**Access Through SQL Primary**	10	10 (100%)	10 (100%)	0 (0%)	10 (100%)	0 (0%)	0 (0%)
601	**Open Redirect Servlet**	11	7 (63,6%)	6 (54,5%)	9 (81,8%)	11 (100%)	7 (63,6%)	3 (27,2%)
TP			**81/124**	**114/154**	**84/154**	**123/154**	**46/115**	**14/133**
% TP mean			**60**	**78,9**	**58,9**	**86,2**	**38**	**5,3**

Table 31 shows a summary of results in the total of 329 false positive test cases. There are more test cases than in table 30 because the bad function of each test case has several good versions (see appendix C for detailed results) .

To normalize the results we calculate the percentage of false positives for each type of vulnerability that each tool is designed for detecting. Last file of table 31 shows the *arithmetic mean* of false positives percentage of all vulnerabilities types that each tool is able to detect

Table 31.
Vulnerabilities detection for Group 1. False positive ratio [Bermejo, 2011]
ND: Tool not designed for a vulnerability
NA: Vulnerability not analyzed

CWE	VULN.	Nº TEST CASES	Checkmarx	SCA	Klocwork	Lapse+	Veracode	Findbugs
23	**Relative Path Traversal**	18	ND	15 (83,3%)	13 (72,2%)	18 (100%)	6 (30%)	1 (0,05%)
36	**Absolute Path Traversal**	14	ND	11 (78,5%)	8 (57,1%)	14 (100%)	2 (14,2%)	0 (0%)
78	**Command Injection**	16	5 (31,2%)	15 (93,7%)	8 (50%)	16 (100%)	4 (25%)	ND
80	**XSS**	15	0 (0%)	6 (40%)	8 (53.3%)	15 (100%)	NA	2 (13,3%)
81	**XSS Error Message**	23	18 (78,2%)	6 (26%)	8 (34,7%)	8 (34,7%)	NA	0 (0%)
83	**XSS Attribute**	13	0 (0%)	4 (30,7%)	6 (46,1%)	12 (92,3%)	NA	0 (0%)
89	**SQL Injection**	58	50 (86,2%)	50 (86,2%)	39 (67,2%)	54 (93,1%)	26 (44,8%)	11 (18,9%)
90	**ldap Injection**	17	9 (52,9%)	15 (88,2%)	7 (41,1%)	7 (41,1%)	0 (0%)	ND
113	**http Response Splitting**	88	35 (39,7%)	41 (46,5%)	19 (21,5%)	49 (55,6%)	16 (18,1%)	6 (0,06%)
352	**CSRF**	20	17 (85%)	18 (90%)	11 (55%)	18 (90%)	NA (0%)	0 (0%)
566	**Access Through SQL Primary**	30	28 (93,3%)	27 (90%)	0 (0%)	28 (93,3%)	0 (0%)	0 (0%)
601	**Open Redirect Servlet**	17	9 (52,9%)	7 (41,1%)	12 (70,5%)	17 (100%)	6 (35,2%)	0 (0%)
FP			**171/297**	**215/329**	**148/329**	**256/329**	**54/276**	**20/296**
% FP mean			**51,9**	**66,1**	**47,4**	**83,3**	**20,6**	**3,2**

Vulnerabilities Group 2 has the goal of analyzing the coverage degree of vulnerabilities in each tool as well the true and false positives ratio. This group includes disclosure vulnerabilities, weaknesses in cryptographic procedures, synchronization variables, weak random numbers uses and others described in table 9. Table 32 shows a test execution summary of the second group vulnerabilities.

Table 32
Group 2. Vulnerabilities coverage and TRUE/FALSE positive ratio [Bermejo, 2011]

GROUP 2 – VULNERABILTIIES COVERAGE – 32 V. CATEGORIES – 62 VULNERABILITIES						
METRIC / TOOL	**checkmarx**	**SCA**	**Klocwork**	**Lapse+**	**Veracode**	**Findbugs**
TOTAL TRUE POSITIVES	14 (22%)	23 (37%)	15 (24%)	4 (6%)	15 (15%)	1 (1%)
TOTAL FALSE POSITIVES	20 (32%)	36 (58%)	18 (29%)	8 (12%)	19 (30%)	1 (1%)
NUMBER OF VULN. CATEGORIES WHICH A TOOL IS DESIGN TO DETECT.	12 (37%)	32 (100%)	9 (32%)	1 (3%)	13 (40%)	1 (3%)
NUMBER OF VULN. NOT DETECTED NONE TOOL	30 (total: 62 vulnerabilities)					

5.5.4. ASSESSMENT RESULTS.

Vulnerabilities Group 1 result. The assessment of execution results against the benchmark is accomplished applying the following metrics of section 5.3 to the vulnerability group 1.

- False positive percent
- Recall
- Precision
- F-measure

Precision metric indicates the best relation between *true positive* and *false positive* score. Usually a tool has a direct proportionality between their true and false positives results. A good tool should break this direct proportionality. In a new version of a tool, it should not have more false positives if the tool detects more vulnerabilities. A tool with better *precision* indicates it has a better relationship between true and false positives.

The classification order is determined by *F-measure* metric because it normalizes *precision* and *recall* metrics, see table 33.

- *Recall mean* is calculated as the arithmetic mean of all vulnerabilities types recall got by each tool. Table 30 shows how *recall mean* is calculated.
- *TP mean* is a value obtained from % TP mean (recall x 100). For example, for SCA, *TP mean* is:

  % TP mean *X vulnerabilities covered /100* = 78,9 x 154 /100 = 121,5
- *FP mean* is a value obtained from % FP mean. For example, for SCA, *FP mean* is:

  % FP mean X vulnerabilities covered / 100 = 66,1 x 154 / 100 =101,7

Table 33
Assessment results computing the selected metrics [Bermejo, 2011]

TOOL	TP mean	FP mean	PRECISION	RECALL mean	F-MEASURE
FORTIFY SCA	121,5	101,7	0,542	0,789	**0,642**
LAPSE+	132,7	128,2	0,508	0,862	**0,639**
KLOCWORK	90.7	72,9	0,554	0,589	**0,570**
CHECKMARX	74,4	64,3	0,536	0,60	**0,566**
VERACODE	47,1	23,6	0,666	0,38	**0,483**
FINDBUGS	7	4,2	0,625	0,053	**0,100**

Vulnerabilities Group 2 results:

- The true and false positive percentages.
- The number of categories of vulnerabilities that a tool does detects with respect to the total of 62 categories of vulnerabilities that are tested in group 2.
- The number of vulnerabilities categories not detected by any tool: 30 corresponding to 15 vulnerability categories (table 34).

Table 34
Vulnerabilities categories not detected by any tool. [Bermejo 2011]

Vulnerabilities categories	CWE
Storing_Password Rec._Format	257
Hard_Coded_Password	259
Using_Referer_Field_for Auth.	293
Plaintext_Tx_Sensitive_Info	319

Same_Seed_in_PRNG	336
TOC_TOU	367
Insufficient_Resource Locking	413
Unprotected_Cred_Transport	523
Hardcoded_Security Constants	547
Missing_Password_Masking	549
Information_Leak QueryString	598
Insufficient_Session Exp.	613
Client_Side_Authentication	603
Unsalted_One_Way_Hash	759
Predictable_Salt_One_Way Hash	760
Information_Leak_Error	209

The correlation of results of the six tools together gives a total of 147 detections this is a 95% of the 154 possible vulnerabilities.

Table 35 shows the correlation results of tools by pairs. This option can be more economic with respect to the better results obtained than the other one obtained by only tool. FINDBUGS doesn´t appear in tables, due to its poor detection results that make it not worth making an effort to combine it with another tool.

Table 35
Results correlation of detections (true positives) between pair of tools [Bermejo, 2011]

RESULTS CORRELATION OF DETECTIONS (True Positives) BETWEEN PAIR OF TOOLS, 154 TEST CASES	
CHECKMARX – LAPSE+	142
LAPSE+ – KLOCWORK	139
FORTIFY SCA - LAPSE+	133
FORTIFY SCA - KLOCWORK	130
FORTIFY SCA - CHECKMARX	127
CHECKMARX – KLOCWORK	123
VERACODE - FORTIFY SCA	99
VERACODE – LAPSE+	99
VERACODE- CHECMARKX	83
VERACODE – KLOCWORK	71

Table 36 shows the results of test execution in group two (2) with 62 category weakness, give a good idea of coverage degree in the vulnerabilities categories included in that group.

Table 36.
Vulnerabilities categories coverage for group two (2)

TOOL	Nº CATEGORÍES COVERED
FORTIFY SCA	32
VERACODE	13
CHECKMARX	12
KLOCWORK	9
LAPSE+	1
FINDBUGS	1

5.5.5. CONCLUSIONS.

The main conclusion from the analysis of the results of this assessment is that the use of static tools source and executable code is very useful within the SSDLC for web applications. Very high vulnerability detections percentages are achieved reaching in some cases to exceed 80% in isolation. The average ratio of precision for all tools is 0.571 and the average ratio of recall is 0,545. The tools cover the most dangerous vulnerabilities, but they do not cover any other also important vulnerabilities categories (See table 32).

The number of false positives, high in general, in four cases over 50%, must be reduced in a subsequent audit the results. All tools, except FINDBUGS, have good error trace facilities to accomplish an adequate post audit. The subsequent performing of the audit requires adequate preparation about vulnerabilities knowledge in the code language that is being audited.

The tools are consistent across all test cases in the sense of when they no detect a real vulnerability, they don't give false positive warnings in the corrected versions of the functions for analysis of false positives. In classification of table 33, by *f-measure* metric, FORTIFY SCA has the best score followed by LAPSE+, followed by KLOCWORK, CHECKMARX and VERACODE with similar score.

Combining two or more tools may improve the outcome of total detections reaching more than 92% (142) using an open source tool as LAPSE+ with another commercial tool as CHECKMARX. Note that by combining the six tools we reached just over 95% (147), but, by selecting only two of them, the same result can be obtained. With five tools of this comparison very good results are obtained combining them by pairs. Four of the tools, SCA, CHECKMARX, KLOCWORK y LAPSE+ combined between them by pairs, obtain percentages of detections between 80% and 92%.

It is also important to consider the possibility of using a SAST for executable code. A company has not source code for commercial web applications and the average percentage of commercial software is about 22% and almost 75% of them does not have acceptable conditions of security [Veracode 2012].

The degree of coverage of vulnerabilities from the second group test is bad for open source tools, which focus on the most frequent vulnerabilities of the first group with the exception of some information disclosure vulnerabilities. If LAPSE+ would expand the degree of weaknesses coverage, it could significantly raise its category. The degree of coverage is higher for commercial tools (Fortify SCA covers all categories of vulnerabilities). It is quite acceptable covering the most frequent and important, however, almost all of them have wide field for improving.

In general, the detections in the categories of vulnerabilities of the second group are related to disclosure of information in the code, weak cryptographic protocols and weak random numbers. All tools obtain worse ranking results in second group than in the first group with 30 of 62 potential vulnerabilities not detected by any tool. In general all tools analyzed except Fortify SCA need to increase the vulnerability coverage. The changing nature and evolution of the categories of vulnerabilities over time, requires a continuous study to adapt the tools to this development to have them always adapted to the time of use.

Finally, let's note that there are not many more chances in the open source market and, therefore, and given the importance of these tools, it is necessary to promote and enhance their development and research to extend and improve them and include them in a SSDLC of web applications. The frequency and dangerous trend of vulnerabilities changes over time. This involves the analysis of security vulnerabilities trends and attacks to exploit the vulnerabilities. The study of vulnerabilities trends is necessary to adapt the tools. Also new benchmarks development for all type of tools and for mores languages are essential for accomplishing new assessments that aid companies and developers make the best election.

5.6. DAST, IAST AND HYBRID ASSESSMENT IN WEB APPLICATIONS.

Other types of security automatic tools that can be used to analyze the security of a web application are DAST, IAST or HYBRID tools (see sections 4.4, 4.5, 4.6) that combine some of SAST, DAST, IAST or HYBRID tools. In this section several known tools are compared to evaluate their performance about vulnerability detections, false positive ratio and vulnerability coverage degree. The tools are evaluated following the methodology of section 5.2 and metrics of section 5.3. The metrics applied to the results allow accomplishing an exhaustive analysis and rank the performance degree of tools and extract conclusions about their usability and recommendations for their using by companies or organizations.

5.6.1. BENCHMARK SELECTION.

To evaluate DAST or IAST tools the most adequate benchmarks are always application because these tools are dynamic and analyze an application in runtime.

WAVSEP application is the benchmark selected from the analysis of benchmarks applications in section 4.8. Project WAVSEP includes the following test cases**:**

1. For detection of vulnerabilities:

    - Path Traversal/LFI, CWE 22: 816 test cases, implemented in 816 jsp pages (GET & POST).
    - Remote File Inclusion (XSS via RFI) CWE 73: 108 test cases, implemented in 108 jsp pages (GET & POST).
    - Reflected XSS, CWE 79: 66 test cases, implemented in 64 jsp pages (GET & POST).
    - Error Based SQL Injection, CWE 79: 80 test cases, implemented in 76 jsp pages (GET & POST).
    - Blind SQL Injection, CWE 79: 46 test cases, implemented in 44 jsp pages (GET & POST).
    - Time Based SQL Injection, CWE 79: 10 test cases, implemented in 10 jsp pages (GET & POST).

2. False Positives:

    - 7 different categories of false positive Reflected XSS vulnerabilities (GET & POST)
    - 10 different categories of false positive SQL Injection vulnerabilities (GET & POST)

- 8 different categories of false positive path traversal/LFI vulnerabilities (GET & POST)
- 6 different categories of false positive remote file inclusion vulnerabilities (GET & POST)

The selected tools have been executed against above WAVSEP test cases. WAVSEP, as analyzed in section 4.8.1, is a test suite with 1126 different test cases for vulnerabilities detection checks and 31 test cases for false positive checks. The vulnerabilities set of this application benchmark are between the most dangerous and frequents according to Veracode report volume 5 [Veracode, 2012]. In this report, XSS, SQLI, RFI and LFI vulnerabilities are the 65% percent of total vulnerabilities found in all applications analyzed.

5.6.2. DAST, IAST, HYBRID TOOLS SELECTION.

After examining these types of security tools in sections 4.4, 4.5 and 4.6, and according to the tools availability, the tools selected for the assessment are the following 11 DAST, IAST AND HYBRID tools:

- HP-WEBINSPECT (DAST). V. 9.30 [HP-Fortify, 2013]. HP-WebInspect will trace and record code paths through JavaScript-Ajax, Adobe Flash, anti-CSRF support, Web Service requests and WSDL crawler. HP WebInspect can integrate dynamic and real-time analysis to find more vulnerabilities and fix them faster. It works in concert with HP Fortify SecurityScope to observe attacks at the code level during dynamic scans. It identifies and crawls more of an application to expand the coverage of the attack surface and detect new types of vulnerabilities. It provides stack traces and line-of-code detail to confirmed vulnerabilities.

- NETSPARKER (DAST) v. 2.3 [Mavituna, 2013]. Netsparker incorporates a JavaScript engine that can parse, execute and analyze the output of JavaScript and VBScript. This allows Netsparker to crawl and interpret web applications that rely on client-side scripting, including custom code execution, AJAX operations or page content that is dynamically created using frameworks such as jQuery. It has also support for anti-CSRF.

- BURP SUITE (DAST) v. 1.4.10 [Portswigger, 2013]. Burp Scanner identifies vulnerabilities such as SQL injection, cross-site scripting and file path traversal. It has support for CSRF tokens.

- W3AF (DAST) v. 1.2 [W3AF, 2103]. It is an open source tool with support for AJAX and Web application firewall integration.

- OWASP-ZAP (DAST) v. 2.2.2 [OWASP, 2103]. It is an open source OWASP 7987project scanner with support for AJAX, XML, JSON or CSRF tokens.

- IRONWASP (DAST) v. 0.9.7.1 [Ironwasp, 2013]. It includes Javascript static analysis with support for SAP analysis and anti-CSRF support. It detects XSS, SQLI, RFI, LFI, XPATH, OPEN redirect, LDAP injection and others.

- ARACHNI (DAST) v. 2.2.1 [Arachni, 2013]. Wapiti is an open source tool that can detect the vulnerabilities as LFI, RFI, XSS, SQLI, CRLF Injection, HTTP Response Splitting, CSRF, LDAP injection or XPATH injection.

- ACUNETIX (DAST-IAST) v. 9 [Acunetix, 2013]. Acunetix-Acusensor is a security technology with feedback from sensors placed inside the source code, while the source code is executed (IAST). It has support for AJAX, WSDL and their results

can now be imported into a Web Application Firewall (WAF). Acusensor is an option for PHP and .NET applications (not J2EE).

- IBM SECURITY APPSCAN STANDARD (DAST-IAST) v. 8.5 [IBM-Appscan, 2013] performs correlation and triages security testing results from dynamic black box testing and IAST glass box solution scans and JavaScript Security Analyzer for static taint analysis of client-side security issues.

- SEEKER (IAST) v. 2.6 [Quotium, 2013]. Seeker analyzes the application code and data as it runs, in response to simulated attacks. Seeker monitors application behavior and data flow across modules, components, tiers and servers to accurately identify application threat. Seeker detects all OWASP top ten 2010 and 2103 vulnerabilities. Seeker's BRITE (Behavioral Runtime Intelligent Testing Engine) conducts runtime analyses of the application code and of memory and data flow based on the application behavior by using agents on each of the application servers. Seeker tracks code-flow through multiple tiers in distributed architectures and in different code modules. By assimilating into the application environment it learns its behavior and identifies application security vulnerabilities.

- HP FORTIFY HYBRID ANALYSIS (SAST-DAST-IAST) v. 3.20 [HP-Fortify, 2013]. It is composed by a IAST tool (SecurityScope) that interchanges real-time attack information to a DAST tool (Webinspect). After that, the results can be correlated with SAST analysis (Fortify SCA) to decrease false positives and increase the number of detections or true positives.

Tables 37 and 38 summarize the vulnerability coverage of DAST, IAST and HYBRID tools selected.

Note: ☺ covered, 🙁 not covered.

Table 37
Vulnerability coverage of DASD, IAST and HYBRID tools selected (1)

	CWE	WEBINSPECT	APPSCAN	ACUNETIX	NETSPARKER	BURP
SQLI	**89**	☺	☺	☺	☺	☺
XSS	**79**	☺	☺	☺	☺	☺
LFI	**22**	☺	☺	☺	☺	☺
RFI	**73**	☺	🙁	☺	☺	🙁
CMDI	**78**	☺	☺	☺	☺	☺
OPEN REDIRECT	**601**	☺	☺	☺	☺	☺
CLRF	**113**	☺	☺	☺	☺	☺
LDAPI	**90**	☺	☺	☺	☺	☺
XPATHI	**643**	☺	☺	☺	☺	☺
XMLI	**91**	🙁	☺	🙁	🙁	☺
Buffer O.	**120**	☺	☺	🙁	🙁	🙁
Integer O.	**190**	☺	☺	🙁	🙁	🙁
FMT	**134**	☺	☺	🙁	🙁	🙁
XXE	**611**	🙁	☺	☺	🙁	☺
SESSION	**384**	☺	☺	🙁	🙁	☺
CSRF	**352**	☺	☺	☺	🙁	🙁

Table 38
Vulnerability coverage of DASD, IAST and HYBRID tools selected (2).

	CWE	SEEKER	HP-HYBRID	W3AF	ARACHNI	ZAP	IRONWASP
SQLI	**89**	☺	☺	☺	☺	☺	☺
XSS	**79**	☺	☺	☺	☺	☺	☺
LFI	**22**	☺	☺	☺	☺	☺	☺
RFI	**73**	🙁	☺	☺	☺	🙁	☺
CMDI	**78**	☺	☺	☺	🙁	🙁	☺
OPEN REDIRECT	**601**	☺	☺	☺	☺	☺	☺
CLRF	**113**	☺	☺	☺	☺	☺	☺
LDAPI	**90**	☺	☺	☺	☺	🙁	☺
XPATHI	**643**	☺	☺	☺	☺	🙁	☺
XMLI	**91**	☺	☺	🙁	🙁	🙁	🙁
Buffer O.	**120**	☺	☺	☺	🙁	🙁	🙁
Integer O.	**190**	☺	☺	🙁	🙁	🙁	🙁

FMT	**134**	☺	☺	☺	😐	😐	😐
XXE	**611**	☺	☺	😐	😐	😐	😐
SESSION	**384**	😐	☺	☺	😐	😐	☺
CSRF	**352**	☺	☺	☺	☺	😐	😐

5.6.3. EXECUTION RESULTS.

The tools are executed against WAVSEP test suite. The 1126 test cases correspond to XXS (66), SQLI(136), LFI(816) and RFI(108) vulnerabilities. Table 39 shows the benchmark detection results and table 40 shows the false positive results, where:

ND: Non designed for detecting a vulnerability.

TC: Test Case.

TP: True positive.

FP: False positive

Table 39
WAVSEP Benchmark detection results.
ND: Non designed for detecting a vulnerability

	XSS (66 TC)	SQLI (136 TC)	RFI (108 TC)	LFI (816 TC)	TOTAL DETECTIONS	TP % Mean
Seeker	66 (100%)	136 (100%)	ND	816 (100%)	1018	**100**
HP-Fortify Hybrid	66 (100%)	136 (100%)	85 (78.7%)	446 (54,6%)	733	**83,32**
Appscan	66 (100%)	136 (100%)	ND	396 (48,5%)	598	**82,83**
Burp	60 (90,9%)	136 (100%)	ND	436 (53,4%)	632	**81,43**
Webinspect	66 (100%)	135 (99.3%)	66 (61,11%)	406 (49,75%)	673	**78,78**
Ironwasp	50 (75,7%)	136 (100%)	108 (100%)	288 (35,2%)	582	**77,72**
Nestparker	64 (96,9%)	136 (100%)	48 (44,4%)	453 (55,5%)	701	**74,20**
ZAP	66 (100%)	103 (75,7%)	ND	342 (42%)	511	**72,56**
Acunetix	66 (100%)	136 (100%)	48 (44,4%)	262 (32,11%)	512	**69,12**

Arachni	136 (100%)	65 (98,5%)	48 (44,5%)	168 (20,6%)	417	**65,90**
W3AF	20 (30,3%)	81 (59,5%)	12 (11,1%)	469 (57,8%)	582	**39,6**

IAST (Sekeer) and HYBRID (HP-Fortify hybrid and Appscan) tools obtain the best results.

Notes about the results summarized in table 39:

- Tools have been ordered by their vulnerabilities percentage average of detections (TP % Mean).
- TP % Mean is the mean of vulnerabilities detection percentages.
- The TP percent mean normalizes the vulnerability detections.
- The vulnerabilities not designed to be detected by a tool have not been considered.

Table 40
WAVSEP Benchmark false positive results.
ND: Non designed for detecting a vulnerability

	XSS (7 TC)	SQLI (10 TC)	RFI (6 TC)	LFI (8 TC)	TOTAL FP	FP % Mean
Acunetix	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0	**0**
Burp	0 (0%)	0 (0%)	ND	0 (0%)	0	**0**
Seeker	0 (0%)	0 (0%)	ND	0 (0%)	0	**0**
HP-Fortify Hybrid	0 (100%)	3 (30%)	0 (0%)	0 (0%)	3	**0**
Nestparker	0 (0%)	3 (30%)	0 (0%)	0 (0%)	3	**0,75**
Appscan	0 (0%)	3 (30%)	ND	0 (0%)	3	**0,75**
Webinspect	0 (0%)	3 (30%)	0 (0%)	1 (12,5%)	4	**10,62**
Arachni	0 (0%)	5 (50%)	0 (0%)	0 (0%)	5	**12,5**
ZAP	0 (0%)	5 (50%)	ND	4 (50 %)	9	**25**
W3AF	3 (42,8%)	3 (30%)	1 (16,6%)	1 (12,5%)	8	**25,4**
Ironwasp	0 (100%)	5 (50%)	0 (0%)	1 (12,5%)	6	**40,6**

The worst results are obtained by open source DAST tools.

Notes about the results summarized in table 40:

- Results have been ordered by FP percent mean (FP % Mean).
- FP % mean is the average of vulnerabilities false positive percentages.
- The FP percent main normalizes the vulnerability false positive results.
- The vulnerabilities no designed to be detected by a tool have not been considered.

5.6.4. ASSESSMENT RESULTS.

The next step is to apply the metrics of section 5.3 to the results of previous section to obtain a strict ranking of the tools based in F-measure metric that normalizes the precision and recall results.

The metrics (see section 5.3) shown in table 41 are:

- % Vulnerabilities covered.
- TP mean. *TP mean* is a value obtained from % TP mean in table 39. For example, for Appscan, *TP mean* is:

  % *TP mean X vulnerabilities covered* = 82,8 x 1018 / 100
- FP percent mean. *FP mean* is a value obtained from % FP mean calculated in table 40. For example, for *Appscan*, *FP mean* is:

  % FP mean X vulnerabilities covered / 100 = 0,75 x 1018 / 100
- Recall.
- Precision.
- F-measure.

Table 41
Metrics applied to WAVSEP test suites weighted results

METRICS APPLIED TO RESULTS	% Vul. COVERED Tables 37, 38	TP (mean)	FP (mean)	PRECISION	RECALL (mean)	F-MEASURE
Seeker	**87,5 %**	1018	0	1	1	**1**
HP-Fortify Hybrid	**100 %**	943,5	0	1	0,833	**0,908**
Appscan	**93,7 %**	842,9	7,63	0.991	0,828	**0,902**
Burp	**31,2 %**	828,6	0	1	0,814	**0,897**
Nestparker	**56,2 %**	835,4	8,44	0,989	0,742	**0,847**
Webinspect	**87,5 %**	886,1	119,3	0,881	0,787	**0,831**
Acunetix	**68,7 %**	778,0	0	1	0,691	**0,817**
Arachni	**56,2 %**	742,0	140.7	0,840	0,659	**0,738**
ZAP	**31,2 %**	738,0	281,5	0,721	0,725	**0,722**
Ironwasp	**62,5 %**	874,9	457,1	0,656	0,772	**0,709**
W3AF	**81,2 %**	445,8	286	0,609	0,396	**0,479**

5.6.5. CONCLUSIONS.

Table 41 ranks the tools performance according to F-measure metric. The tools score for F-measure are good for almost all tools selected.

The vulnerabilities of WAVSEP benchmark are adequate to test DAST. These vulnerabilities can be detected by using almost any DAST tool as can be seen in section 4.4. This is the main reason for DAST good results. DAST must be tested against a benchmark according to the DAST possibilities. WAVSEP vulnerabilities are between the most dangerous in OWASP top ten 2013 or SANS top 25. The results obtained by tools are very good, with detection percentages from 65% to 83% for all tools except for W3AF. DAST tools have more problems to detect the LFI vulnerability. DAST tools generally exhibits good rates of false positives (see table 40). Acunetix, Burp and Appscan obtain false positive ratios < 1%. Open source tools as Arachni, ZAP and Ironwasp and W3AF commercial tool obtain higher false positive ratio. Their false positive alerts vary from 5 to 9 of 31 in total

The analysis performed against WAVSEP benchmark shows the crawling capacity to find the complete structure and links of an application is higher for commercial DAST tools than open source tools.

Some IAST tools, as SEEKER, have been tested with very good results as shown in table 41. This a white box tool for runtime testing of web applications. Seeker gives also the possibility of manual crawling and performs a test of all input sources it finds in their analysis and it also permits using its crawling tool or another external crawling tool. Seeker IAST tool has zero false positive alerts confirming the few false positives alerts for IAST tools based in runtime white box testing.

Also HYBRID tools as HP-Fortify hybrid and APPSCAN standard edition (DAST-IAST). Acunetix is a DAST and IAST solution (PHP and .NET). IAST is not available for J2EE. HP-Fortify hybrid IAST (SecurityScope) confirm the test attack results of DAST (Webinsptect) and their result are correlated along with SAST (SCA). HP-Fortify hybrid obtains a F-measure score of 0,908. APPSCAN is also a DAST-IAST tool where IAST is used to confirm the test results of DAST. Many of the results (SQLI and XSS) of DAST (Webinspect) have been confirmed by IAST (SecurityScope).

Open source DAST tools obtain worse results than commercial tools but their results are good. The f-measure score for three tools are higher of 0,700. Commercial tools also have more features for web 2.0 analyses as AJAX, HTML5, JAVASCRIPT or WEB SERVICES. Today is a requisite that a tool can perform client side code in rich internet applications. Moreover commercials tools have better vulnerabilities coverage degree (tables 37 and 38). However some commercial tools as Seeker, Burp and Appscan do not cover the RFI vulnerability (see tables 37 and 38) included in the assessment with WAVSEP test suites. Another vulnerabilities for web services as XMLI, XPATH and XXE are not covered by some commercial and open source tools. Also CSRF, Buffer overflow and Integer overflow

are no covered by some commercial and open source tools. The requisite of testing web service applications make necessary improving the vulnerability coverage of the tools. DAST tools are incorporating additional utilities to improve their performance as:

- Static analyzers for javascript code for server side or client side code (AJAX applications) as Appscan or Ironwasp by example.
- The posibilitiy of analizing web services as Acunetix, HP-Webinspect or Appscan.
- The possibility of incorporate a IAST tool to comfirm the veracity of the vulnerabilities (false positives) as HP-Webinspect (Secutityscope) or Appscan (Glassbox).
- The possibility of incorporte an IAST tool to discover additional vulnerabilities as Acunetix with Acusensor option for .NET and PHP languages only.
- Supplying the information of analysis to a WAF to configure more precise rules to protect a Web application in production phase.

HP-Fortify hybrid tool requires correlating the results of its component SAST-DAST-IAST tools. This correlation implies the confirmation of many true positives and makes easier checking if an alert is a false positive in the posterior auditory. The false positive ratio obtained is 0% (3 alerts) for FP % mean (see table 40).

Actually there are no many commercial implementations of Hybrid tools that incorporate SAST, DAST and IAST integrated tools. Only enterprises as HP and IBM (it has not been possible to obtain IBM Security Appscan Enterprise for the assessment) offer Hybrid solutions and Whitehat Security offers the SaaS Whitehat Sentinel product. The open source Hybrid solutions are mainly academic research. As can be seen in section 4.6.2, there are other hybrid implementations SAST-DAST (Check 'n' Crash [Csallner, 2005]), SAST-IAST (PHP VULNERABILITY HUNTER [Hunter, 2013], AMNESIA [Halfond, 2006]) or DAST-IAST (Acunetix-acusensor, IBM Security Appscan Standard). In our opinion,

Hybrid solutions have a large field for development and investigation and they can be object of future works.

5.7. ASSESSMENT CONCLUSIONS.

The methodology is repeatable and applies widely known metrics based on rates of true and false positives and vulnerabilities coverage degree of tools, producing a strict scale for the performance of static analysis tools. Then, a company can choose a tool by analyzing the precision, recall, F-measure and vulnerabilities coverage metrics.

Following we summarize the main conclusions of the assessments accomplished in this section about SAST, DAST, IAST and HYBRID security tools. They have been organized in three groups:

- **SAST assessment for C/C++ applications. The** obtained results against SAMATE test suites demonstrate that commercial tools (as *Prevent, SCA, K8-Insight, Cx-enterprise and Codesonar*) show a better performance, usability and vulnerabilities coverage than the other analyzed tools. However, all the analyzed tools obtain different results for different types of vulnerabilities and cover different subsets of them.

  A simple execution of many of these tools against a piece of code is not enough to get reliable results, and raw results (results from the first execution) must be reviewed. Of course the automatic execution of tools gives a formidable first step, especially for analyzing lengthy code, but this is not enough. A careful analysis of the results by an experienced user or team (with security skills and experience in the language used in the target code) is always necessary.

The use of tools for static source code analysis to search security vulnerabilities must be integrated as a part of the security policy of any development organization. But current state of these tools does not allow indistinctly using them. The tools' internal designs and reporting output formats are different, so they produce substantially different results.

- **SAST assessment for J2EE web applications.** Very high vulnerability detections percentages are achieved reaching in some cases to exceed 80% in isolation. The average ratio of precision for all tools is 0.571 and the average ratio of recall is 0,545. The tools cover the most dangerous vulnerabilities, but they do not cover any other also important vulnerabilities categories (See table 32). The number of false positives, high in general, in four cases over 50%, must be reduced in a subsequent audit the results. Combining two or more tools may improve the outcome of total detections reaching more than 92% (142 detections)

  It is also important to consider the possibility of using a SAST for executable code. A company has not source code for commercial web applications and the average percentage of commercial software is about 22% and almost 75% of them does not have acceptable conditions of security [Veracode 2012].

  The degree of coverage of vulnerabilities from the second group test is bad for open source tools, which focus on the most frequent vulnerabilities of the first group with the exception of some information disclosure vulnerabilities. The degree of coverage is higher for commercial tools (Fortify SCA covers all categories of vulnerabilities). It is quite acceptable covering the most frequent and important, however, almost all of them have wide field for improving.

  In general, the detections in the categories of vulnerabilities of the second group are related to disclosure of information in the code, weak cryptographic protocols and weak random numbers. All tools obtain worse ranking results in second group than

in the first group with 32 of 62 potential vulnerabilities not detected by any tool. The changing nature and evolution of the categories of vulnerabilities over time, requires a continuous study to adapt the tools to this development to have them always adapted to the time of use.

- **DASD, IAST and HYBRID tools for J2EE web applications.** The tools score for F-measure are very good for almost all tools selected. The vulnerabilities of WAVSEP benchmark are adequate to test DAST. DAST must be tested against a benchmark according to the DAST possibilities (see section 4.5). DAST only perform syntactic analysis and can detect a limited set of vulnerabilities [Sttutard, 2008]. WAVSEP vulnerabilities are between the most dangerous in OWASP top ten 2013 or SANS top 25. There are important vulnerabilities as RFI not covered by four commercial tools. The using of new IAST and HYBRID tools must be promoted to achieve better detections results. Seeker is a IAST tool that obtains excellent results in WAVSEP assessment. Different approximations of HYBRID tools are appearing and we think they are yet in the beginning of their development.

The assessments show that all types of evaluated tools must improve the detection, false positive and vulnerability coverage ratios. These assessments allow establishing a strict rank between the tools involved in each evaluation according their performance, showing their capabilities about additional features and vulnerability coverage degree. The performance degree of tools is calculated with F-measure, recall and precision metrics obtained from execution results against selected benchmarks.

6. DISCUSSION

Following the methodology, for the assessments performed in sections 5.4, 5.5, 5.6, this chapter tries to answer a number of research questions that complement our study:

1- Which is the true positives / false positives balance for the analyzed tools?

2- Which is the usability level of the tools? After executing each tool, do the output of the tool needs interpretation?, and to what degree?, by an experienced user? In other words, is any user able to interpret properly the output of the tools and correct the vulnerabilities found in code? Or do we need people with information security skills?

3- Which is the degree of adequacy of benchmarks within the proposed methodology?

4- How static, dynamic and hybrid tools must be integrated in SSDLC?

6.1. RESEARCH QUESTION 1: Which is the true positives / false positives balance for the analyzed tools?

For the case of SAST tools assessment in C-C++ applications, the results in table 26 for F-measure metrics normalize the different number of test cases that each vulnerability type has in test suites 45 and 46. In our opinion this weighted value is probably the best option to rank the tools based on the precision and recall metrics with normalized results for each vulnerability category. F-measure expresses the performance degree and the balance between true positive and false positive ratios. F_2-measure, which weights recall higher than precision, and the $F_{0,5}$-Measure, which puts more emphasis on precision than recall, are shown to give other additional point according with organizations preferences on detections and false positives rates.

Perhaps the best criterion to select a tool could be the combination of F-measure and vulnerability coverage metrics. When F-measure (table 33) is combined with the percentage of vulnerabilities covered, *SCA* has the best result.

The large difference in results among the analyzed tools proves that many efforts must yet be done to standardize the behavior of static analysis tools.

SAST for web applications assessment. The results for F-measure metrics (table 33) normalize the different number of test cases that each vulnerability type has in SAMATE Juliet test suites for J2EE applications. The F-measure results of SAST tools for web applications are generally worse than SAST tools for C/C++. By example, the results of HP-Fortify SCA are:

- SCA for C/C++:**0,674**.
- SCA for J2EE: 0,642
- KLOCWORK for C/C++: **0,652**
- KLOCWORK for J2EE: 0,570

Precision metric normalizes TP (true positives) and FP (false positives). Findbugs and Veracode have the best results for precision metric. They have few false positives but also very few true positives. The goal of F-measure is to normalize precision and recall metrics to obtain the definitive ranking of the tools (section 5.5.4). Finally Findbugs and Veracode have the worst rank for F-measure metric.

DAST, IAST, HYBRID assessment. These tools have a very good balance of true positives and false positives alerts taking into account that:

- The application benchmark used is designed with the common vulnerabilities that DAST tools can detect.
- DAST tools have less false positives than SAST tools by design.

- IAST tools have also less false positives because it performs more precise runtime whitebox analysis based on real variable and process information.
- HYBRID tools eliminate more false positives by correlating the individual components or leveraging the feedback that individual components supply each others.
- HYBRID tools detect more true positives adding the individual capabilities of individual components.

Table 41 (section 5.6) shows the very good values for precision and F-measure metrics. Eight tools have a F-measure value higher than 0,817.

6.2. RESEARCH QUESTION 2: Which is the usability level of the tools?

For the case of SAST tools assessment for C/C++ applications, *SCA, Prevent, K8-insight, Cx-enterprise* and *Codesonar* are easy to install, use, and upgrade. They have a great amount of user documentation and give the possibility to developers of learning quickly with their interface help. This interface is designed also to make the audit more easily with its help trace of warnings detections to investigate in the code a concrete vulnerability. The background and code knowledge provided by these tools help the developer to learn quickly. *Goanna* must improve its trace warnings help and the other analyzed tools do not have these features, therefore the audit of a report scan will be always a manual code review.

SCA, Codesonar and *Cx-enterprise* give the option of reporting vulnerabilities in different standards formats, as those of OWASP TOP 10, SANS TOP 25 or MITRE CWE. *Prevent, K8-Insight* and *Goanna* have online documentation about vulnerabilities correspondence

with MITRE CWE standard. *Pc-lint, Satabs and CBMC* has no reports in OWASP, CWE or TOP SANS 25 standards formats and they are more difficult to use. *Satabs* and *CBMC* has no support to compile large projects, give only basic information about detected vulnerabilities, has no trace help for eliminating false positives and for each detected vulnerability, only report its type and code line where it occurs.

SAST for web applications assessment. *HP-Fortify SCA* and *K8-Insight* usability skills have been commented in the previous paragraph of SAST tools assessment for C/C++ applications.

Checkmarx Cx-Enterprise permits to select a set of vulnerabilities for an adapted analysis. It presents complete reports that can be correlated with OWASP TOP TEN or SANS TOP 25, classifying the vulnerabilities in 3 degrees of dangerousness. The trace information for the audits is adequate, with flow graphs to better understand the vulnerability occurrence. It does not permit the addition of rules for new vulnerabilities by the users.

Veracode SaaS classifies the vulnerabilities in 5 dangerousness degrees and supplies good flow graphs and trace information to audit the vulnerabilities.

Lapse+ and *findbugs* do not have acceptable trace information to audit the vulnerabilities and do not classify the vulnerabilities by degrees of dangerousness.

DAST, IAST, HYBRID assessment. Analyzing these types of tools we can classify their usability skills in two groups: commercial and open source tools. It is important to see that DAST open source tools have a higher level of performance (detection ratio, false positive ratio and vulnerability coverage), functionality and **usability** than SAST tools when comparing with DAST and SAST commercial tools respectively.

However, the usability skills of commercial tools for performing analysis are also better than the ones of open source tools:

- They incorporate more individual utilities to perform manual analysis to check the veracity of alerts reported by an automatic analysis. For example Acunetix, Appscan and HP-Webinspect have tools as HTTP editors, HTTP Sniffers, HTTP Fuzzers and Authentication Testers to perform manual checks.
- Their interface is friendlier and the reports are more complete and easier to understand and classify the vulnerabilities.
- Their analysis speed is higher to analyze large projects.

Seeker IAST tool has a very good interface with many information about vulnerabilities found based on the code. It provides videos explaining how many of the vulnerabilities found can be exploited based on the analysis performed of a web application.

HP-Fortiy hybrid tool has an audit tool (*auditworkbench*) that allows correlating the results of individual tools components. Figure 41 shows an example of a vulnerability correlation found by HP-Fortify SCA and Securityscope tools. The correlation function allows the fusion of the results to optimize the global results.

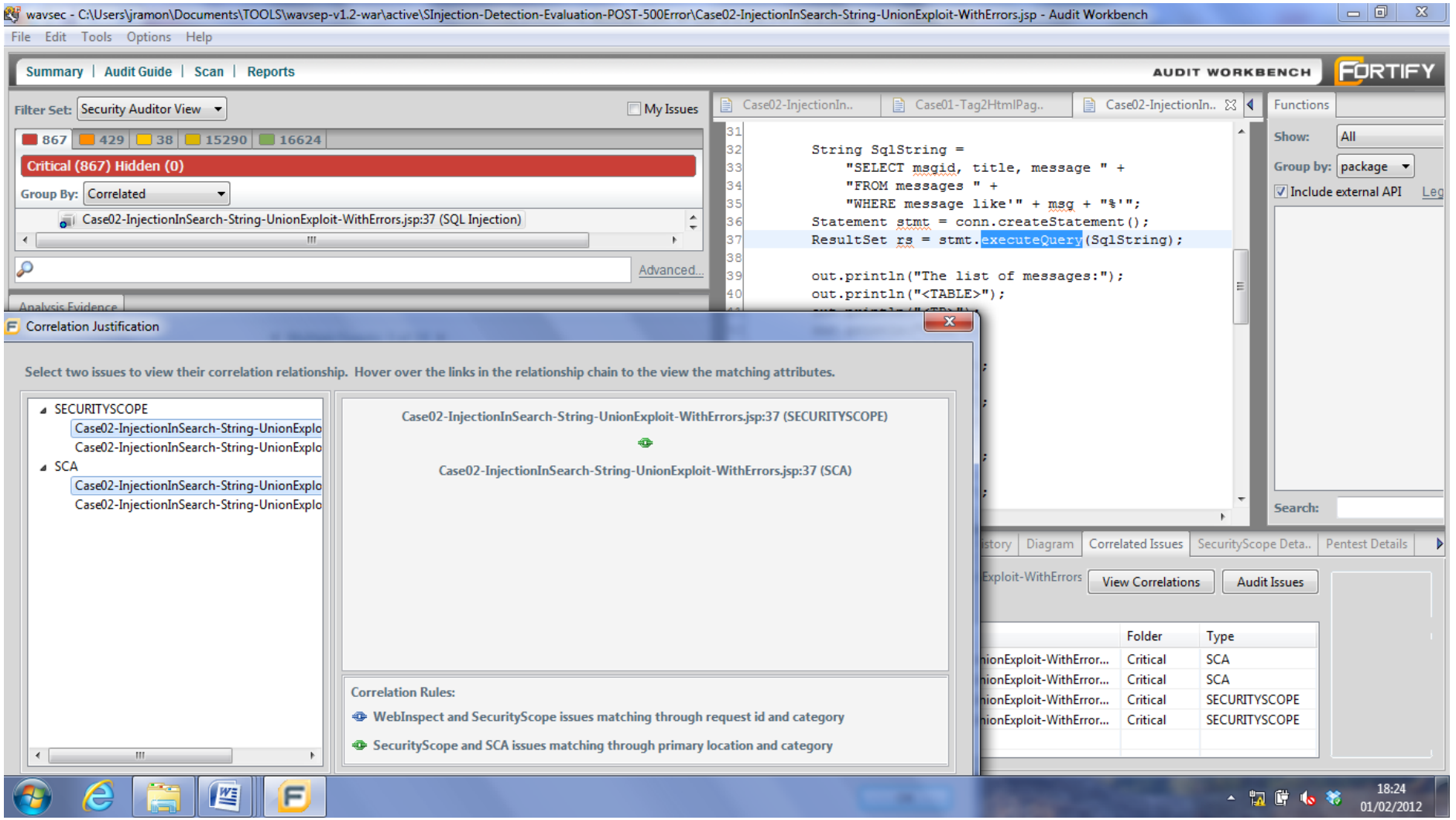

Figure 41. Vulnerability correlation with auditworkbench.

6.3. RESEARCH QUESTION 3: Which is the degree of adequacy of the selected benchmarks within the proposed methodology?

SAST for C/C++ applications assessment. A questionable restriction of our study is the selection of the SAMATE test suites as a benchmark. We believe the vulnerabilities selected by NIST [NIST, 2013], the common used format, the CWE of MITRE [Mitre, 2013], and the code complexity variations of Kratkiewicz [Kratkiewicz, 2005] added are a wise selection. However, as mentioned at chapter 5, eight test cases in test suite 46, designed for evaluating false positives, must be changed.

Also, SAMATE is not really a standard and some other groups could use their own schemes to do a similar evaluation. SAMATE defines only minimum functional requirements of tools. These requirements could be upgraded to include additional ones, as more languages support, reporting capabilities, product signatures updates**,** the information provided around a finding (explanation of the vulnerability, recommendations, accuracy level) and the relevance of the actual finding. Other potential improvements are adding the ability to merge assessments, the ability to diff assessments or remediation advise customization.

The main contribution of SAMATE in our opinion is its great number of test suites available for several languages and test cases with code complexity variety.

SAST for web applications assessment. The benchmark used, SAMATE Juliet 2010 is a very complete test suite with more than 13000 test cases for J2EE web applications, with a wide vulnerabilities coverage (106 CWE vulnerabilities) and a great variety of *source inputs* (such as *tcpip connections, console input, database, file, cookies, requests input parameters,* etc.) and *code complexity* (see section 5.5.1) [Kratkiewicz, 2005]. We think SAMATE Juliet 2010 is the most adequate benchmark to analyze the security of a web

application and meets all requisites that a benchmark must have, according to Gray [Gray, 1993], that the benchmark must be ‘‘repeatable, portable, scalable, representative, requires minimum changes in the target tools and simple to use’’. Actually SAMATE has delivered the new version SAMATE Juliet 2013 for J2EE and C/C++ languages with more vulnerabilities coverage.

DAST, IAST, HYBRID assessment. The selected benchmark WAVSEP, as analyzed in section 4.8.1, is an application benchmark with a test suite composed of 1126 different test cases for vulnerabilities detection checks and 31 test cases for false positive checks. The vulnerabilities categories of this application benchmark are between the most dangerous and frequents according to Veracode report volume 5 [Veracode, 2012]. In this report, XSS, SQLI, RFI and LFI vulnerabilities are the 65% percent of total vulnerabilities found in all applications analyzed. WAVSEP is a web application benchmark and it is the best suitable for the assessment of DAST tools due to the vulnerabilities categories of WAVSEP that can be detected by DAST tools.

Also WAVSEP is adequate to test IAST and HYBRID tools to compare their results with DAST ones. However, in our opinion, WAVSEP should be improved with more test cases corresponding to additional vulnerabilities categories. A new version of WAVSEP has been delivered in the beginning of 2014 with new test cases for open redirect and Old, Backup and Unreferenced Files [OWASP, 2013] vulnerabilities. Old, Backup and Unreferenced Files vulnerabilities permit to find unreferenced and/or forgotten files that can be used to obtain important information about either the infrastructure or the credentials. Most common scenarios include the presence of renamed old versions of modified files, inclusion files that are loaded into the language of choice and can be downloaded as source, or even automatic or manual backups in form of compressed archives. Backup files can also be generated automatically by the underlying filesytem your application is hosted on, a feature

usually referred to as "snapshots". All these files may grant the pentester access to inner workings, backdoors, administrative interfaces, or even credentials to connect to the administrative interface or the database server.

6.4. RESEARCH QUESTION 4: How static, dynamic and hybrid tools must be integrated in SSDLC?

The SSDLC models analyzed in section 4.2 incorporate SAST and DAST tools in the implementation and test phases respectively. However, actually IAST or HYBRID tools are not mentioned in these models. **As we have shown in previous section 5.6, there are IAST and HYBRID tools implementations that can be considered to be introduced in a SSDLC process.** These tools get the best score in the comparative assessment performed in section 5.6 with DAST tools for penetration testing of web applications. Seeker and HP Fortify Hybrid obtain the best results in the assessment.

Regarding to **HP Fortify Hybrid**, the SAST tool component (HP Fortify SCA) finds additional true positives to the findings of DAST component (HP Webinspect). Their correlated true positive results are better than their individual ones. IAST (Securityscope) tool component of HP Fortify Hybrid confirms many of the findings of DAST and SAST components (HP Webinspect and HP-Fortify hybrid). In HP Fortify Hybrid, IAST is installed between DASD (HP Webinspect) and the application to confirm in runtime analysis the findings of DAST. When correlating the results of SAST-DAST-IAST components of HP-Fortify hybrid tool:

- IAST Securityscope confirms the 60% of DAST findings for XSS and SQLI vulnerabilities but does not confirm any of LFI and RFI vulnerabilities. As

Securityscope depends on HP-Webinspect attacks, it cannot find additional vulnerabilities.

- The SAST component (HP-Fortify SCA) confirms 100% of DAST component (HP-Webinspect) findings for XSS and SQLI vulnerabilities and 10% of findings for RFI and LFI vulnerabilities.
- SCA finds additional vulnerabilities for SQLI (1), LFI (40) and RFI (19) (see table 39).

It is very improbable that vulnerabilities found by the three components of HP-Fortify hybrid tool be false positives, mainly because Securityscope confirms the vulnerabilities in runtime analyzing variables in the real process environment.

HYBRID tools can be used together in test phase analyzing the application by correlating the results obtained by each different tool. Also HYBRID tools could integrate several types of analysis leveraging the synergies between different types of tools without necessity of correlating results because it presents a integrated result (See section 4.6 for PHP Vulnerability Hunter tool by example).

With respect to **IAST tools**, they should be incorporated at test or deployment phases depending on their work mode. By example Seeker tool or Fortify Securityscope (see section 4.6) must be used only in test phase because they perform a white box analysis of the application with the objective of reporting the vulnerabilities they find. Other tool as Fortify RTA (see section 4.6) can be used as a firewall in deployment phase because it can block an attack attempt exploiting a vulnerability in runtime.

Figure 42 represents where in a SSDLC each type of tool can be used in order to incorporate new tendencies in white box, black box and HYBRID analysis tools. Based on this framework, an organization must choose the most appropriate tools according to the characteristics of their applications, their development policy, security policy and their code

revision process. Following a SSDLC model (section 4.2) an organization should have adequate type tools for each phase of SSDLC. At least it should have a SAST tool and a DAST and/or IAST tool.

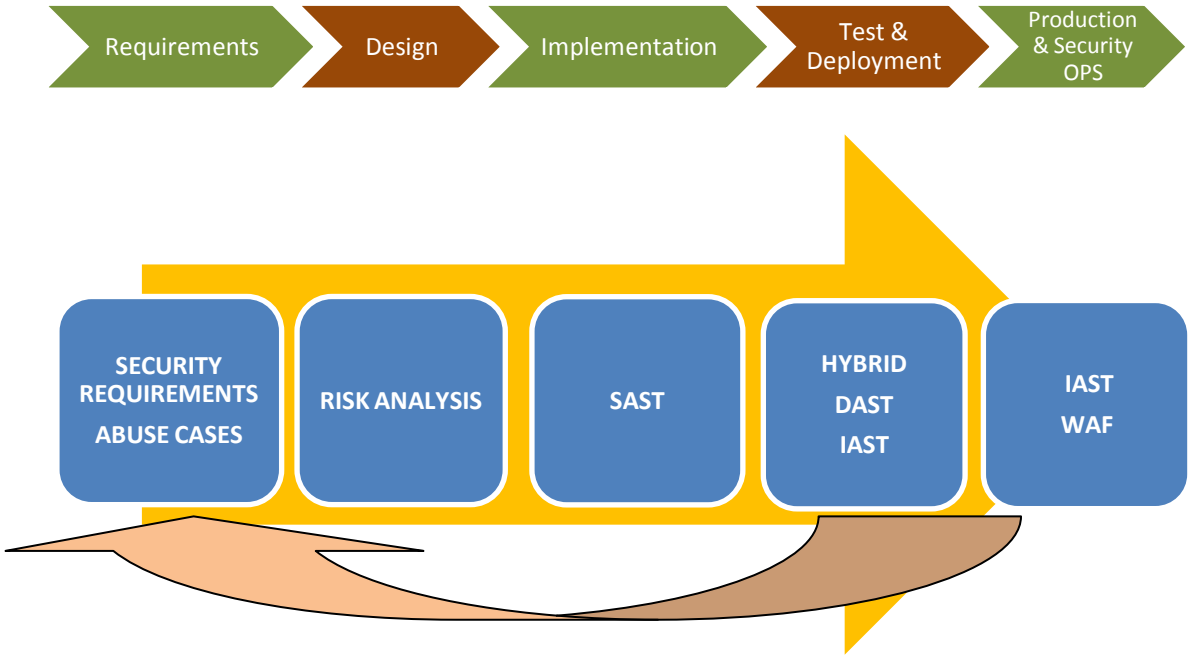

Figure 42. Security tools integration in a Secure Software Development Life cycle.

Actually, new HYBRID tools solutions of various types are appearing while the debate between static and dynamic tools continues. Therefore, it is necessary to continue studying and investigating SAST, DAST, IAST and HYBRID tools possibilities to determinate adequate conclusions about:

1. Languages coverage, vulnerabilities coverage, false positives and false negatives ratios and other similar complementary metrics obtained from available and public assessments or performing the assessment by the organization.
2. Usability, knowledge degree of the tool required and vulnerability trace facilities of the vulnerabilities found to aid eliminating false positives.

3. Phase of development life cycle to apply the tool according to its type and purpose. The type of tools determines the phase or phases where it can be used.
4. Adequate knowledge (web weakness, secure languages considerations…) of developing personal to handle the tools and perform the audit and vulnerability correlations of the application analysis.
5. The economic possibilities of acquiring the tools by a company. The organization must evaluate the availability and most adequate tools to be included in the SSDLC process according to the tool capabilities and economic possibilities.

6.5. Conclusions.

This chapter has discussed the answers to research questions that complement our study by using the results of the assessments of the different types of tools and benchmarks selected to perform a security analysis of an application. The research questions try to cover aspects of the tools as:

- **Balance** between vulnerabilities detection and false positive ratios.
- **Usability** and **reports**.
- Posterior **audit** of the tool reports.
- The integration of the different types of tools in the phases of the **SSDLC**.

The study of all these possibilities and characteristics for a security analysis tool is required to make the best election to evaluate the security of an application inside of the SSDLC.

7. RELATED WORKS

This chapter reviews and discusses previous relative assessments and comparative evaluations of SAST, DASD, IAST and HYBRID tools used to audit any type of applications. It also compares their results with those obtained in this dissertation. The section is organized in three subsections, according to the assessments performing in this work in sections 5.4, 5.5 and 5.6:

- **Works related with SAST assessments for C/C++ applications.**
- **Works related with SAST assessments for web applications.**
- **Works related with DAST, IAST and HYBRID assessments for web applications**.

7.1. SAST assessments for C/C++ applications related works.

The thesis of Kratkiewicz [Kratkiewicz, 2005] evaluates five tools against 291 buffer overflow test cases in C code. The details of this work shows that *Polyspace* [MathWorks, 2013], the only commercial tool in the comparison, obtains a detection rate of 99% (recall of 0.99) and a false alarm rate of 2.4% with a precision value of 0.976. The results, in the same work, for the open source tool *Archer* [Yichen, 2005], obtains a detection rate of 90% and a false alarm rate of 0,0%. The others tools (*UNO* [Holzmann, 2002], *Splint* [Evans, 2002] and *BOON* [Wagner, 2000]), obtain worse results. This work is also complete with respect to *code complexity* variety. The *code complexity* concept refers to variants in the code, which can lead to a concrete vulnerability: directly in main body, in a called function, in a loop, array, pointer, etc. **Tools sometimes may not detect a concrete vulnerability**

when the *code complexity* changes, therefore a benchmark should include code complexity diversity for each vulnerability.

However, the tool *Archer* got worse results with other vulnerabilities related with strings operations in C code, as demonstrated in another tool comparison results in [Zitser, 2004], also with *Polyspace, UNO, BOON* and *Splint* studies, These works compare open source tools with few exceptions. Moreover in many cases the available comparisons are on earlier open source tools, based only on lexical and syntactic analysis, with poor results and only searching for a reduced set of vulnerabilities.

The results in Emanuelsson and Nilsson [Emanuelsson, 2008] of a technical comparison of three commercials tools, *Polyspace, Prevent* [Coverity, 2013] and *K7 Insight* [Klocwork, 2013], indicate that *Prevent* and *K7 Insight* find largely disjoint sets of vulnerabilities. **This study concluded that *Prevent* and *K7 Insight* are prepared to sacrifice finding all vulnerabilities in favor of reducing the number of false positives while *PolySpace* has a high rate of *false positives*.** Results for *Prevent* and *K8-Insight* tools in our comparison (see section 5.4) are consistent with those in Emanuelsson and Nilsson [Emanuelsson, 2008], with older versions, showing a large percentage of disjoint found vulnerabilities. We agree with these authors in that, although the documentation offered by both companies claims the same mechanisms and objectives, there must be significant differences in their analysis algorithms.

A study by Hofer [Hofer, 2010] presents a comparison of 12 static analysis tools, eleven of them open source. The metrics in the study are: "installation process", "configuration", "support", "reports understand", "vulnerabilities coverage degree" and "support for handling projects". Although interesting, the study lacks of execution metrics that allow a correct comparison of the tools' performance detecting security vulnerabilities.

O.V. Pomorova and D.O. Ivanchyshyn [Pomorova, 2103] performed an assessment of several static analyzers for C/C++ code. They used SAMATE test suites and U.S. Department of Homeland Security test suites. It includes 23 test samples designed specifically for static tools testing as a benchmark and recall, precision and the F-measure metrics for analyzing the results. The tools analyzed were *CppCheck*, *PVS-Studio*, *Goanna* and *PC-Lint*. The tools were executed against three test suites:

1. 23 Homeland Security test cases.
2. 14 SAMATE test cases for buffer overflow and memory leak vulnerabilities.
3. 10 SAMATE test cases for null pointer dereference CWE 476.

Figure 43 shows the summary of results computing the F-measure metric (F-score).

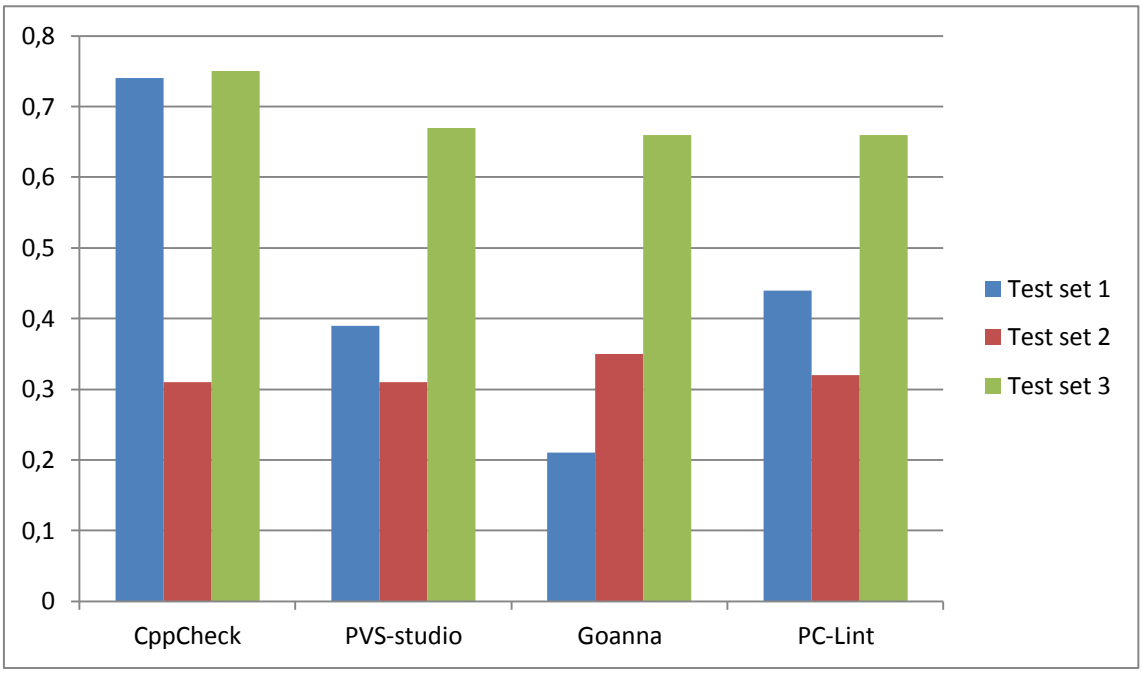

Figure 43. F-score metric results of Pomorova assessment of SCA tools [Pomorova, 2103]

In the case of PVS-Studio and Goanna Studio analyzers, the precision metric is equal to 1 for all test sets. Recall metric shows opposite results. It characterizes small percentage of defects detection. Figure 43 shows that F-measure changes depending on the test set. For

the first test set effectiveness of SAST tools analyzed varies significantly. CppCheck is the most effective in this case. Static analyzer allows to detect the vast majority of defects present in test samples and to generate small percentage of false positives. Values are similar for the second and third sets. The F-measure value is the same for three analyzers in test set 3 for F Null Pointer Dereference vulnerability. The result for CppCheck tool is slightly different. Thus, the metric F-measure reflects effectiveness of SCA. The conclusions of the authors were:

- The study found that the effectiveness of static analysis is a complex measure that depends on the complexity and implementation details of the testing source code.
- The value of the F-measure metric is useful for comparison of the results for only single target software.
- The high results for particular project do not guarantee a similar result for any other software.

In our opinion the study of Pomorova [Pomorova 2013] analyzes test cases for a reduced set of vulnerabilities to be properly representative. Three of the static tools are commercial and Cpp-check is an open source tool. The F-measure metric of the test set 3 (null derrefence CWE 476) is over 0,65 for the four tools in the comparison. The F-measure metric for the test set 2 is over 0,3 for the four tools. Cpp-check obtains the best result with a F-measure metric of 0,74. The F-measure results for test set 3 are similar to the one obtained in our assessment detailed in section 5.4. For example, the F-measure results for Goanna and PC-Lint are 0,67 in Pomorova comparison. In our assessment of section 5.4 Goanna and PC-Lint were also included and their F-measure results were 0,64 and 0,56 respectively. However the results of Pomorova comparison for test sets 1 and 2 are lower than the one of our comparison. *Cpp-check* is an open source tool that obtains good results when compared with commercial solutions as Goanna or PC-lint involved also in the SAST comparison of this thesis (see section 5.4). We think that more studies about commercial solutions are

necessary to understand all possibilities they offer. This study agreed with the Pomorova [Pomorava, 2103] conclusions.

7.2. SAST assessments for web applications related works.

The work of [Rutar et al, 2004] analyzed a number of open source SAST tools for java, (Findbugs, PMD, SC/java, JLINT and FLAG) and **their results show that the tools find non-overlapping vulnerabilities**. Authors proposed a meta-tool to allow developers to identify the different classes of vulnerabilities. The different vulnerabilities findings of the tools are confirmed by our comparison results of section 5.5. Also our investigations about SAST tools possibilities in section 5.5 confirm that only Findbugs, of all evaluated tools, is valid to analyze J2EE applications.

Wagner, Jrjens, Koller and Trischberger [Wagner, 2005] compared three different open source static analysis tools: *FindBugs*, *PMD* and *QJ Pro*. They were executed against five industrial projects and one development project from a university environment, which were in use or in the final testing phase. **Their results showed again very different results for the three tools.** Our assessment (see section 5.5) confirms that the SAST tools obtain also different results. Therefore SAST tools can be combined to obtain better results (see section 5.5).

The comparison of Secologic [SECOLOGIC 2006] is an assessment of open source static analysis tools for security testing of java web applications. It focuses on web application vulnerabilities. The most important conclusion is pointing that there is no open source tool that is involved in the comparative test with enough support to security. Of the tools analyzed, Findbugs is the best performing and would be the best choice for safety tests. **It highlights that commercial tools like HP-FORTIFY SCA, COVERITY Prevent [Coverity, 2013] available on the market, can provide better and more sophisticated possibilities about vulnerabilities detection, interfaces and reports.** Findbugs covers

only some of the most important major vulnerabilities of web applications, but gives the possibility to expand the development of new vulnerabilities detectors. Findbugs obtains bad results when comparing with the results obtained in our assessment of section 5.5. Findbugs is the tool with the worst result in the comparison of section 5.5.

Another comparison analyzed is that of Michael S. Ware, J. Christofher Fox [Ware et al, 2008] "*Securing Java Code: Heuristics and an Evaluation of Static Analysis Tools*". It concludes that, of the total of nine tools involved, only two are valid tools for J2EE web applications: Findbugs [Findbugs, 2013] and HP-FORTIFY SCA [HP-Fortify, 2013]. One interesting finding obtained is that, from the total of 115 different vulnerabilities (not J2EE), only 50 were identified by summing all the detections of the 9 tools. The best result is achieved by HP-FORTIFY SCA that identifies 27 vulnerabilities. Lastly the authors mention the need for a subsequent audit results. HP-FORTIFY SCA obtains good performance results in our comparison of section 5.5. It finds the 78,9% of the total vulnerabilities.

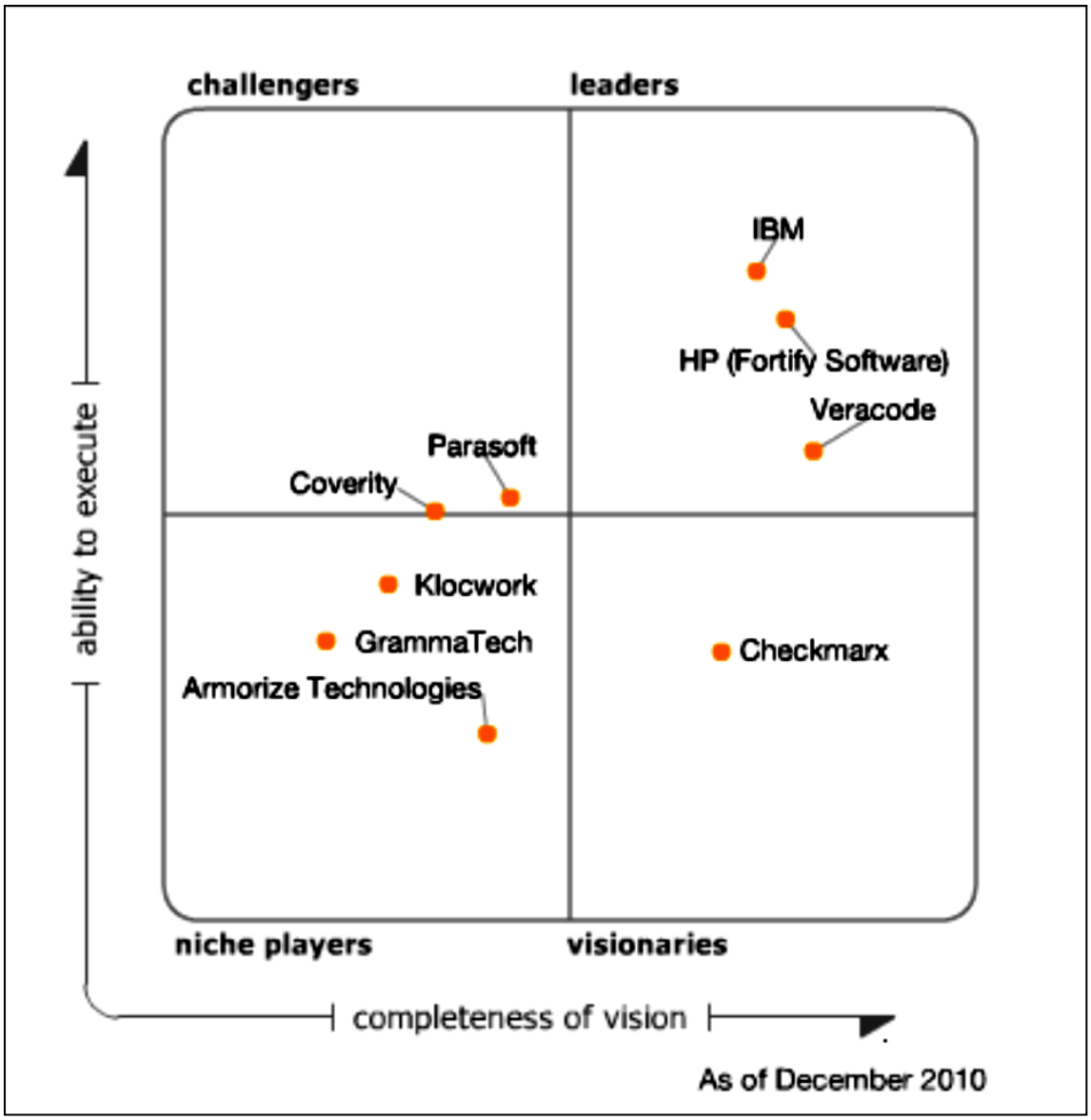

Figure 44. Gartner magic quadrant for static analysis [Gartner, 2010]

Gartner research, in its publication of December 13, 2010, Magic Quadrant for Static Application Security Testing, [Gartner, 2010], analyzes the market for static analysis of security according to their business vision and technology (see figure 44). The study claims that by 2015, over 60% of companies will use in their processes static analysis tools for application development. This report discusses advantages and disadvantages of the tools involved and examine them to have an approximation of their quality. Figure 44 shows the tools skills about completeness of vision and ability of executing. SAST commercial tools

from HP-FORTIFY, IBM, VERACODE or CHECKMARX have the best consideration. The tools of Gartner study, from HP-FORTIFY, VERACODE, CHECKMARX and KLOCWORK are also investigated in our comparison of section 5.5. Our results confirm the Gartner position of HP-FORTIFY, CHECKMARX and KLOCWORK about their performance (78,9%, 60%, 58,9 of detections respectively) and usability but VERACODE SaaS does not obtain good results (38% of detections) in our comparison of section 5.5.

Finally, the master thesis of Jayesh Shrestha **[**Shrestha, 2013] studies the state-of-art of open source static analysis tools. This study was focused to research on C/C++ and Java based static analyzers, which are open sourced. It also uses the recall and precision metrics (see section 5.3) and the SAMATE Juliet benchmark used in our assessment for SAST in web applications. In particular, Shrestha work studies Findbugs, analyzes the results and compares it with *Parasoft Jtest* commercial tool. The work shows that the overall performance of Findbugs found to be relative good enough in comparison to the result obtained from *Jtest* focused more in quality errors (see section 4.3.3). Findbugs detects a 20% of vulnerabilities and *Jtest* detects only a 0.05% of vulnerabilities in SAMATE Juliet test suites. But, since Findbugs generates high number of false positives, its performance can be questionable. Most often Findbugs was not able to distinguish the sanitized lines of code in the good methods and points to the same location where there is a flaw in the bad method. **It becomes crucial when true bugs get lost in the false positive.** So, there seems to be still a room for improvement with Findbugs. In fact, Findbugs was not found to be complete and sound tool. The results of Findbugs in Jayesh Shrestha [Shrestha, 2013] master thesis are consistent with the results obtained in this work, detailed in section 5.5.

Only three commercial tools (HP-FORTIFY SCA, Coverity Prevent and Parasoft Jtest) were included in the related comparisons examined. Regarding to SAST open source tools, the previous assessments confirm their poor results detecting web vulnerabilities.

The number of commercial SAST tools available for analyzing web applications is reduced, see WASC static analysis tools evaluation project [SAST-wasc, 2013] and table 18.

The related work examined does not analyze the SAST coverage degree of vulnerabilities categories according to vulnerabilities in tables 28 and 29 of section 5.5.1.

7.3. DAST, IAST and HYBRID assessments for web applications related works.

The Magic quadrant for application security testing [Gartner, 2013] shows a review of commercial solutions for SAST, DAST, IAST and HYBRID capabilities. The Evaluation Criteria is about two aspects:

- Ability to Execute:
    - Product/Service
    - Overall Viability (Business Unit, Financial, Strategy and Organization)
    - Sales Execution/Pricing
    - Market Responsiveness and Track Record
    - Customer Experience.

- Completeness of Vision:
    - Market Understanding
    - Sales Strategy
    - Offering (Product) Strategy
    - Innovation
    - Geographic Strategy

It also remarks the major trends shaping the market are summarized below.

- Expansion of Application Testing as a Service
- The Importance of Testing Client-Side Code (Rich Internet applications)
- The Importance of Testing Mobile Applications
- SDLC Integration
- The Importance of Comprehensive Application Discovery
- IAST
- Web application firewall integration

Figure 45 shows the security solutions position in a schema based in the topics analyzed mentioned above. Vendors of IAST products and subscription services were considered for this Magic Quadrant if their offerings:

- Provided a dedicated static or dynamic application security testing capability — a tool, subscription service or both
- Had at least $2 million in specific revenue from AST-related products or services
- Were generally available (not beta) before 1 January 2013
- Vendors must also be determined by Gartner to be significant players in the market, because of market presence or technology innovation.

Figure 45. Gartner magic quadrant for application security testing

The assessment of Nuno Antunes and Marco Vieira [Antunes, 2009] compares how effective are SAST and DAST tools on the detection of SQL Injection vulnerabilities in web services code. To understand the strengths and limitations of these techniques, they used several commercial and open source tools to detect vulnerabilities in a set of vulnerable web services. Results suggest that, in general, static code analyzers are able to detect more SQL Injection vulnerabilities than penetration testing tools. Another key observation is that tools implementing the same detection approach frequently detect different vulnerabilities. Finally, many tools provide a low coverage and a high false positives rate, making them a bad option for programmers. The SAST tools analyzed were FINDBUGS, YASCA and Intellij IDEA and the DAST tools were HP-Webinspect, IBM-appscan, Acunetix and a tool proposed by the authors. For the results presentation we have

decided not to mention the brand of the commercial scanners to assure neutrality and because licenses do not allow, in general, the publication of evaluation results. Figures 46 and 47 (VSx are DAST tools and SAx are SAST tools) compare the detection and the false positive ratios for the tools tested in the present work. As we can see, in general, the static code analyzers present better coverage results than the penetration testing tools. The only exception is SA3 that has a detection coverage lower than VS1. The other two static analyzers achieved a detection ratio much higher than any of the penetration testers. However, all the static code analyzers reported many more false positives than any of the penetration testing tools. The difference is in fact high (more than 10%), even if we compare the analyzers with VS1, which is the penetration-testing tool with higher rate of false positives (but it is also the one with higher coverage).

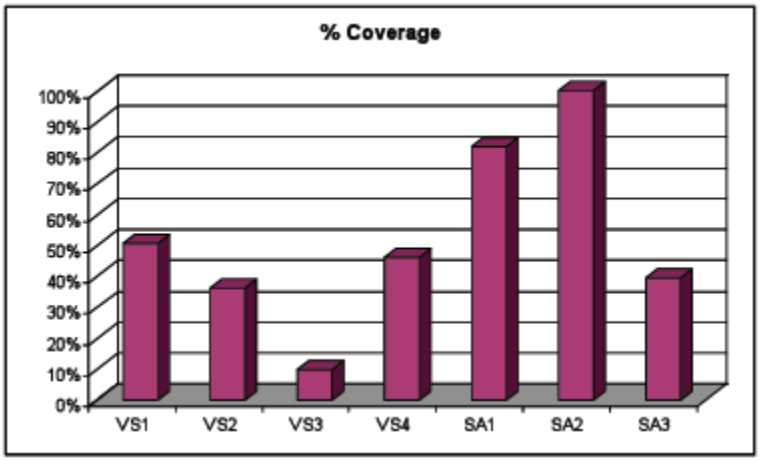

Figure. 46. Vulnerability detection percentage for Antunes comparison

[Antunes, 2009]

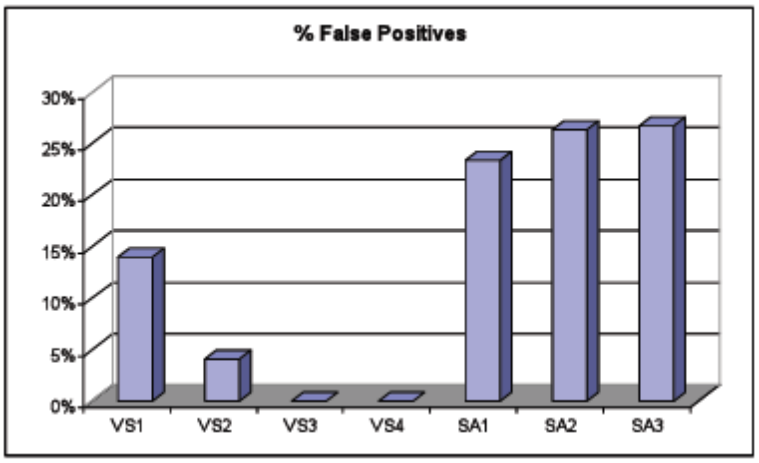

Figure 47. Vulnerability false positive percentage for Antunes comparison [Antunes, 2009]

In the paper "*Defending against vulnerabilities in web applications"* Nuno Antunes and Marco Vieira [Antunes, 2012] analyzes the SAST, DAST, IAST and HYBRID capabilities for web application security testing within a SSDLC and concluded that achieving better results and improved effectiveness requires new techniques to overcome the intrinsic limitations of vulnerability-detection tools. However, overcoming these limitations is not easy because it requires shifting from traditional approaches to disruptive methods. The key is to relax some constraints and combine different methods to overcome individual limitations. The results of Nuno Antunes and Marco Vieira [Antunes, 2009], [Antunes, 2011] and Web Application Security Statistics project 2008, [Wasc-Statistics, 2008] confirm that SAST tools obtain better detection results and higher false positive rates than DAST tools. White box analysis (static analysis with audit) can detect 91% of vulnerabilities per web application and black box only 3% according WASC statistics project.

AnantaSec perform an interesting comparison with DAST and HIBRID tools [AnantaSec, 2009]. The tools included in this report were:

- Acunetix WVS 6.0 (Build 20081217) (DAST)

- Acunetix WVS 6.0 (Build 20081217) (DAST) + acusensor (IAST)
- IBM Rational AppScan 7.7.620 Service Pack 2 (DAST)
- HP WebInspect 7.7.869 (DAST)

The testing procedure was testing 13 web applications as benchmark (some of them containing a lot of vulnerabilities), 3 demo applications provided by the vendors (testphp.acunetix.com, demo.testfire.net, zero.webappsecurity.com) and also some tests were done to verify Javascript execution capabilities. In total, 16 applications were tested. The goal was to cover all the major platforms with applications in PHP, ASP, ASP.NET and Java. The report included vulnerabilities like SQL injection, Local/Remote File Inclusion and XSS. The results are shown in figure 48. Acunetix+acusensor DAST-IAST hybrid tool obtains the best result in the comparison with better performance in 7 of 12 applications tested.

Best scores / application						
Nr.	**Tested application**	**Platform**	**AppScan**	**WebInspect**	**Acunetix**	**Acunetix + AcuSensor**
1	Javascript tests	N/A		✓		
2	Vanilla-1.1.4	PHP				✓
3	VivvoCMS-3.4	PHP				✓
4	fttss-2.0	PHP				✓
5	Wordpress-2.6.5	PHP	No clear winner			
6	vbulletin_v3.6.8	PHP	No clear winner			
7	riotpix v0.61	PHP				✓
8	javabb_v0.99	Java			✓	
9	Yazd Discussion Forum_v3.0	Java	✓	✓		
10	pebble_v2.3.1	Java	No clear winner			
11	TriptychBlog_v.9.0	ASP.NET	No clear winner			
12	DMG Forums_v3.1	ASP.NET				✓
13	Dave's CMS_v2.0.2	ASP.NET	No clear winner			
14	Acunetix Demo Application - Acunetix Acuart	PHP				✓
15	AppScan Demo Application - Altoro Mutual	ASP.NET	✓			
16	WebInspect Demo Application - free Bank online	ASP		✓		
	Summary		**2 wins**	**3 wins**	**7 wins**	

Figure 48. AnantaSec comparison results [AnantaSec, 2009]

The increased accuracy is achieved by combining black box scanning techniques with feedback from sensors placed inside the source code while the source code is executed. Black box scanning does not know how the application reacts and source code analyzers do not understand how the application will behave while it is being attacked. Therefore combining these techniques together achieves more relevant results than using source code analyzers and black box scanning independently. The DAST, IAST, HYBRID assessment of this work (see section 5.6) results confirms that HYBRID tools can obtain better results by leverage the individual capabilities and synergies of members tools.

Another interesting comparative about DAST by *Larry Suto* [Suto, 2010] shows the capabilities of seven commercial important tools. This paper is intended as a follow-on study to the October 2007 study of the same author, "Analyzing the Effectiveness and Coverage of Web Application Security Scanners." The execution against benchmarks are in mode "point and shoot" with no any configuration and "trained" with some configuration providing known users. This paper focuses on the accuracy and time needed to run, review and supplement the results of the web application scanners (Accunetix, Appscan by IBM, BurpSuitePro, Hailstorm by Cenzic, WebInspect by HP, NTOSpider by NT OBJECTives) as well as the Qualys managed scanning service. NTOSpider found over twice as many vulnerabilities as the average competitor having a 94% accuracy rating, with Hailstorm having the second best rating of 62%, but only after additional training. Appscan had the second best 'Point and Shoot' rating of 55% and the rest averaged 39%. One of the most surprising results was the findings for market share leader WebInspect, which consistently landed at the bottom of the pack in its ability to crawl the sites and find vulnerabilities; it missed approximately half of the vulnerabilities on its own test site. Figures 49 and 50 show the results of the test execution.

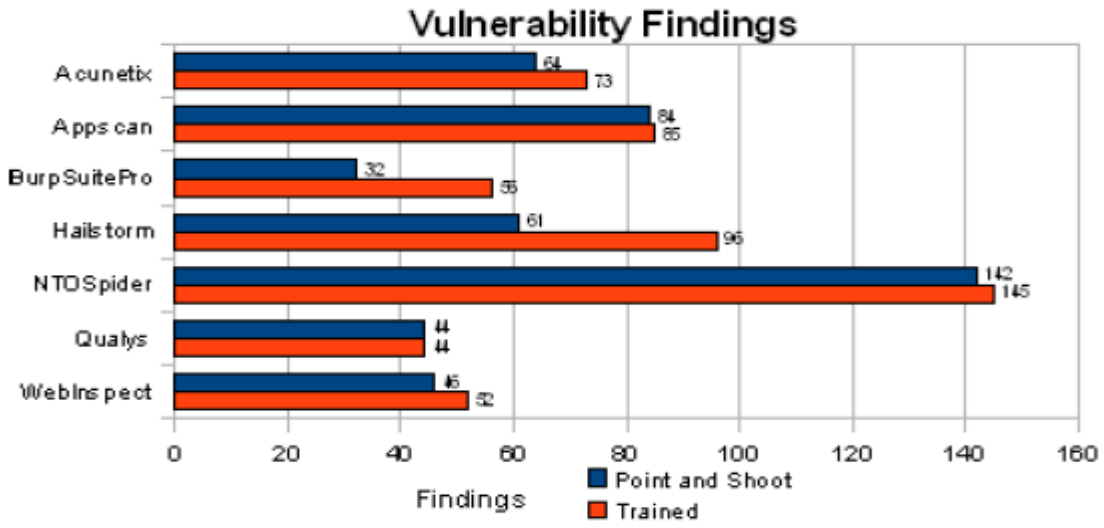

Figure 49. DAST tools comparison detection results [Suto, 2010]

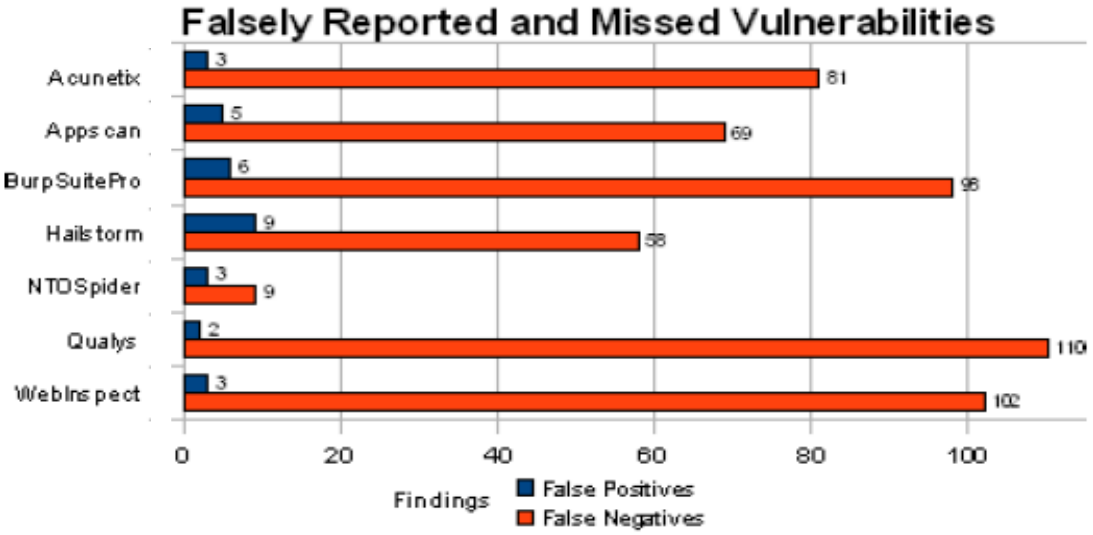

Figure 50. DAST tools comparison false positive/negative results [Suto, 2010]

The application benchmarks selected in Larry Suto comparison [Suto, 2010] contains vulnerabilities types that can be detected by DAST tools. Most of the vulnerabilities categories in the applications are *SQLI, XSS.* The other vulnerabilities are *HTTP response splitting, local file inclusion, OS command injection, XPATH injection* and *remote file include.* NTObjectives had a result of 92% of detections. The other tools detection results are about 40%. The assessment confirms that the false positive ratio of the tools is low. The

worst false positive ratio was 5,8% for Hailstorm tool. The low false positive ratios are according the assessment of this work performed in section 5.6. The true positive ratio for Acunetix, Appscan and Webinspect are worse than the one obtained by these tools in our comparison (section 5.6) where the tools have an additional average of 35% of true positives. It is important to see that the benchmarks used are not the same and there are some differences in the vulnerabilities that they take into account. Anyway, our comparison, performed two and a half years later, suggests that maybe the tools have improved their detection performance

Adam Doupe and Marco Cova [Bau, 2010] present the evaluation of eleven black-box web vulnerability scanners. The results of the evaluation clearly show that the ability to crawl a web application and reach "deep" into the application's resources is as important as the ability to detect the vulnerabilities themselves. It is also clear that although techniques to detect certain kinds of vulnerabilities are well-established and seem to work reliably, there are whole classes of vulnerabilities that are not well-understood and cannot be detected by the state-of-the-art scanners. **They found that eight out of sixteen vulnerabilities were not detected by *any* of the scanners.** The vulnerabilities not detected are *session fixation, parameter manipulation, stored SQL injection, logic flaws, bypassing the authentication logic, directory traversal, design vulnerabilities, weak password* and *stored XSS*. They have also found areas that require further research so that web application vulnerability scanners can improve their detection of vulnerabilities. Deep crawling is vital to discover all vulnerabilities in an application. Improved reverse engineering is necessary to keep track of the state of the application, which can enable automated detection of complex vulnerabilities.

The Sectoolmarket web site [Sectoolmarket, 2014] shows price and feature and performance data of many commercial and open source Web Application Scanners (DAST). The current information is updated on 06/02/2014 and sorted in an ascending

order according to the scanner audit features, various prices, benchmark results with Wavsep application (version 1.5) [Wavsep, 2014]. The prices presented might be different in reality due to special offers, bundles, discounts, negotiations or other reasons.

7.4. Conclusions.

This section is a review of related works about assessments and evaluation of the types of automatic security tools to perform an analysis of an application.

The review confirms that:

- There are yet few tools available of IAST and HYBRID types. Therefore there are yet few comparative assessments with IAST and HYBRID tools.
- There are more comparative assessments with SAST and DAST tools.
- There are more SAST tools implementations for C/C++ languages than for web applications.
- The comparisons analyzed about SAST tools for C/C++ applications are mainly for open source tools. However, our comparison in 5.4 section, mainly focused in leaders commercials tools,confirms their better results.
- The comparisons analized about SAST tools for web applications confirm the bad results obtained by open source tools. Our comparison in section 5.5 confirms the bad results for open source tools and includes commercial tools that obtain better results in true positive detections but they generally obtain worse false positive results that the one obtained by SAST tools comparison for C/C++ applications (section 5.4).
- Regarding to DAST, the related works revised confirm that these tools are capable of detect a concrete set of vulnerabilities due to these tools only perform a syntactic analysis of the applications (see section 4.4.1). However these sets of vulnerabilities are among the most dangerous ones, according to standards as SANS TOP 25:

- **Reflected XSS**
- **SQL injection**
- **Path traversal**
- **Command injection**
- **Directory listings**
- **Frame injection**
- **Backup files discovery**

Because of these scanners perform syntactic parsing of the web application, they cannot understand the semantics of various parameters as a whole that can hide an attempted attack. Therefore it is difficult the detection of other vulnerabilities as:

- **Broken access controls**,
- **Parameter manipulation**
- **Logic flaws**
- **Vulnerabilities in the design**.
- **Session hijacking**
- **Leakage of sensitive information**

- The comparisons about DAST tools analyzed and our DAST assessment of section 5.6 confirm that the ratio of false positive is lower than the one of SAST tools. All the studies, including our study, confirm that SAST tools find more true positives than DAST tools.
- The only HYBRID tool analyzed in Anantasec comparison [AnantaSec, 2009], Acunetix+Acusensor obtains the best result than the other DAST tools in the comparison.

- The HYBRID tool HP-FORTIFY Hybrid obtains also better results than the all DAST tools analyzed test in our comparison of section 5.6. It also includes an IAST tool, Seeker, that obtains the best results in the comparison.

8. CONCLUSIONS AND FUTURE WORKS

As in many other real life situations, the most desirable mean to avoid vulnerabilities in applications code is prevention. Developers should have been trained in security programming to avoid making "mistakes" involving programming vulnerabilities. Even when a very good training of programmers exists, there will always be vulnerabilities in the code and it will be difficult finding these vulnerabilities once the first version of the application is developed. Software engineers must consider a variety of strategies to build secure software before release. Achieving this goal is only possible by using various techniques and automatic tools to ensure security in all phases of SSDLC.

This dissertation has developed a repeatable methodology to evaluate the performance detecting security vulnerabilities of automatic security analysis tools. This methodology allows to accomplish several different assessments of the different types of available automatic security analysis tools. This final chapter summarizes the results of our research, highlights the main contributions and briefly describes some other areas that deserve future research.

8.1. Research summary.

This dissertation examines the state of art of last tendencies in applications development and architectures, applications security problems, assessment methodologies of security analysis tools and benchmarks for testing. It also examines the state of the art of all automatic security analysis tools categories available to perform a security process in a SSDLC:

- Static Application Security Testing (SAST). White box tools that perform a static analysis of source or executable code of the application.

- Dynamic Application Security Testing (DAST). Black box tools that perform a dynamic analysis of the application.
- Real time Application Security Testing (RAST) or Interactive Application Security Testing (IAST). White box tools that perform a runtime analysis of the application.
- Hybrid tools SAST-DAST, SAST-DAST-RAST, SAST-RAST, and DAST-RAST.
- How all types of tools can integrate in the SSDLC.

This work has conducted also several studies to analyze and define the real capabilities of the security analysis tools types to perform an efficient and complete vulnerability analysis of an application. The study is based on three comparative assessments:

1. Static analysis security tools (SAST, static white box analysis) for C/C++ applications.

2. Static analysis security tools (SAST, static white box analysis) for J2EE web applications.

3. Dynamic analysis security tools (DAST, Black box analysis), Interactive analysis security tools (IAST, runtime white box analysis) and Hybrid security tools (HYBRID).

The assessments show that all types of tools evaluated must improve the detection, false positive and vulnerability coverage ratios. These assessments allow also establishing a strict rank between the tools involved in each evaluation, according to their performance, showing their capabilities about additional features and vulnerability coverage degree. The performance degree of tools is calculated with F-measure, recall and precision metrics obtained from execution results against selected benchmarks.

Besides, we have used the results of the assessments to study how the tools can better be integrated in a Secure Software Development Life Cycle.

8.2. Assessment Methodology.

The developed methodology applies widely known metrics, based on rates of true and false positives, and vulnerabilities coverage degree of tools, producing a strict scale for the performance of the different types of analysis tools. Then, a company can choose a tool by analyzing the precision, recall, F-measure and vulnerabilities coverage metrics obtained against selected benchmarks for each assessment in the study.

A common conclusion for all types of semiautomatic tools analyzed in this work is that they **requires a posterior auditory to confirm all vulnerabilities alerts and classify them as true or false positives**.

8.3. Conclusions of SAST assessment for C/C++ applications.

The study provides objective evidence of the performance of static analysis tools, using a well defined benchmark test suite and a repeatable methodology, and provides results from the analyzed state-of-the-art tools.

The present study demonstrates objectively that some commercial tools (*Prevent, SCA, K8-Insight, Cx-enterprise and Codesonar*) show a better performance, usability and vulnerabilities coverage than the other analyzed tools. However, all the analyzed tools obtain different results for different types of vulnerabilities and cover different subsets of them.

The vulnerability coverage of tools must be improved to detect important and dangerous vulnerabilities of SAMATE test suites 45 and 46, based on the vulnerabilities of table 20 [NIST268, 2007], and the included in OWASP TOP TEN 2013 and SANS TOP 25.

A simple execution of many of these tools against a piece of code is not enough to get reliable results, and raw results (results from the first execution) must be reviewed. Of course the automatic execution of tools gives a formidable first step, especially for analyzing lengthy code, but this is not enough. A careful analysis of the results by an experienced user or team (with security skills and experience in the language used in the target code) is always necessary.

We strongly agree that the use of tools for static source code analysis to search security vulnerabilities must be integrated as a part of the security policy of any development organization. But current state of these tools does not allow indistinctly using them. The tools' internal designs and reporting output formats are different, so they produce substantially different results.

8.4. Conclusions of SAST assessment for J2EE web applications.

Web applications can have all the vulnerabilities that can have C/C++ applications and other specific vulnerabilities as XSS, CSRF, CLRF, etc. These additional vulnerabilities make web applications and web services more dangerous than other applications types.

Very high vulnerability detections percentages are achieved by some of the analyzed tools, reaching in some cases to exceed 80% in isolation. The average ratio of precision for all analyzed tools is 0.571 and the average ratio of recall is 0,545. The analyzed tools cover the most dangerous vulnerabilities, but they do not cover other also important vulnerabilities

categories (See table 32). The number of false positives, high in general, in four cases over 50%, must be reduced in a subsequent audit of the results**. Combining two or more tools may improve the outcome of total detections** reaching more than 92% (142 detections out of 147)

It is also important to consider the possibility of using a SAST for executable code. A company has not the source code for commercial web applications and the average percentage of commercial software is about 22% and almost 75% of them do not have acceptable conditions of security [Veracode 2012].

The degree of vulnerabilities coverage is analyzed in the second group test suite (see table 29). The vulnerabilities included in this group are less frequent and dangerous than the vulnerabilities included in the first group. The vulnerability coverage is bad for open source tools, which focus on the most frequent vulnerabilities of the first group (table 28) with the exception of some information disclosure vulnerabilities. The degree of coverage is higher for commercial tools (Fortify SCA covers all categories of vulnerabilities). It is quite acceptable, covering the most frequent and important ones. However, almost all of these tools can yet improve very much their performance. In general, the detections in the categories of vulnerabilities of the second group are related to disclosure of information in the code, weak cryptographic protocols and weak random numbers. All tools obtain worse ranking results in second group than in the first group with 32 of 62 potential vulnerabilities not detected by any tool. The changing nature and evolution of the categories of vulnerabilities over time, requires a continuous study to adapt the tools to this development to have them always adapted to the time of use.

The results of SAST assessment for web applications show that the tools analyzing web applications have a higher ratio of false positives than the tools analyzed in the

comparison of SAST tools for C/C++ applications. This is even more relevant if we see that two of the tools (HP-Fortify SCA and K8-Insight) were in both assessments.

8.5. SAST tools recommendations.

As a consequence of the analysis of the results in this study, we propose some recommendations for improving the use of static analysis tools*:*

- Their rates of true and false positives, and their balance, must be improved.
- The set of vulnerabilities and languages coverage must be increased, to allow accomplishing a thorough security analysis.
- If it is needed to select only one tool for a project, a good criterion is attending to its classification: "security review", "bug finding" and "program verification". As demonstrated by the results of this work, "security review" tools have less false negatives with more false positives than "bug finding" or "program verification" ones. A company should also take into account the time needed for performing the report audit of a static tool.
- The use of several tools with different designs and with different detection algorithms/heuristics can improve the analysis results when making a real analysis of a big project. However, taking into account the high prices of some of the commercial tools, this solution will be valid only for professional development teams, but not for individual users.
- The use of SAMATE tests as a benchmark for objectively evaluating and comparing the performance of static source code security analyzers should be promoted. When deciding which tool to use for a particular project, SAMATE can be used as a reliable baseline for comparing tools. However, SAMATE can be improved by adding new test suites, with new vulnerabilities, with more code complexity variety.

The functional requirements document could also be improved to cover more requirements.

- The definition of a common output format for the results of the execution of the tools, based on a comprehensive set of standard vulnerabilities, will allow better and more reliable comparisons of static analysis tools. Another helpful idea is to add result modes that normalize the number of reports for each fundamental problem. For instance, if a buffer overflow occurs in a function, all tools could report that as a single problem, instead of listing each vulnerable call as a separate result.
- More studies are needed to compare static analysis tools as one of the foundations for consolidating a reliable industry. We hope that our study will help improve further similar studies.

Some of these recommendations are complementary and closely related with those of the two "Static Analysis Tool Exposition (3)" workshops [NIST297, 2012] conducted by the NIST SAMATE project. In these workshops participating tool makers ran their tools on a set of selected programs written in C and Java languages, including only a small number of SAMATE tests. Researchers led by NIST performed a partial analysis of tool reports.

Many efforts must be still done by software industry to obtain a reliable index of the trustworthiness of software. Some of these efforts are promising as, for example, a work based on SAMATE initiative, (NIST Interagency Report 7755, 2010) [NIST7755, 2010] that proposes a preliminary framework for assessing the trustworthiness of software.

8.6. Conclusions of DAST, IAST and HYBRID assessment for J2EE web applications.

Taking into account that WAVSEP benchmark is designed with vulnerabilities that DAST tools detect by design, the F-measure score ranks the **performance of tools** (DAST, IAST

and HYBRID) and is generally good for almost all tools selected. All tools except for W3AF have a recall metric higher than 0,772. Seeker IAST tool and HP-Fortify hybrid have the best rank in the comparison. DAST, IAST and HYBRID tools have less false positives than SAST tools. IAST tools perform runtime analysis examining the process environment and have very few false positives. Seeker is a IAST tool that obtains excellent results in WAVSEP assessment (see section 5.6). HYBRID tools correlation of the results of individual tools components also permits reducing the false positive ratio. DAST tools have more false positive ratio than IAST and HYBRID tools.

Different approximations of HYBRID tools are appearing and we think they are yet in the beginning of their development.

Commercial DAST tools have obtained better F-measure score than open source tools. The analysis performed against WAVSEP benchmark shows that crawling capacity to find the complete structure and links of a web application of commercial DAST tools is higher than the one of open source tools.

DAST tools perform a syntactic analysis of the application. Therefore the **coverage degree of vulnerabilities** is more reduced (see section 4.4.1). IAST tools can perform a more real analysis, analyzing the environment of processes in runtime and increasing the vulnerabilities coverage. Also HYBRID tools leverage the combination of several types of tools to increase the vulnerability coverage. There are important vulnerabilities, as RFI, not covered by the four commercial tools. Vulnerabilities for web services, as XMLI, XPATH and XXE, are not covered by some commercial and open source tools. The requisite of testing web service applications makes necessary improving the vulnerability coverage of the tools. Commercial tools have better vulnerabilities coverage degree (tables 37 and 38) than open source tools.

The **usability** skills of DAST commercial tools for performing analysis are better than the ones of open source tools, with respect to scanning speed, quality of reports, utilities for checking false positives and integration of IAST tools or SAST tools for hybrid analysis. The output formats of tools are improving because they are increasingly using MITRE CWE, OWASP TOP TEN 2013 and SANS TOP 25 standards to formatting the reports.

The vulnerabilities of **WAVSEP benchmark** are adequate to test DAST tools. WAVSEP vulnerabilities are between the most dangerous in OWASP top ten 2013 or SANS top 25.

Finally we have confirmed that the level of performance of the DAST open source tools on detection ratio, false positive ratio, vulnerability coverage, functionality and usability is closer to commercial versions than SAST open source tools with respect to SAST commercial versions.

8.6.1. DAST, IAST and HYBRID Recommendations.

As a consequence of the analysis of the results in this study, we propose some recommendations for improving the use of DAST, IAST and HYBRID analysis tools*:*

- The selection of a tool for security analysis of an application must be based on a good comparative study of the capabilities of tools taking into account the economic possibilities.
- The different designs of the tools with different detection findings makes interesting combining several tools or use an HYBRID tool to increase the results performance and obtain more true positives and eliminating more false positives.
- DAST tools must improve the detection ratio of vulnerabilities. By example DAST tools have problems to detect the LFI vulnerability.

- DAST tools must increase the vulnerability coverage degree for important vulnerabilities of web services, RFI and other important vulnerabilities.
- The IAST and HYBRID tools' performance results are very good in this study but it is important to develop new implementations to obtain more objective results about their performance. Also future studies and assessments are necessary to improve this particular field.
- It is necessary to make efforts for standardization of formats of reports (SANS TOP 25 or MITRE CWE) and improving the vulnerability trace skills incorporating utilities to check false positives to perform the required posterior auditory.
- WAVSEP is an adequate application benchmark but must be improved with new test cases for new vulnerabilities (recently two new vulnerabilities have been included [Wavsep, 2014]).

8.7. Integration of tools in SSDLC.

The SSDLC models analyzed in section 4.2 incorporate SAST and DAST tools into their SSDLC in the implementation and test phases respectively. However, actually IAST or HYBRID tools are not mentioned in these SSDLC processes. We think that there are enough arguments to introduce **IAST and HYBRID tools in a SSDLC process,** as supported by our results**.** Seeker and HP Fortify Hybrid obtain the best results in this assessment.

HYBRID tools can be used in test phase analyzing the application by correlating the results obtained by each different component tool. Also HYBRID tools could integrate several types of analysis, leveraging the synergies between different types of tools (next generation of HYBRID tools) without necessity of correlating results because they present an integrated result (See section 4.6 for PHP Vulnerability Hunter tool by example).

With respect to **IAST tools**, they must be incorporated at test or deployment phases depending on their work mode. By example Seeker tool or Fortify Securityscope (see section 4.6) must be used only in test phase because it performs a white box analysis of the application with the objective of reporting the vulnerabilities it finds. Other tool as Fortify RTA (see section 4.6) can be used as a firewall in deployment phase because it can block an attack attempt exploiting a vulnerability in runtime.

This dissertation has shown (figure 42) where each type of tool can be used in order to incorporate new tendencies in white box, black box and HYBRID analysis tools. Based on this framework, an organization must choose the most appropriate tools according to the characteristics of their applications, their development policy, security policy and their code revision process. Following a SSDLC model an organization should have adequate type tools for each phase of SSDLC. At least it should have a SAST tool and a DAST and/or IAST tool. Actually, new HYBRID tools solutions of various types are appearing while the debate between static and dynamic tools continues.

A first premise to take into account is that SAST is probably the most important type of tool according to their capabilities, and existing comparatives between static and dynamic tools analyzing their effectiveness and width of surface attack application and weakness coverage. SAST tools finds more true positives because DAST tools only can check the parts of web application externally accessible. **The starting point must be a good choice of SAST tool.**

A second premise is that HYBRID tools have the possibility of correlating or combining same or different types of tools exploiting their potential synergies in order to minimize false positives and false negatives ratios. **A good HYBRID tool could entirely cover the security analysis requisites.**

The organization must adopt a specific existing SSDLC model or design a new adapted model to the organization characteristics about type of applications, time of development for a project, availability of personal.

All **SSDLC implementations must be cyclic**, that is, the cycle must be iterated the necessary times required to achieve an application the most secure possible. Several strategies can be adopted for each SSDLC iteration:

1. Audit the application at implementation phase with a SAST tool and fix in the code the true positives found. Following in test phase audit the application with DAST, IAST or some HYBRID tool and fix the true positives found in the code. If a HYBRID with a SAST tool is used, a second audit with static analysis will be performed in the same SSDLC iteration.
2. Audit the application at implementation phase with a SAST tool and fix in the code the true positives found. Following in test phase audit the application with DAST, IAST or some HYBRID tool (with no SAST) and fix the true positives found in the code.

This cycle would end when the application has achieved an acceptable security level according to the criteria of the development and security teams performing the application analysis. Also the available time or/and economics possibilities can determine the end of the SSDLC iterations. It is mandatory to schedule all projects and applications to develop in the organization.

The organizations have a compromise on the choice of who performs the analysis: the programmers, the security team or both, this choice depends on the several factors [Chess, 2007]: team compositions, knowledge level of programmers, analysis time available or number and scope of applications.

8.8. Future work.

It is necessary to develop new implementations of SAST, DAST, IAST and HYBRID tools that improves performance, functionality and usability of the actual implementations.

Our future work will be evaluating the performance and other necessary skills of SAST, DAST, IAST and HYBRID tools to determinate adequate conclusions about:

1. Languages coverage, vulnerabilities coverage, false positives and false negatives ratios and other similar complementary metrics obtained from available and public assessments or performing the assessment by the organization.
2. Usability, required knowledge degree of the tool and vulnerability trace facilities of the vulnerabilities found to aid eliminating false positives.
3. Phase of development life cycle to apply the tool according to its type and purpose.
4. Adequate knowledge (web vulnerabilities, secure languages considerations…) of developing personal to handle the tools and perform the audit and vulnerability correlations of the application analysis.
5. The economic prices of acquiring the tools by a company. The organization must evaluate the availability and most adequate tools to be included in the SSDLC process according to the tool capabilities and economic possibilities.

Actually I am promoting and advising the implementation of a software security analysis project in the Marañosa Institute of technology of Spanish Defense Ministry, with the purpose of analyzing the security of source code software and performing the test phase security activities of all projects belonging to the Defense Ministry.

BIBLIOGRAPHY

[Acunetix, 2013] Acunetix official site. URL last accessed online on August 2013. http://www.acunetix.com/websitesecurity/rightwvs/

[Aerts, 2003] Aerts A.T.M., Goossenaerts J.B.M., Hamer and D.K., Wortmann J.C., Architectures in context: on the evolution of business, application software, and ICT platform architectures. Journal Information and Mangement, Volume 41 Issue 6, July 2004, Pages 781–794, Elsevier Science Publishers, doi 10.1016/j.im.2003.06.002.

[Aiken, 2006] Aiken A., Bugrara S., Dillig I., Dillig T., Hackett B. and Hawkins P. The Saturn program analysis system. 2006, URL last accessed online on May 2013. http://saturn.stanford.edu

[Ajax, 2013] Ajax: A New Approach to Web Applications. URL last accessed online on May 2013, http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

[AnantaSec, 2009] AnantaSec DAST-HYBRID tools comparison. URL last accessed online on January 2014. http://anantasec.blogspot.com.es/

[Antunes, 2009] Antunes N., Vieira M., Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing

[Aoki, 2010] Aoki, Y., Matsuura, S., A Method for Detecting Defects in Source Codes Using Model Checking Techniques.

Computer Software and Applications Conference (COMPSAC), 2010 IEEE 34th Annual. Digital Object Identifier: 10.1109/COMPSAC.2010.61. Page(s): 543 – 544

[Appscan, 2103] IBM security appscan products official site. URL Last accessed online on August 2013. http://www-03.ibm.com/software/products/us/en/appscan/

[Arachni, 2013] Arachni official site. Last accessed online on august 2103. http://www.arachni-scanner.com/

[Armorize, 2103] Armorize products official site. Last accessed online on august 2103. http://www.armorize.com/codesecure/

[Aspect, 2013] Aspect Security official site. URL Last accessed online on august 2103. https://www.aspectsecurity.com/contrast/

[Artho, 2005**]** Cyrille A., Armin B., Preliminary Version Combined Static and Dynamic Analysis. Electronic Notes in Theoretical Computer Science (ENTCS). Volume 131, May, 2005 Pages 3-14.

[Babic, 2011] Domagoj B., Martignoni L., McCamant S., Song D., Proceeding ISSTA '11 Proceedings of the 2011 International Symposium on Software Testing and Analysis. Pages 12-22.

[Balzarotti, 2008] Balzarotti D., Cova M., Felmetsger V., Jovanovic N., Kirda E., Kruegel C., Vigna G., Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. IEEE

Symposium on Research on Security and Privacy, Oakland, CA. May 2008.

[Bau, 2010] Jason B., Bursztein E., Gupta D., Mitchell J., State of the Art: Automated Black-BoxWeb Application Vulnerability Testing. IEEE Symposium on Security and Privacy, 2010.

[Bermejo, 2009] Bermejo J.R., Secure Coding Application: Analysis, design and development without security vulnerabilities. Final degree Project for Computer Engineering. 2009. UNED. Madrid.

[Bermejo, 2011] Bermejo J.R., Study of automatic analysis techniques security vulnerability in web application. 2011. Final degree master. UNED. Madrid.

[Beyer, 2007] Beyer D., Henzinger T. A., Jhala R. and Majumdar R., The software model checker BLAST. Applications to software engineering, International Journal Software Tools for Technology Transfer (2007) 9:505–525

[Black, 2007] Black E., Software Assurance with SAMATE reference dataset, tool standards and studies, Proc. 26th IEEE/AIAA Digital Avionics Systems Conference, Dallas, Texas, Oct. 2007

[Boop, 2013] Boop static tool official site. URL last accessed online on May 2013, http://boop.sourceforge.net

[Bsimm, 2013] The Building Security In Maturity Model official site. URL last accessed online on May 2013, http://bsimm.com/

[Buguroo, 2013] Buguroo products official site. URL last accessed online on May 2013. https://buguroo.com/productos/bugscout/

[Cat, 2013] Microsoft CAT.NET SAST tool official site. URL last accessed online on May 2013. http://www.microsoft.com/en-us/download/details.aspx?id=19968

[Chevaro, 2012] Chebaro O., Kosmatov N., Giorgetti A., and Julliand J., Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis. In *SAC 2012, 27-th ACM Symposium On Applied Computing*, Trento, Italy, pages 1284--1291, March 2012.

[Cenciz, 2013] Cenciz Vulnerability scanner official site. http://www.cenzic.com/index.html URL last accessed online on May 2013.

[Chaki, 2004] Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H., Modular verification of software components in C. IEEE Trans. Softw. Eng. 30(6), pages 388–402. Wiley. 2004.

[Checkmarx, 2013] Checkmarx products official site. URL last accessed online on May 2013. http://www.checkmarx.com/technology/cxsuite/

[Chen, 2008] Chen F., Serbanuta T. F. and Rosu G., JPredictor: a predictive runtime analysis tool for java. ICSE, page 221-230. ACM, 2008.

[Cheng, 2006] Cheng W., Zhao Q., Yu B.and Hiroshige S., TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In Proocedings of the 11th IEEE Symposium on Computers and Communications, 2006.

[Chess, 2007] Secure Programming with Static Analysis, By Brian Chess and Jacob West. Addison-Wesley Software Security Series. ISBN: 0-321-42477-8.

[Cheswick, 2003] Cheswick W. R., Bellovin S. M., Rubin A., Firewalls And Internet Security Repelling The Wily Hacker. 2 Rev Ed. Pearson Education 2003. ISBN: 9780201634662. ISBN-10: 020163466X.

[Csallner, 2004] Csallner C. and Smaragdakis Y., JCrasher: An automatic robustness tester for Java. Software—Practice & Experience, 34(11):1025–1050, Sept. 2004.

[Csallner, 2005] Csallner C. and Smaragdakis Y., Check 'n' Crash: Combining static checking and testing. In Proc. 27th International Conference on Software Engineering (ICSE), pages 422–431. ACM, May 2005.

[Csallner, 2006] Csallner C. and Smaragdakis Y., DSD-Crasher: A hybrid analysis tool for bug finding. In Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pages 245–254. ACM, July 2006.

[Cifuentes, 1997] Cifuentes C. and Fraboulet A., Intraprocedural static slicing of binary executables. Software Maintenance, 1997. Proceedings., International Conference on, pages 188–195, 1997.

[Cifuentes, 2008] Cifuentes C. and Scholz B., Parfait – Designing a Scalable Bug Checker, SAW '08: Proc. of the 2008 workshop on Static analysis, pp. 4-11, June 2008.

[Clarke, 2004] Clarke E., Kroening D. and Lerda F., A Tool for Checking ANSI-C Programs, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), Lecture Notes in Computer Science, Springer Verlag, 2004

[Clarke, 2005] Clarke E., Kroening D., Sharygina N. and Yorav K., SATABS: SAT-Based Predicate Abstraction for ANSI-C, Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Lecture Notes in Computer Science, Springer Verlag, 2005

[Cmmi, 2013] CMMI for application development official site, URL last accessed online May 2013. http://cmmiinstitute.com/resource/security-by-design-with-cmmi-for-development-version-1-3/

[Connolly 2008] Connolly G. M.; Akin M., Goyal A.; Howlett R.. and Perrins M., Building Dynamic Ajax Applications Using WebSphere Feature Pack for Web 2.0. Publisher: IBM Redbooks Date: November 06, 2008. Part Number: SG24-7635-00 Print ISBN-10: 0-7384-3173-7 Print ISBN-13: 978-0-7384-3173-4.

[Cousot 1977] Cousot P. and Cousot R., Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium in Principles of Programming Languages, POPL '77, 1977, pp. 238–252.

[Coverity, 2013] Coverity products official site. Last accessed on august 2013. http://www.coverity.com/products/coverity-prevent.html

[Cowan, 1998] Cowan C., Pu C., Maier D., Walpole J., Bakke P., Beattie S., Grier A., Wagle P., Zhang Q., Hinton H., StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, in: Proceedings of the 7th USENIX Security Conference, San Antonio, Texas, January 1998, pp. 63-78.

[CppCheck, 2013] CppCheck SAST official site. . Last accessed online on august 2013. http://cppcheck.sourceforge.net/

[CVE, 2013] Mitre Common Vulnerabilities and Exposures official site. URL last accessed online on May 2013. http://cve.mitre.org/

[Davis, 2005] Secure Software Development Life Cycle Processes: A Technology Scouting Report. Noopur Davis. 2005. Technical Note CMU/SEI-2005-TN-024.

[Detlefs et Ne. 2005] Detlefs D., Nelson G, and Saxe J. B., Simplify: A Theorem Prover for Program Checking, *Hewlett-Packard*. Journal of the ACM, Vol. 52, No. 3, May 2005, pp. 365–473.

[Devietti] Devietti J., Blundell C., Martin M. K., Zdancewic S., HardBound: Architectural support for spatial safety of the C programming language, in: Proceedings of the International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS‟08), Seattle, USA, March 2008, pp. 1-12.

[De win, 2009] On the Secure Software Development Process: CLASP, SDL and Touchpoints Compared. Bart De Win, Riccardo Scandariato, Koen Buyens, Johan Gr´egoire, Wouter Joosen. Information and

Software TechnologyVolume 51, Issue 7, July 2009, Pages 1152–1171

[Diaz, 2013] Díaz G., Bermejo J. R., Static analysis of source code security: Assessment of tools against SAMATE tests. Information and Software Technology, Volume 55, Issue 8, August 2013, Pages 1462-1476,

[Doupe, 2012] Doupé A., Cavedon L., Kruegel C., Vigna G.. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. Proceedings of the USENIX Security symposium. 2012.

[Ernst, 2003] Ernst M. D., Static and dynamic analysis: synergy and duality. In WODA 2003: ICSE Workshop on Dynamic Analysis, (Portland, OR), May 9, 2003, pp. 24-27

[Esc, 2013] ESC SAST official site. URL last accessed on august 2013. http://kindsoftware.com/products/opensource/ESCJava2/download.html

[Evans, 2002] Evans D., Larochelle D., Improving Security Using Extensible Lightweight Static Analysis, IEEE Software Jan/Feb 2002

[Lapse+, 2103] Lapse+ SAST official site. URL last accessed on December 2103. http://www.evalues.es/?q=node/14

[Emanuelsson, 2008] Emanuelsson P. and Nilsson U., A Comparative Study of Industrial Static Analysis Tools (Extended Version), Technical reports in Computer and Information Science. Report number 2008:3, January 7 2008.

[Evans, 2002] Evans D., Larochelle D., Improving Security Using Extensible Lightweight Static Analysis, IEEE Software Jan/Feb 2002

[FAA-iCMM, 2013] Federal Aviation Administration Integrated Capability Maturity Model. URL online last accessed on May 2013, http://www.faa.gov/about/office_org/headquarters_offices/aio/library/media/v2-mapsupplement_web.pdf

[Findbugs, 2013] SAST tool official site. Last accessed online on August 2103. http://findbugs.sourceforge.net/

[Flame, 2012] Flame virus. Aaccessed online on May 2013. http://www.symantec.com/connect/blogs/flamer-highly-sophisticated-and-discreet-threat-targets-middle-east

[Flash, 2103] Adobe flash official site. URL last accessed on May 2013, http://www.adobe.com/es/products/flash-builder.html, http://www.adobe.com/es/products/flash.html

[Fong, 2007] Fong E. and Okun V., Web Application Scanners: Definitions and Functions, 40th Annual Hawaii International Conference on System Sciences (HICSS'07), p. 280b, 2007.

[Fong, 2008] Fong E., Gaucher R., Okun V., Black P. E. and Dalci E., Building a Test Suite for Web Application Scanners, Hawaii International Conference on System Sciences (HICSS'08), to appear.

[Foster, 1999] Foster J., Fahndrich M. and Aiken A., A theory of type qualifiers, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99), pages 192.203, May 1999

[Fujaba, 2013] Fujaba Tool suite official site. URL Last accessed online on September 2013. http://www.fujaba.de/

[FxCop, 2013] Fxcop SAST Microsoft official site. URL Last accessed online on september 2013. http://www.microsoft.com/en-us/download/details.aspx?id=6544

[Gartner, 2010] Gartner static, 2010. Magic cuadrant for static application security testing, URL last accessed online on May 2013. https://www.fortify.com/ssa-basics/Gartner2010MQ_SAST.html ,

[Gartner, 2013] Gartner static, 2010. Magic cuadrant for static application security testing, URL last accessed online on May 2013. http://www.gartner.com

[Gimpel, 2103] Gimpel products official site. Last accessed online on august 2013. http://www.gimpel.com/html/index.htm

[Grabber, 2013] Grabber product official site. Last accessed online on october 2103. http://rgaucher.info/beta/grabber/

[Grammatech, 2013] SAST tool Codesonar official site. Last accessed online on august 2103. http://www.grammatech.com/products/

[Gray,1993] Gray J., The Benchmark Handbook. Morgan, Kaufmann Publishers, San Francisco, CA, USA, 1993.

[Grepper, 2013] VisualCodeGrepper SAST official site. Last accessed online on august 2013. http://sourceforge.net/projects/visualcodegrepp/

[Haldar, 2005] Haldar V., Chandra D. and Michael Franz M., Dynamic Taint Propagation for Java**.** Proceeding ACSAC '05 Proceedings of the 21st Annual Computer Security Applications. Pages 303 – 311.

[Halfond, 2006] W. G. J. Halfond and Orso A., Preventing SQL injection attacks using AMNESIA. Proceeding ICSE '06 Proceedings of the 28th international conference on Software engineering. Pages 795-798

[Halfond, 2011] Halfond W. G. J., Choudhary S. R. and Orso A., Improving penetration testing through static and dynamic analysis. Softw. Test. Verif. Reliab. (2011) Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.450

[Hanov 2005] Hanov S., Static Analysis of Binary Executables. University of Waterloo 200 University Avenue West Waterloo, Ontario, Canada N2L 3G1. Personnal web page. Last accessed online on August 2013. http://stevehanov.ca/cs842_project.pdf

[Hofer, 2010] Hofer T., Evaluating Static Source Code Analysis Tools. Ecole Polytechnique Federale de Lausanne. 2010. URL Last accessed online on August 2013, http://infoscience.epfl.ch/record/153107?ln=en

[Holzmann, 2002] Holzmann G., UNO: Static Source Code Checking for UserDefined Properties. In 6th World Conf. on Integrated Design and Process Technology, IDPT ’02

[Homeland, 2014] Build Security In [Electronic resource]. Source Code Analysis Tools Example Programs. Last accessed online on August 2013:

https://buildsecurityin.us-cert.gov/bsi/articles/tools/code/498-BSI.html

[Howard, 2003] Howard M, LeBlanc D.. Writing Secure Code. 2nd ed. PUBLISHED BY Microsoft Press. ISBN 0-7356-1722-8.

[HP-Fortify, 2013] HP Fortify products official site. URL last accessed online on August 2103. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812#tab=TAB2

[HP-report, 2012] HP ciber risk report 2012, URL last accessed online on May 2013 http://www.hpenterprisesecurity.com/register/guarding-against-a-data-breach-hp.com

[HTML5, 2013] HTML5 official site, URL last accessed online on May 2013, http://www.w3schools.com/html/html5_intro.asp

[Huang, 2004] Huang Y. W., Yu F., Hang C, Tsai C. H., Lee D. T., Kuo S. Y., Securing Web Application Code by Static Analysis and Runtime Protection. In Proceedings of the 13th international conference on World Wide Web (2004), pp. 40-52.

[Hunter, 2013] PHP VULNERABILITY HUNTER official product 2103. URL last accessed online on November 2013. https://phpvulnhunter.codeplex.com/

[IBM, 2012] IBM x-force 2012 mid-year trend and risk report. URL last accessed online on May 2013. http://www-03.ibm.com/security/xforce/downloads.html

[IBM-Appscan, 2013] IBM Appscan products official site, URL last accessed online on May 2013, http://www-03.ibm.com/software/products/us/en/appscan/

[IETF, 1999] IETF, Site Security Handbook. RFC 2196., URL last accessed online on May 2013. http://www.ietf.org/rfc/rfc2196.txt

[Insure, 2013] Insure product official site. URL last accesed online on October 2103. http://www.parasoft.com/jsp/products/insure.jsp/

[Ironwasp, 2013] IRONWASP product official site. URL last accesed online on October. http://ironwasp.org/

[Klocwork, 2013] Klocwork products official site. URL last accessed online on May 2013. http://www.klocwork.com/products/insight/?source=feature

[JavaFX, 2013] JavaFX official site. URL last accessed online on May 2013. http://docs.oracle.com/javafx/

[Javascript, 2013] Javascript language. URL last accessed online May 2013 http://es.wikipedia.org/wiki/JavaScript

[Johnson, 1977] Johnson S., Lint, a C program Checker. Computer Science Technical Report 65, Murray Hill, NJ: Bell Laboratories, December 1977.

[JQuery, 2013] JQuery official site. URL last accessed online on May 2013 http://jquery.com/

[J2EE, 2013] J2EE official site. URL last accessed online on May 2013 http://www.jcp.org/en/jsr/detail?id=151

[Kara, 2012] Mehmet K. Review on Common Criteria as a Secure Software Development Model. International Journal of Computer Science & Information Technology (IJCSIT) Vol 4, No 2, April 2012.

[Keugh, 2005] Luk C. K., Cohn R., Muth R., Patil H., Klauser A., Lowney G., Wallace S., Janapa V., and Hazelwood R. K., Building customized program analysis tools with dynamic instrumentation. In Programming Language Design and Implementation, pages 190–200. ACM Press, 2005.

[Kiezun, 2009] Kieyzun, A., Guo, P.J., Jayaraman, K., Ernst, M.D., Automatic Creation of SQL Injection and Cross-Site Scripting AttacksICSE '09 Proceedings of the 31st International Conference on Software Engineering.

[Klocwork, 2013] Klocwork Insight SAST official site. URL last accessed online on August 2013. http://www.klocwork.com/products/insight/

[Kratkiewicz, 2005] Kratkiewicz K., Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code, Master's Thesis, Harvard University, 2005. URL last accessed online on August 2013. http://www.ll.mit.edu/mission/communications/ist/publications/KratkiewiczThesis.pdf

[Krishnan, 2008] Krishnan R., Nadworny M. and Bharill N., Static Analysis Tools for Security Checking in Code at Motorola, SIGAda Ada Letters, vol. 28 issue 1, April 2008.

[Lam, 2008] Lam M., livshits B., Waley J., Securing Web Applications with Static and Dynamic Information Flow Tracking. PEPM '08

Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation.

[Lapse, 2013] Lapse+ SAST official site. Last accessed online on August 2013. http://evalues.es/?q=node/14

[Li, 2012] Li D., Liu Z., Zhao Y., HeapDefender: A mechanism of defending embedded systems against heap overflow via hardware, in: Proceedings of the International Conference on Ubiquitous Intelligence and Computing and International Conference on Autonomic and Trusted Computing (UIC-ATC), Fukuoka, Japan, September 2012, pp. 851-856.

[Livshits, 2005] Livshits B. and Lam M. S., Finding Security Vulnerabilities in Java Applications with Static Analysis. Computer Science Department. Stanford University. Technical ReportSeptember 25, 2005.

[Livshits 2006] Livshits B. Doctoral Dissertation. Improving software security with precise static and runtime analysis. Stanford University Stanford, CA, USA ©2006. ISBN: 978-0-542-98404-4.

[Lobo, 2013] Lobo V., Rodrigues C., Soares F. A., Leonardo C., Rizzo A. M. Static Analysis Techniques and Tools: A Systematic Mapping Study. *ICSEA 2013:* The Eighth International Conference on Software Engineering Advances.

[Long, 2005] Long F.. Software Vulnerabilities in Java. CERT Technical note CMU/SEI-2005-TN-044. URL last accessed online on May 2013

http://repository.cmu.edu/cgi/viewcontent.cgi?article=1427&context=sei

[Martin, 2008] Martin R. A., and Barnum S., Creating the Secure Software Testing Target List. Proc. of the 4th annual workshop on Cyber security and information intelligence research: developing strategies to meet the cyber security and information intelligence challenges ahead, CSIIRW, Vol. 288, article no. 33, 2008.

[Mavituna, 2013] Mavituna security products official site. URL last accessed online on December 2103. https://www.mavitunasecurity.com/

[McGraw 2006] McGraw G., Software Security: Building Security In. Publisher: Addison Wesley Professional. Print ISBN-10: 0-321-35670-5.

[MathWorks, 2013] SAST tool Polyspace official site. URL last accessed online on August 2013. http://www.mathworks.es/products/polyspace/

[Microsoft, 2013] Applications architecture and design by Microsoft. URL last accessed online onMay 2013 http://msdn.microsoft.com/en-us/library/ee658086.aspx

[Microsoft-SDL, 2013] Microsoft SDL official site. URL last accessed online on August 2013. http://www.microsoft.com/security/sdl/default.aspx

[Mitre, 2013] Mitre CWE official site. URL last accessed online on May 2013. http://cwe.mitre.org/

[Mobile, 2013] Mobile application development. URL last accessed online May 2013, http://en.wikipedia.org/wiki/Mobile_application_development

[Monga, 2009] Monga M., Paleari R. and Passerini E., A hybrid analysis framework for detecting web application vulnerabilities Universit_a degli Studi di Milano. Milano, Italy. SESS'09: Proceedings of the 5th International Workshop on Software Engineering for Secure Systems.

[Moskewicz, 2001] Moskewicz M. W., Madigan C. F., Zhao Y., Zhang L., Malik S. Chaff: Engineering an Efficient SAT Solver., DAC '01 Proceedings of the 38th annual Design Automation Conference. Pages 530-535. ACM New York, NY, USA ©2001.

[Nagarajan, 2009] Nagarajan V., Gupta R., Architectural support for shadow memory in multiprocessors, in: Proceedings of the ACM Conference on Virtual Execution Environments (VEE‟09), Huston, USA, March 2009, pp. 1-10.

[Nazario, 2002] Nazario J. Source Code Scanners for Better Code. linuxJournal, see last accessed on August 2013. http://www.linuxjournal.com/article/5673?page=0,0

[.NET, 2013] J2EE official site. URL last accessed online on May 2013. http://www.microsoft.com/net

[Newsome, 2005] Newsome J. and Song D.. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In 12th Annual Network and Distributed System Security Symposium, 2005.

[Nicholson, 2012] SCADA security in the light of Cyber-Warfare, A. Nicholson, S.Webber, S. Dyer, T. Patel, H. Janicke. Computer and Security, volume31, issue 4, June 2012, pages 418-436

[Nist, 2013] National Institute of Standards and Technologies from U.S. Department of commerce official site. URL last accessed online on http://www.nist.gov/index.html

[NIST268, 2007] NIST Special Publication 500-268, Source Code Security Analysis Tool Functional Specification Version 1.0, 2007. URL accessed online on September 2013. http://samate.nist.gov/docs/source_code_security_analysis_tool_spec_05_07_07.pdf

[NIST269, 2008], NIST Special Publication 500-269, webapp scanner specification 2008. URL accessed online on September 2013. http://samate.nist.gov/docs/webapp_scanner_spec_sp500-269.pdf

[NIST270, 2009], NIST SP 500-270, NIST Special Publication 500-270, Source Code Security Analysis Tool Test Plan, 2009. URL accessed online on September 2013. http://samate.nist.gov/docs/source_code_security_analysis_test_plan_01_09_08.pdf

[NIST297, 2012] NIST SP 500-283, NIST Special Publication 500-297, 2012, "Static Analysis Tool Exposition (SATE) 2012", URL last accessed online on September 2013. http://samate.nist.gov/SATE4.html

[NIST7755, 2010] NIST Interagency Report 7755, 2010, “Toward a Preliminary Framework for Assessing the Trustworthiness of Software”, URL last accessed online on September 2013 http://samate.nist.gov/docs/toward-1119.pdf

[Nsa, 2013] U.S. National security agency official site. URL last accessed online on July 2013. http://www.nsa.gov/

[Oasis, 2103] Organization for the Advancement of Structured Information Standards official site. URL last accessed online on July 2013. https://www.oasis-open.org/org

[O’Donoghue 2002] O’Donoghue, A. Leddy, J. Power, and J. Waldron., Bigram analysis of Java bytecode sequences PPPJ '02/IRE '02 Proceedings of the inaugural conference on the Principles and Practice of programming, pages 187–192.

[Oissg, 2013] Open Information Systems Security Groups official site. URL last accessed online on July 2013. http://www.oissg.org/

[OPenLaszlo, 2013] OPenLaszlo official site, URL last accessed online on May 2013 http://www.openlaszlo.org/

[OpenSAMM, 2013] Open Software Assurance Maturity Model official site, URL last accessed online on May 2013. http://www.opensamm.org/

[Osvdb, 2013] Open Sourced Vulnerability Database official site. URL last accessed online on July 2013. http://www.osvdb.org/

[Owasp, 2013] OWASP TOP TEN 2013 security vulnerabilities classification, URL last accessed online on May 2013 https://www.owasp.org/index.php/Top_10_2013

[Owasp-CLASP, 2013] OWASP CLASP project official site. URL last accessed online on May 2013. https://www.owasp.org/index.php/Category:OWASP_CLASP_Project/es

[Palmieri, 2012] Palmieri M., Inderjeet S., Antonio Cicchetti A., Comparison of Cross-Platform Mobile Development Tools. 2012 16th International Conference on Intelligence in Next Generation Networks.

[Parasoft, 2013] Parasoft products official site. URL Last accessed online on august 2013. http://www.parasoft.com/jsp/home.jsp

[Paros, 2013] Paros official site. Last accessed online on august 2013. http://sourceforge.net/projects/paros/

[Pathfinder, 2013] Java Pathfinder SAST official site. Last accessed on August 2013. http://kindsoftware.com/products/opensource/ESCJava2/download.html

[Pmd, 2013] PMD SAST tool official site. Last accessed online on August 2013. http://pmd.sourceforge.net/

[Portswigger, 2013] Portswigger official site. Burp suite tool. URL Last accessed on August 2013. http://portswigger.net/burp/successstories.html

[Pranith, 2009] Kumar D. P., Nema A. and Kumar R., Hybrid Analysis of Executables to Detect Security Vulnerabilities. ISEC '09 Proceedings of the 2nd India software engineering conference. Pages 141-142. 2009.

[Purify, 2013] IBM purify product official site. URL Last accessed online on october 2013. http://www-03.ibm.com/software/products/us/en/rational-purify-family

[Quotium, 2013] Quotium official site. URL last accessed online on August 2013. http://www.quotium.com/prod/security.php

[Petukhov, 2008] Petukhov A., Kozlov D., Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing, OWASP-AppSecEU08, Belgium, 2008.

[Pomorova, 2103] Pomorova, O.V. and Ivanchyshyn, D.O., Assessment of the source code static analysis effectiveness for security requirements implementation into software developing process. Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2013 IEEE 7th International Conference on (Volume:02). 12-14 Sept. 2013. Berlin. Pags. 640–645. ISBN 978-1-4799-1426-5. 10.1109/IDAACS.2013.6663003

[Prefast, 2013] Prefast Analysis Tool official site. URL Last accessed online on August 2013. http://msdn.microsoft.com/en-us/library/ms933794.aspx

[RedLizard, 2013] RedLizard products official site. URL last accessed online on August 2013. http://redlizards.com/

[Rijsbergen, 1979] Rijsbergen V., Cornelis J., "Keith" (1979); *Information Retrieval*, London, GB; Boston, MA: Butterworth, 2nd Edition, ISBN 0-408-70929-4

[Rips, 2013] Rips SAST official site. URL Last accessed online on August 2013. http://sourceforge.net/projects/rips-scanner/

[Samate, 2013] Nist-Samate official site. URL Last accessed online on august 2013. http://samate.nist.gov/Main_Page.html

[Sans, 2013] SANS TOP 25 security vulnerabilities classification, URL last accessed online on May 2013. http://www.sans.org/top25-software-errors/

[SAST-samate, 2103] NIST-Samate project official site, list of SAST tools (static analysis). URL Last accessed online on August 2013. http://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html

[SAST-wasc, 2013] Wasc official site, list of SAST tools. URL Last accessed online on August 2013. http://projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList

[SAST-wiki, 2013] Wikipedia official site, list of SAST tools. URL Last accessed online on August 2013. http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

[SAST-owasp, 2013] Owasp official site, list of SAST. URL Last accessed online on August 2013. https://www.owasp.org/index.php/Static_Code_Analysis

[Saxena, 2011] Saxena P., Molnar D. and Livshits B., SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. CCS '11 Proceedings of the 18th ACM conference on Computer and communications security. 2011.

[SEC, 2012] Web Application Scanners (DAST) evaluation project official site. URL last accessed online on November 2013. http://www.sectoolmarket.com/

[Securebench, 2103] Securebench micro project official site. URL last accessed online on November 2013 http://suif.stanford.edu/~livshits/work/securibench-micro/

[Shrestha, 2013] Shrestha J., Static Program Analysis. Degree of Masters of Information System Jayesh Shrestha. Uppsala University. September, 2013. URL last accessed online on November 2013. http://www.diva-portal.org/smash/get/diva2:651821/FULLTEXT01.pdf

[Sipser, 2005] Sipser M., Introduction to the Theory of Computation, Second Edition. New York, NY: Course Technology, 2005.

[SSE-CMM. 2013] Systems Security Engineering Capability Maturity Model Official site, URL last accessed online on May 2013,

http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=44716

[Silverlight, 2013] Microsoft Silverlight official site. URL last accessed online on May the 22. http://www.microsoft.com/silverlight/

[Smutny, 2012] Smutny P., Mobile development tools and cross-platform solutions. 2012 13th International Carpathian Control Conference (ICCC).

[Schneier, 2010] Schneier, B., Stuxnet, URL last accessed online on December 2103, http://www.schneier.com/blog/archives/2010/10/stuxnet.html

[SP-800-82, 2011] Guide for securing SCADA and ICS special publications of NIST. URL accessed last online on May. http://csrc.nist.gov/publications/nistpubs/800-82/SP800-82-final.pdf

[Sridharan , 2011] Sridharan M., Artzi S., Pistoia M., Guarnieri S., Tripp O. and Berg R., F4F: Taint Analysis of Framework-based Web Applications. *OOPSLA'11:* ACM Conference on Systems, Programming, Languages and Applications.

[Sttutard, 2008] Stuttard D. and Pinto M., The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws Published. Copyright © 2008 Published by Wiley Publishing, Inc., Indianapolis, Indiana. ISBN: 978-0-470-17077-9

[Suto, 2010] Suto, L.: Analyzing the Accuracy and Time Costs of Web Application Security Scanners. 2010. URL accessed September

2013.
http://www.google.es/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CDIQFjAA&url=http%3A%2F%2Fha.ckers.org%2Ffiles%2FAccuracy_and_Time_Costs_of_Web_App_Scanners.pdf&ei=oLoRU5DFPMuO7QbKi4HIBQ&usg=AFQjCNFtZlz1PS7DIAEUWiG2E3sE0K-INQ&bvm=bv.62286460,d.Yms

[Tripp, 2009] Tripp O., Pistoia M., Fink S., Sridharan M., and Weisman O., TAJ: Effective Taint Analysis of Web Applications. PLDI'09: ACM Conference on Programming Language Design and Implementation.

[Tripp, 2011] Omer Tripp O. and Weisman O., Hybrid Analysis for JavaScript Security Assessment. ESEC/FSE'11: ACM Conference on the Foundations of Software Engineering.

[Tripp, 2013] Tripp O., Pistoia M., Cousot P., Cousot R. and Salvatore Guarnieri S., Andromeda: Accurate and Scalable Security Analysis of Web Applications. FASE'13: ETAPS Conference on Fundamental Approaches to Software Engineering.

[Trustwave, 2013] Trustwave Global Security Report 2013. URL last accessed online on July 2013. http://www2.trustwave.com/rs/trustwave/images/2013-Global-Security-Report.pdf

[Turing 1936] Turing A., On computable numbers, with an application to the Entscheidungsproblem, Proceedings of the London Mathematical Society, Series 2, 42 (1936), pp 230-265. Turing defines Turing

machines, formulates the halting problem, and shows that it (as well as the Entscheidungsproblem) is unsolvable.

[Yahoo, 2013] Yahoo hijacking of identities in Japón. URL last accessed online on May the 22. http://www.csospain.es/Yahoo-Japon-confirma-el-robo-de-22-millones-de-ID-de-usuario/seccion-actualidad/noticia-132995

[Valgrind, 2013] Valgrind product official site 2013. URL last accessed online on October 2103. http://valgrind.org/

[Veracode, 2010] Hybrid SAST and DASD correlation considerations. URL last accessed online on August 2013 http://www.veracode.com/blog/2010/12/whitepaper-a-dose-of-reality-on-automated-static-dynamic-hybrid-analysis/

[Veracode, 2011] State of Security Software report volume 3. Veracode official site. URL last accessed online on May. http://info.veracode.com/rs/veracode/images/soss-v3.pdf

[Veracode, 2012] State of Security Software report volume 5. Veracode official site. URL accessed online on May. https://info.veracode.com/state-of-software-security-report-volume5.html

[Venkataramani, 2008] Venkataramani G., Doudalis I., Solihin Y. and Prvulovic M., Flexitaint: A programmable accelerator for dynamic taint propagation, in: Proceedings of the the 14th International Symposium on High-Performance Computer Architecture (HPCA‟08), Salt Lake City, UT, Februrary 2008, pp. 173-184.

[Veracode, 2013] VERACODE static analysis official site. URL Last accessed online on august 2013. http://www.veracode.com/security/static-code-analysis

[Viega, 2000] Viega J., Bloch J., Khono Y., Mcgraw G., "ITS4: a static vulnerability scanner for C and C++ code". Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference.

[W3c, 2013] Web services guide. URL last accessed online on May 2013. http://www.w3c.es/Divulgacion/GuiasBreves/ServiciosWeb

[Wagner, 2000] Wagner D., Foster J., Brewer E., Aiken A., A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In Network and Distributed System Security Symposium (February 2000), pp. 3-17.

[Wapiti, 2013] Wapiti product official site. URL last accessed online on May 2103. http://wapiti.sourceforge.net/

[Wagner, 2005] Wagner S., Jrjens J., Koller C., and Trischberger P., Comparing bug finding tools with reviews and tests, Proceedings 17th International Conference on Testing of Communicating Systems, volume 3502 of Lecture Notes in Computer Science, June 2005.

[Wasc, 2013] Web Application Security Consortium official site, URL last accessed online on May 2103. http://www.webappsec.org/

[Wasc-Statistics, 2008] Web Application Security Statistics project 2008. URL accessed online on May 2103. http://projects.webappsec.org/w/page/13246989/Web%20Application%20Security%20Statistics

[Wasc-WAF 2013] WASC Web Application Firewalls Evaluation criteria. URL last accessed online on January 2014. http://projects.webappsec.org/w/page/13246985/Web%20Application%20Firewall%20Evaluation%20Criteria

[Wassermann, 2008]. Gary Wassermann G., Yu D. and Chander A., Dynamic Test Input Generation for Web Applications. ISSTA '08 Proceedings of the 2008 international symposium on Software testing and analysis University of California.

[Wavsep, 2014] Wavsep web application benchmark. URL last accessed online on January 2014. http://code.google.com/p/wavsep/

[Whid, 2103] Web hacking incident database official site. URL last accessed online on May 2013. https://www.google.com/fusiontables/DataSource?snapid=S283929Jw2s

[WhiteHat, 2013] Top Ten Web Hacking Techniques list, WhiteHat official site. URL last accessed online on May 2013. https://blog.whitehatsec.com/top-ten-web-hacking-techniques-of-2012/#.Udvr_eibv4g

[WhiteHat2, 2013] WhiteHat Sentinel product official site. URL last last accessed online on October 2013. https://www.whitehatsec.com/sentinel_services/sentinel_services.html

[Wilander, 2003] Wilander J. and Kamker M., A comparison of publicly available tools for dynamic buffer overflow prevention, in: Proceedings of the International Symposium on Network & Distributed System Security, February 2003.

[W3AF, 2013] W3AF official site. URL last accessed online on May 2013. http://w3af.org/

[Yasca, 2013] Yasca SAST official site. URL last accessed online on May 2013. http://www.scovetta.com/yasca.html

[Yichen, 2005] Yichen X., Chou A., Engler D., ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT International Symposium on foundations on software engineering. 2003. Pags. 327-336.

[Zdi, 2103] The Zero Day Initiative (ZDI), official site, URL last accessed online on July. http://www.zerodayinitiative.com/

[Zhang, 2011] Zhang R., Huang S., Qi Z. and Guan H., Static program analysis assisted dynamic taint tracking for software. Vulnerability discovery. Computers and Mathematics with Applications 63 (2012) 469–480. 2011.

[Zitser, 2004] Zitser M., Lippmann R. and Leek T., Testing Static analysis Tools Using Exploitable Buffer Overflows from Open Source Code., ACM SIGSOFT Software Engineering Notes, Vol. 29, Issue 6, November 2004.

APPENDIX A – CD CONTENTS.

1. Static analysis of source code security: assessment of tools against SAMATE tests. Gabriel Díaz, Juan Ramón Bermejo. Information and Software Technology Volume 55, August (2013) 1462–1476. http://dx.doi.org/10.1016/j.infsof.2013.02.005.

2. NIST-SAMATE TEST SUITE 45

3. NIST-SAMATE TEST SUITE 46

4. NIST-SAMATE TEST SUITE JULIET 2010

5. WAVSEP APPLICATION BENCHMARK

APPENDIX B – SAMATE TEST SUITES 45 – 46 RESULTS.

This appendix shows the test suites 45 and 46 execution results. Table 42 shows the percentage of detections for each vulnerability category and tool in test suite 45. In last row detection percentage mean for each tool is calculated.

Table 42 shows test suite 45 execution results:

Table 42
Summary of results of execution against SAMATE Test suite 45 [Díaz, 2013]

ND: A tool is not designed to detect a vulnerability.

VULNERABILITY DETECTION STATISTICS	CWE	Nº TC	SCA	GOA	PCL	SAT	CBMC	CDS	CX	PRE	K8I
basic XSS	**80**	***5***	0	ND	ND	ND	ND	ND	0	ND	ND
command injection	**78**	***1***	100	ND	ND	ND	ND	ND	100	100	100
double free	**415**	***5***	100	80	20	100	100	100	80	100	100
format string	**134**	***5***	100	0	60	ND	ND	60	80	80	40
hard-coded-password	**259**	***5***	0	ND	ND	ND	ND	ND	20	ND	ND
heap inspection	**244**	***1***	100	0	0	0	0	0	100	0	0
heap overflow	**122**	***5***	100	60	80	80	80	40	100	100	60
improper null termination	**170**	***5***	100	40	100	0	0	20	100	80	80
leftover debug	**489**	***1***	0	100	0	0	0	ND	0	100	0
memory leak	**401**	***2***	50	0	50	0	0	100	100	100	100
null dereference	**476**	***4***	100	75	50	100	100	100	ND	75	75
often missused string management	**251**	***5***	100	100	40	40	40	60	100	100	100
os command injection	**78**	***4***	100	ND	ND	ND	ND	ND	75	75	100
resource injection	**99**	***4***	100	ND	ND	ND	ND	ND	100	100	0
SQL injection	**89**	***3***	100	ND	ND	ND	ND	ND	0	ND	ND
stack overflow	**121**	***9***	100	66.6	77.7	77.7	77.7	55.5	66.6	77.7	66.6
TOCTOU	**367**	***3***	33	ND	ND	ND	ND	33	ND	0	0
unchecked error condition	**391**	***1***	0	100	0	100	100	100	0	0	0
uninitailized variable	**457**	***3***	100	100	66.6	33.3	66.6	66.6	66.6	100	100
unintentional pointer scaling	**468**	***1***	0	0	0	0	0	0	0	0	0
unrestricted critical resource lock	**412**	***1***	0	0	0	0	0	0	0	0	0
use after free	**416**	***5***	80	20	20	80	80	100	60	60	80
DETECTION PERCENTAGE MEAN			**66.5**	**49.4**	**37.6**	**43.6**	**46**	**55.6**	**57.4**	**65.6**	**52.7**

Table 43 shows the percentage of false positives for each vulnerability category and tool in test suite 46. In last row false positive percentage mean for each tool is calculated.

Table 43

Summary of results of execution against SAMATE Test suite 46 [Díaz, 2013]

ND: A tool is not designed to detect a vulnerability.

FALSE POSITIVES STATISTICS	CWE	Nº TC	SCA	GOA	PCL	SAT	CBMC	CDS	CX	PRE	K8
basic XSS	**80**	***5***	0	ND	ND	ND	ND	ND	0	ND	ND
double free	**415**	***4***	25	25	0	50	50	0	0	50	0
format String	**134**	***5***	40	0	60	ND	ND	0	0	0	0
hard-coded-password	**259**	***4***	0	ND	ND	ND	ND	ND	0	ND	ND
heap overflow	**122**	***6***	83.3	66.6	66.6	83.3	83.3	33.3	50	16.6	16.6
improper null termination	**170**	***5***	20	0	100	0	20	0	0	0	20
leftover debug	**489**	***1***	0	100	0	0	0	ND	0	0	0
memory leak	**401**	***5***	0	0	0	40	40	0	20	0	0
null dereference	**476**	***4***	50	25	25	100	100	50	ND	25	25
often missused string management	**251**	***5***	0	0	20	40	100	0	0	0	0
os command injection	**78**	***4***	100	ND	ND	ND	ND	ND	100	0	100
resource injection	**99**	***4***	100	ND	ND	ND	ND	ND	100	0	0
SQL injection	**89**	***4***	100	ND	ND	ND	ND	ND	0	ND	ND
stack overflow	**121**	***8***	25	0	37.8	62.5	75	12.5	25	0	0
TOCTOU	**367**	***2***	100	ND	ND	ND	ND	50	ND	0	0
unchecked error condition	**391**	***1***	0	100	0	0	0	100	100	0	0
uninitailized variable	**457**	***1***	0	0	0	0	0	0	0	0	0
unintentional pointer scaling	**468**	***1***	0	0	100	0	0	0	0	0	0
unrestricted critical resource lock	**412**	***1***	0	0	0	0	0	100	0	0	0
use after free	**416**	***4***	0	0	0	100	100	0	0	0	0
FP PERCENTAGE MEAN			**32.1**	**22.6**	**29.2**	**36.6**	**43.7**	**24.7**	**21.9**	**5.3**	**9.5**

APPENDIX C – SAMATE JULIET 2010 TEST SUITES SELECTION EXECUTION RESULTS.

This appendix shows the results of test cases execution for the SAST assessment of section 5.5.

Legend:

True positives**:** ✓ detected; ● not detected

False positives: ☺ not alarm 🕷 alarm

Table 44. Test cases CWE 23 [Bermejo, 2011]

CWE	DESCRIPTIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
23	**Relative_Path_Traversal**	01	connect_tcp	No	✓ ☺ -	✓ ☺ -	✓ 🕷 -	✓ ☺ -	● ☺ -
11 TEST CASES T. POSITIVES **18 TEST CASES F. POSITIVES**		02	connect_tcp		✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ ☺ ☺	● ☺ ☺
		08	connect_tcp		✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	● ☺ ☺
		04	console_readline		✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	● ☺ ☺	● ☺ ☺
		08	environment		✓ 🕷 🕷	● ☺ ☺	✓ 🕷 🕷	● ☺ ☺	● ☺ ☺
		11	fromDB		✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	● ☺ ☺
		16	fromFile		✓ 🕷 -	✓ 🕷 -	✓ 🕷 -	● ☺ ☺	✓ 🕷 -
		41	getCookiesServlet		✓ ☺ -	✓ ☺ -	✓ 🕷 -	✓ ☺ -	● ☺ -
		07	getParameterServlet		✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	● ☺ ☺
		45	getQueryStringServlet		✓ ☺ -	● ☺ -	✓ 🕷 -	● ☺ -	● ☺ -
		09	listen_tcp		✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ ☺ ☺	● ☺ ☺
TOTAL TRUE POSITIVES				-	**11**	**9**	**11**	**7**	**1**
TOTAL FALSE POSITIVES				-	**15**	**13**	**18**	**6**	**1**

Table 45. Test cases CWE 36 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
36	**Absolute_Path_Traversal**	03	console_readline	no	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	● ☺ ☺	● ☺ ☺
9 TEST CASES T. POSITIVES **14 TEST CASES F. POSITIVES**		04	console_readline		✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	● ☺ ☺	● ☺ ☺
		05	environment		✓ 🕯 🕯	● ☺ ☺	✓ 🕯 -	● ☺ ☺	● ☺ ☺
		14	fromDB		✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	● ☺ ☺
		31	fromFile		✓ ☺ -	✓ ☺ -	✓ 🕯 -	● ☺ -	✓ ☺ -
		13	getCookiesServlet		✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	✓ ☺ ☺	● ☺ ☺
		42	getParameterServlet		✓ ☺ -	✓ ☺ -	✓ 🕯 -	✓ ☺ -	● ☺ -
		19	getQueryStringServlet		✓ 🕯 -	● ☺ -	✓ 🕯 -	● ☺ -	● ☺ -
		12	listen_tcp		✓ ☺ -	✓ ☺ -	✓ 🕯 -	✓ ☺ -	● ☺ -
TOTAL TRUE POSITIVES					**9**	**7**	**9**	**4**	**1**
TOTAL FALSE POSITIVES					**11**	**8**	**14**	**2**	**0**

Table 46. Test cases CWE 78 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
78	**Command_Injection**	05	connect_tcp	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	NO
10 TEST CASES T. POSITIVES **16 TEST CASES F. POSITIVES**		17	console_readline	● ☺ -	✓ 🕯 -	✓ 🕯 -	✓ 🕯 -	● ☺ -	
		05	Environment	● ☺ ☺	✓ 🕯 🕯	● ☺ ☺	✓ 🕯 🕯	● ☺ ☺	
		06	Environment	● ☺ ☺	✓ 🕯 🕯	● ☺ ☺	✓ 🕯 🕯	● ☺ ☺	
		11	Environment	● ☺ ☺	✓ 🕯 🕯	● ☺ ☺	✓ 🕯 🕯	● ☺ ☺	
		06	fromDB	● ☺ ☺	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 -	✓ ☺ ☺	
		16	fromFile	● ☺ -	✓ 🕯 -	✓ 🕯 -	✓ 🕯 -	● ☺ -	
		10	getCookiesServlet	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	
		19	getQueryStringServlet	✓ 🕯 -	✓ 🕯 -	● ☺ -	✓ 🕯 -	● ☺ -	
		45	listen_tcp	✓ ☺ -	✓ ☺ -	✓ ☺ -	✓ 🕯 -	✓ ☺ -	
TOTAL TRUE POSITIVES				**4**	**10**	**6**	**10**	**4**	
TOTAL FALSE POSITIVES				**5**	**15**	**8**	**16**	**4**	

Table 47. Test cases CWE 80 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
80	**XSS**	08	servlet_connect_tcp	● ☺ ☺	● ☺ ☺	● ☺ ☺	✓ 🕱 🕱	NO ANAL.	✓ 🕱 🕱
11 TEST CASES T. POSITIVES **15 TEST CASES F. POSITIVES**		41	servlet_console_readline	● ☺ -	● ☺ -	✓ 🕱 🕱	✓ 🕱 🕱	NO ANAL.	● ☺ ☺
		31	servlet_environment	● ☺ -	● ☺ -	● ☺ -	✓ 🕱 -	NO ANAL.	● ☺ -
		07	servlet_fromDB	● ☺ ☺	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	NO ANAL.	● ☺ ☺
		08	servlet_fromDB	● ☺ ☺	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	NO ANAL.	● ☺ ☺
		31	servlet_fromDB	● ☺ -	✓ ☺ ☺	✓ ☺ ☺	✓ 🕱 🕱	NO ANAL.	● ☺ ☺
		17	servlet_fromFile	● ☺ -	● ☺ -	● ☺ -	✓ 🕱 -	NO ANAL.	● ☺ -
		01	servlet_getCookiesServlet	● ☺ -	✓ ☺ -	✓ ☺ -	✓ 🕱 -	NO ANAL.	● ☺ -
		08	servlet_getParameterServlet	● ☺ ☺	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	NO ANAL.	● ☺ ☺
		31	servlet_getQueryStringServlet	✓ ☺ -	✓ ☺ -	● ☺ -	✓ 🕱 -	NO ANAL.	● ☺ -
		16	servlet_listen_tcp	● ☺ -	● ☺ -	✓ ☺ -	✓ 🕱 -	NO ANAL.	● ☺ -
TOTAL TRUE POSITIVES				**1**	**6**	**7**	**11**		**1**
TOTAL FALSE POSITIVES				**0**	**6**	**8**	**15**		**2**

Table 48. Test cases CWE 83 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
83	**XSS_Attribute**	10	servlet_connect_tcp	● ☺ ☺	● ☺ ☺	✓ 🕯 🕯	✓ 🕯 🕯	NO ANAL.	● ☺ ☺
8 TEST CASES T. POSITIVES **13 TEST CASES F. POSITIVES**		42	servlet_console_readline	● ☺ -	● ☺ -	✓ ☺ -	● ☺ -	NO ANAL.	● ☺ -
		08	servlet_environment	● ☺ ☺	● ☺ ☺	● ☺ ☺	✓ 🕯 🕯	NO ANAL.	● ☺ ☺
		14	servlet_fromDB	● ☺ ☺	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	NO ANAL.	● ☺ ☺
		06	servlet_fromFile	● ☺ ☺	● ☺ ☺	● ☺ ☺	✓ 🕯 🕯	NO ANAL.	● ☺ ☺
		11	servlet_getCookiesServlet	● ☺ ☺	✓ 🕯 🕯	✓ 🕯 🕯	✓ 🕯 🕯	NO ANAL.	● ☺ ☺
		12	servlet_getCookiesServlet	● ☺ -	✓ ☺ -	✓ ☺ -	✓ 🕯 -	NO ANAL.	● ☺ -
		42	servlet_getCookiesServlet	● ☺ -	✓ ☺ -	✓ ☺ -	✓ 🕯 -	NO ANAL.	● ☺ -
TOTAL TRUE POSITIVES				**0**	**4**	**6**	**7**		**0**
TOTAL FALSE POSITIVES				**0**	**4**	**6**	**12**		**0**

Table 49. Test cases CWE 81 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
81	**XSS_Error_Message**	07	servlet_connect_tcp	✓ 🕱 🕱	● ☺ ☺	✓ 🕱 🕱	● ☺ ☺	NO ANAL.	NO
13 TEST CASES T. POSITIVES **23 TEST CASES F. POSITIVES**		04	servlet_console_readline	✓ 🕱 🕱	● ☺ ☺	● ☺ ☺	● ☺ ☺	NO ANAL.	
		05	servlet_environment	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺	NO ANAL.	
		08	servlet_fromDB	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	NO ANAL.	
		09	servlet_fromFile	✓ 🕱 🕱	● ☺ ☺	● ☺ ☺	● ☺ ☺	NO ANAL.	
		10	servlet_fromFile	✓ 🕱 🕱	● ☺ ☺	● ☺ ☺	● ☺ ☺	NO ANAL.	
		15	servlet_fromFile	✓ 🕱 🕱	● ☺ ☺	● ☺ ☺	● ☺ ☺	NO ANAL.	
		11	servlet_getCookiesServlet	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	NO ANAL.	
		51	servlet_getParameterServlet	✓ ☺ -	✓ ☺ -	✓ ☺ -	✓ ☺ -	NO ANAL.	
		10	servlet_getQueryStringServlet	✓ 🕱 🕱	✓ 🕱 🕱	● ☺ ☺	✓ 🕱 🕱	NO ANAL.	
		14	servlet_listen_tcp	✓ 🕱 🕱	● ☺ ☺	✓ 🕱 🕱	● ☺ ☺	NO ANAL.	
		52	servlet_getParameterServlet	● ☺ -	✓ ☺ -	✓ ☺ -	✓ 🕱 -	NO ANAL.	
		41	servlet_listen_tcp	● ☺ -	● ☺ -	● ☺ -	✓ 🕱 -	NO ANAL.	
TOTAL TRUE POSITIVES				**10**	**5**	**6**	**6**		
TOTAL FALSE POSITIVES				**18**	**6**	**8**	**8**		

Table 50. Test cases CWE 89 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
89	**SQL_Injection**	07	connect_tcp_execute	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	● ☺☺☺☺
19 TEST CASES T. POSITIVES **58 TEST CASES F. POSITIVES**		01	connect_tcp_executeBatch	✓ ☺ ☺	✓ ☺ ☺	✓ ☺ 😈	✓ 😈 ☺	✓ ☺ ☺	● ☺ ☺
		19	console_readLine_execute	✓ 😈 😈	✓ 😈 😈	✓ 😈 😈	✓ 😈 😈	● ☺ ☺	✓ ☺ ☺
		02	console_readLine_executeQuery	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	● ☺☺☺☺	● ☺☺☺☺
		14	Environment_execute	✓ 😈😈😈😈	✓ 😈😈😈😈	● ☺☺☺☺	✓ 😈😈😈😈	● ☺☺☺☺	✓ 😈😈😈😈
		19	Environment_executeBatch	✓ 😈 😈	✓ 😈 😈	● ☺ ☺	✓ 😈 😈	● ☺ ☺	● ☺ ☺
		19	Environment_executeQuery	✓ 😈 😈	✓ 😈 😈	● ☺ ☺	✓ 😈 😈	● ☺ ☺	● ☺ ☺
		05	fromDB_execute	✓ 😈😈😈😈	✓ 😈😈😈😈	● ☺☺☺☺	✓ 😈😈😈😈	✓😈😈😈 😈	● ☺☺☺☺
		51	fromDB_executeQuery	✓ ☺ ☺	✓ ☺ ☺	✓ ☺ ☺	✓ 😈 ☺	✓ ☺ ☺	● ☺ ☺
		17	fromDB_executeUpdate	✓ 😈 😈	✓ 😈 😈	● ☺ ☺	✓ 😈 😈	✓ 😈 😈	● ☺ ☺
		41	fromFile_execute	✓ ☺ ☺	✓ ☺ ☺	✓ ☺ 😈	✓ 😈 ☺	● ☺ ☺	✓ 😈 ☺
		04	fromFile_executeUpdate	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	● ☺☺☺☺	● 😈😈☺☺
		04	getCookiesServlet_execute	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓☺☺☺☺	✓ 😈😈☺☺
		14	getCookiesServlet_executeBatch	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	● ☺☺☺☺
		05	getCookiesServlet_executeUpdate	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	● ☺☺☺☺
		13	getParameterServlet_execute	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓☺☺☺☺	✓😈😈☺☺
		14	getParameterServlet_execute	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓ 😈😈😈😈	✓😈😈 😈😈	● ☺☺☺☺
		16	getParameterServlet_execute	✓ 😈 😈	✓ 😈 😈	✓ 😈 😈	✓ 😈 😈	✓ 😈 😈	● ☺ ☺
		51	getParameterServlet_executeBatch	✓ ☺ ☺	✓ ☺ ☺	✓ ☺ 😈	✓ 😈 ☺	✓ ☺ ☺	● ☺ ☺
TOTAL TRUE POSITIVES				**19**	**19**	**14**	**19**	**12**	**5**
TOTAL FALSE POSITIVES				**50**	**50**	**39**	**54**	**26**	**11**

Table 51. Test cases CWE 90 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
90	**LDAP_Injection**	13	servlet_connect_tcp	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	● ☺ ☺	● ☺ ☺	NO
11 TEST CASES T. POSITIVES **17 TEST CASES F. POSITIVES**		04	servlet_console_readline	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	● ☺ ☺	● ☺ ☺	
		10	servlet_environment	● ☺ ☺	✓ 🕱 🕱	● ☺ ☺	● ☺ ☺	● ☺ ☺	
		14	servlet_fromDB	✓ 🕱 🕱	✓ 🕱 🕱	● ☺ ☺	✓ 🕱 🕱	● ☺ ☺	
		12	servlet_fromFile	✓ ☺ -	✓ ☺ -	✓ ☺ -	● ☺ -	● ☺ -	
		13	servlet_getCookiesServlet	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	✓ 🕱 🕱	● ☺ ☺	
		16	servlet_getParameterServlet	✓ 🕱 -	✓ 🕱 -	✓ 🕱 -	● ☺ -	● ☺ -	
		15	servlet_getQueryStringServlet	● ☺ ☺	✓ 🕱 🕱	● ☺ ☺	✓ 🕱 🕱	● ☺ ☺	
		16	servlet_getQueryStringServlet	● ☺ -	✓ 🕱 -	● ☺ -	✓ 🕱 -	● ☺ -	
		41	servlet_getQueryStringServlet	✓ ☺ -	✓ 🕱 -	● ☺ -	✓ ☺ -	● ☺ -	
		12	servlet_listen_tcp	✓ ☺ -	✓ ☺ -	✓ ☺ -	● ☺ -	● ☺ -	
TOTAL TRUE POSITIVES				**8**	**11**	**5**	**5**	**0**	
TOTAL FALSE POSITIVES				**9**	**15**	**7**	**7**	**0**	

Table 52. Test cases CWE 113 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
113	**HTTP_Response_Splitting**	06	connect_tcp_addCookieServlet	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺	✓ ☺ ☺	● ☺ ☺
32 TEST CASES T. POSITIVES **88 TEST CASES F. POSITIVES**		16	connect_tcp_addHeaderServlet	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺	● ☺ ☺	✓ 🙁 🙁	● ☺ ☺
		31	connect_tcp_sendRedirectServlet	✓ ☺ 🙁	● ☺ ☺	✓ ☺ 🙁	● ☺ ☺	● ☺ ☺	● ☺ ☺
		07	connect_tcp_setHeaderServlet	✓🙁🙁🙁🙁	●☺☺☺☺	✓🙁🙁🙁🙁	●☺☺☺☺	✓🙁🙁🙁🙁	●☺☺☺☺
		14	console_readLine_addCookieServlet	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺
		12	console_readLine_addHeaderServlet	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺
		16	console_readLine_sendRedirectServlet	● ☺ ☺	● ☺ ☺	✓ 🙁 🙁	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺
		41	console_readLine_setHeaderServlet	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺
		16	Environment_addCookieServlet	● ☺ ☺	● ☺ ☺	✓ ☺ 🙁	● ☺ ☺	● ☺ ☺	● ☺ ☺
		16	Environment_addHeaderServlet	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺	● ☺ ☺
		51	Environment_sendRedirectServlet	● ☺ ☺	● ☺ ☺	● ☺ ☺	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺
		15	Environment_setHeaderServlet	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺
		13	fromDB_addCookieServlet	●☺☺☺☺	✓🙁🙁🙁🙁	●☺☺☺☺	✓🙁🙁🙁🙁	✓☺☺☺☺	●☺☺☺☺
		61	fromDB_sendRedirectServlet	● ☺ ☺	✓ ☺ 🙁	✓ ☺ 🙁	✓ 🙁 🙁	✓ ☺ ☺	● ☺ ☺
		07	fromFile_addCookieServlet	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺
		11	fromFile_addHeaderServlet	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺	●☺☺☺☺
		12	fromFile_sendRedirectServlet	● ☺ ☺	● ☺ ☺	✓ ☺ 🙁	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺
		16	fromFile_setHeaderServlet	● ☺ ☺	● ☺ ☺	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺	● ☺ ☺
		06	getCookiesServlet_addCookieServlet	●☺☺☺☺	✓🙁🙁🙁🙁	●☺☺☺☺	✓🙁🙁🙁🙁	✓☺☺☺☺	●☺☺☺☺
		12	getCookiesServlet_addHeaderServlet	✓ ☺ ☺	✓ ☺ 🙁	● ☺ ☺	✓ 🙁 🙁	✓ ☺ ☺	● ☺ ☺
		14	getCookiesServlet_sendRedirectServlet	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁	●☺☺☺☺
		05	getParameterServlet_addCookieServlet	●☺☺☺☺	✓🙁🙁🙁🙁	●☺☺☺☺	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁
		17	getParameterServlet_addHeaderServlet	✓ 🙁 🙁	✓ 🙁 🙁	● ☺ ☺	✓ 🙁 🙁	✓ 🙁 🙁	✓ 🙁 🙁
		19	getParameterServlet_sendRedirectServlet	✓ 🙁 🙁	✓ 🙁 🙁	✓ 🙁 🙁	✓ 🙁 🙁	✓ ☺ ☺	✓ ☺ ☺
		51	getParameterServlet_setHeaderServlet	✓ ☺ ☺	✓ ☺ 🙁	✓ ☺ 🙁	● ☺ 🙁	✓ ☺ ☺	● ☺ ☺
		06	getQueryStringServlet_addCookieServlet	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁	●☺☺☺☺	✓🙁🙁🙁🙁	●☺☺☺☺	●☺☺☺☺
		17	getQueryStringServlet_addCookieServlet	✓🙁🙁🙁🙁	✓ 🙁 🙁	● ☺ ☺	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺
		19	getQueryStringServlet_addCookieServlet	✓ 🙁 🙁	✓ 🙁 🙁	● ☺ ☺	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺
		11	getQueryStringServlet_addHeaderServlet	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁	●☺☺☺☺	✓🙁🙁🙁🙁	●☺☺☺☺	●☺☺☺☺
		15	getQueryStringServlet_sendRedirectServlet	✓🙁🙁🙁🙁	✓🙁🙁🙁🙁	●☺☺☺☺	✓ 🙁 🙁	●☺☺☺☺	●☺☺☺☺
		31	getQueryStringServlet_sendRedirectServlet	✓ ☺ 🙁	✓ ☺ 🙁	● ☺ ☺	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺
		41	getQueryStringServlet_sendRedirectServlet	✓ ☺ 🙁	✓ ☺ 🙁	● ☺ ☺	✓ 🙁 🙁	● ☺ ☺	● ☺ ☺
TOTAL VERDADEROS POSITIVOS				**15**	**16**	**10**	**18**	**12**	**3**
TOTAL FALSOS POSITIVOS				**35**	**41**	**19**	**49**	**16**	**6**

Table 53. Test cases CWE 352 [Bermejo, 2011]

CWE	DESC	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
352	**Cross_Site_Request_Forgery**	07	GetCookiesServlet	✓ 0 0 0 0	✓ 0 0 0 0	✓ 0 0 0 0	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	● ☺ ☺ ☺ ☺
7 TEST CASES T. POSITIVES **20 TEST CASES F. POSITIVES**		16	GetCookiesServlet	✓ 0 0	✓ 0 0	✓ ☺ 0	✓ 0 0	● ☺ ☺	● ☺ ☺
		17	GetCookiesServlet	✓ 0 0	✓ 0 0	✓ ☺ 0	✓ 0 0	● ☺ ☺	● ☺ ☺
		14	getParameterServlet	✓ 0 0 0 0	✓ 0 0 0 0	✓ 0 0 0 0	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	● ☺ ☺ ☺ ☺
		42	getParameterServlet	✓ ☺ ☺	✓ ☺ 0	✓ ☺ 0	✓ ☺ 0	● ☺ ☺	● ☺ ☺
		03	getQueryStringServlet	✓ 0 0 0 0	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	● ☺ ☺ ☺ ☺
		71	getQueryStringServlet	✓ ☺ 0	✓ ☺ 0	● ☺ ☺	✓ ☺ 0	● ☺ ☺	● ☺ ☺
TOTAL TRUE POSITIVES				**7**	**7**	**5**	**7**	**0**	**0**
TOTAL FALSE POSITIVES				**17**	**18**	**11**	**18**	**0**	**0**

Table 54. Test cases CWE 566 [Bermejo, 2011]

CWE	DESC	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
566	**Access_Through_SQL Primary**	01	servlet	✓ ☺ 0	✓ ☺ 0	● ☺ ☺	✓ ☺ 0	● ☺ ☺	● ☺ ☺
10 TEST CASES T. POSITIVES **30 TEST CASES F. POSITIVES**		03	servlet	✓ 0 0 0 0	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	● ☺ ☺ ☺ ☺
		05	servlet	✓ 0 0 0 0	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	● ☺ ☺ ☺ ☺
		07	servlet	✓ 0 0 0 0	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	● ☺ ☺ ☺ ☺
		09	servlet	✓ 0 0 0 0	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	● ☺ ☺ ☺ ☺
		12	servlet	✓ 0 0	✓ ☺ 0	● ☺ ☺	✓ 0 0	● ☺ ☺	● ☺ ☺
		14	servlet	✓ 0 0 0 0	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	✓ 0 0 0 0	● ☺ ☺ ☺ ☺	● ☺ ☺ ☺ ☺
		16	servlet	✓ 0 0	✓ 0 0	● ☺ ☺	✓ 0 0	● ☺ ☺	● ☺ ☺
		19	servlet	✓ 0 0	✓ 0 0	● ☺ ☺	✓ 0 0	● ☺ ☺	● ☺ ☺
		66	servlet	✓ ☺ 0	✓ ☺ 0	● ☺ ☺	✓ ☺ 0	● ☺ ☺	● ☺ ☺
TOTAL TRUE POSITIVES				**10**	**10**	**0**	**10**	**0**	**0**
TOTAL FALSE POSITIVES				**28**	**27**	**0**	**28**	**0**	**0**

Table 55. Test cases CWE 601 [Bermejo, 2011]

CWE	DESCRIPCIÓN	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
601	**Open_Redirect_Servlet**	02	servlet_connect_tcp	✓ 🕷 🕷	● ☺ ☺	✓ 🕷 🕷	✓ 🕷 🕷	✓ ☺ ☺	● ☺ ☺
11 TEST CASES T. POSITIVES **17 TEST CASES F. POSITIVES**		45	servlet_console_readline	● ☺ -	● ☺ -	● ☺ -	✓ 🕷 -	● ☺ -	● ☺ -
		08	servlet_environment	● ☺ ☺	● ☺ ☺	● ☺ ☺	✓ 🕷 🕷	● ☺ ☺	● ☺ ☺
		11	servlet_fromDB	● ☺ ☺	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	● ☺ ☺
		17	servlet_fromFile	● ☺ -	● ☺ -	✓ 🕷 -	✓ 🕷 -	● ☺ -	● ☺ -
		13	servlet_getCookiesServlet	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	● ☺ ☺
		06	getParameterServlet	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	✓ ☺ ☺	✓ ☺ -
		19	getParameterServlet	✓ 🕷 -	✓ 🕷 -	✓ 🕷 -	✓ 🕷 -	✓ ☺ -	✓ ☺ -
		31	getParameterServlet	✓ ☺ -	✓ ☺ -	✓ ☺ -	✓ 🕷 -	✓ ☺ -	✓ ☺ -
		45	getQueryStringServlet	✓ ☺ -	✓ ☺ -	✓ ☺ -	✓ 🕷 -	● ☺ -	● ☺ -
		14	lisen_tcp	✓ 🕷 🕷	● ☺ ☺	✓ 🕷 🕷	✓ 🕷 🕷	✓ 🕷 🕷	● ☺ ☺
TOTAL TRUE POSITIVES				**7**	**6**	**9**	**12**	**7**	**3**
TOTAL FALSE POSITIVES				**9**	**7**	**12**	**17**	**6**	**0**

Table 56. Group 2 test cases for vulnerability coverage analysis. [Bermejo, 2011]

CWE	DESC	FLOW	DATA SOURCE	Checkmarx	F360	Klocwork	Lapse+	Veracode	Findbugs
209	Information_Leak_Error	54	PropertiesFile	✓ 0 ☺		ND	ND		ND
		61	PropertiesFile						
256	Plaintext_Storage_of Password	66	PropertiesFile	ND	✓ 0 ☺	✓ 0 ☺	ND	✓ ☺ ☺	ND
		67	PropertiesFile		✓ 0 ☺	✓ 0 ☺		✓ 0 ☺	✓ 0 ☺
257	Storing_Password Rec._Format	68	Servlet_connect_tcp	ND		ND	ND	ND	ND
		71	Servlet_connect_tcp						
259	Hard_Coded_Password	1	PasswordAuth			ND	ND		ND
		2	PasswordAuth						
293	Using_Referer_Field_for Auth.	3	Servlet	ND		ND	ND	ND	ND
		4	Servlet						
315	Plaintext_Storage_in_a Cookie	6	Servlet	ND	✓0000	ND	ND	✓0000	ND
		7	Servlet		✓0000			✓0000	✓0000
319	Plaintext_Tx_Sensitive_Info	7	Servlet	ND		ND	ND		ND
		8	Servlet						
321	Hard_Coded_Cryptographic Key	10	Basic			ND	ND	✓ 0 0	ND
		11	Basic					✓ 0 0	✓ 0 0
327	Use_Broken_Crypto	12	Basic	✓ 0 -	✓ ☺ ☺	ND	ND		ND
		13	Basic	✓ 0 0	✓ 0 0				
328	Reversible_One_Way_Hash	13	Basic	✓ 0 0	✓ 0 0	ND	ND	ND	ND
		14	Basic	✓ 0 0	✓ 0 0				
330	Insufficiently_Random Values	17	Basic	ND	✓ 0 -	✓ 0 -	ND	✓ 0 -	ND
		19	Basic		✓ 0 -	✓ 0 -		✓ ☺ -	✓ ☺ -
336	Same_Seed_in_PRNG	01	basic	ND		ND	ND	ND	ND
		02	basic						
338	Weak_PRNG	05	Math	ND	✓ 0 0	✓ 0 0	ND	✓ 0 0	ND
		06	Math		✓ 0 0	✓ 0 0		✓ ☺ ☺	✓ ☺ ☺
367	TOC_TOU	17	basic	ND		ND	ND		ND
		19	basic						
378	Creation_of_File_with Insec_Per	09	basic	✓ 0 0		✓ 0 0	ND	✓ 0 0	ND
		10	basic	✓ 0 0		✓ 0 0		✓ 0 0	✓ 0 0
413	Insufficient_Resource Locking	01	console_reentrant_function_unsync	ND		ND	ND	ND	ND
476	NULL_Pointer_Dereference	01	undefinedValueServlet	ND		✓ 0 -	ND	ND	ND
		02	undefinedValueServlet			✓ 0 ☺			
489	Leftover_Debug_Code	03	Servlet		✓ ☺ ☺		✓ 0 0	✓ ☺ ☺	ND
		04	Servlet		✓ ☺ ☺		✓ 0 0	✓ ☺ ☺	✓ ☺ ☺
497	Information_Leak_SystemData	17	leakPathServlet	ND	✓ 0 -	✓ 0 -	ND		ND
		19	leakPathServlet		✓ 0 -	✓ 0 -			
523	Unprotected_Cred_Transport	41	Servlet	ND		ND	ND	ND	ND
		42	Servlet						
547	Hardcoded_Security Constants	10	Basic			ND	ND	ND	ND

		11	Basic						
549	**Missing_Password_Masking**	12	Servlet	ND		ND	ND	ND	ND
		13	Servlet						
567	**Unsynchronized_Shared_Data**	01	Servlet			✓ 🕴 -	ND	ND	ND
572	**Call_Thread_run_Instead start**	16	Basic	✓ 🕴 -	✓ 🕴 -	✓ 🕴 -	ND	ND	✓ 🕴 -
		17	Basic	✓ 🕴 -	✓ 🕴 -	✓ 🕴 -			
598	**Information_Leak QueryString**	07	Servlet	ND		ND	ND	ND	ND
		08	Servlet						
603	**Client_Side_Authentication**	07	Servlet	ND		ND	ND	ND	ND
		08	Servlet						
613	**Insufficient_Session Exp.**	17	Servlet	ND		ND	ND	ND	ND
		19	Servlet						
614	**Sensitive Cookie Without Secure**	13	Servlet	✓ 🕴 🕴	✓ 🕴 🕴	ND	ND	✓ ☺ ☺	ND
		14	Servlet	✓ 🕴 🕴	✓ 🕴 🕴				
615	**Info_Leak_By_Comment**	07	Servlet	ND	✓ 🕴 🕴	ND	ND	ND	ND
		08	Servlet		✓ 🕴 🕴				
643	**Unsafe_Treatment_XPath Input**	68	getQueryStringServlet	✓ ☺ 🕴	✓ ☺ 🕴	ND	✓ 🕴 🕴	ND	ND
		71	getQueryStringServlet	✓ ☺ 🕴			✓ 🕴 🕴		
759	**Unsalted_One_Way_Hash**	12	Basic	ND		ND	ND	ND	ND
		13	Basic						
760	**Predictable_Salt_One_Way Hash**	66	Environment	ND		ND	ND	ND	ND
		67	Environment						
TOTAL TRUE POSITVES				**14**	**23**	**15**	**4**	**15**	**1**
TOTAL FALSE POSITIVES				**20**	**36**	**18**	**8**	**19**	**1**
VULNERABILITY CATEGORIES A TOOL IS NDT DESIGNED TO DETECT (OF 32 VULNERABILITTY CATEGORIES)				**20**	**0**	**23**	**31**	**18**	**31**
NUMBER OF VULNERABILITIES ND DETECTED BY ANY TOOL				**30 of 62 total vulnerabilities**					