Universiteit Utrecht

Faculty of Science
Department of Information and Computing Sciences
MSc in Computing Science

Master Thesis

SUPPORTING AND AUTOMATING THE SECURITY ASSESSMENT OF SOFTWARE PRODUCTS USING TOOLS

Nikolaos Savvidis
Student ID: 3435784
n.savvidis@students.uu.nl

Thesis Supervisors:

Jurriaan Hage
j.hage@uu.nl

Wishnu Prasetya
s.w.b.prasetya@uu.nl

Internship Supervisors:

Haiyun Xu
h.xu@sig.eu

Joost Visser
j.visser@sig.eu

February 2014

ABSTRACT

IT security incidents are increasingly frequent, increasingly costly and increasingly difficult to prevent. To bring software security to a higher level, international standards like the ISO/IEC 25010 [1] have been developed to address security issues for software quality. This standard provides a powerful framework for analysing software quality aspects, one of which is security. Software Improvement Group (SIG) has proposed a security product quality model that operationalises the ISO/IEC 25010.

Our work started with studying and analysing this security model. The goal was to propose tools that could enhance and support the process of applying this model, since currently the tool support for this is minimal and most of the work is done manually. We broke down the process of applying the security model into steps and identified the steps that could benefit from using tools. We proceeded in looking for already available tools that fit our purposes, as well as in finding ways for measuring their effectiveness.

Our research did not lead us to a tool that was suitable for one of the steps of applying the SIG security model, so we designed and implemented one. We used an internal application of SIG as a ground truth for our tool development, and then tested it on two real-life projects of SIG. The tests were performed by experienced technical consultants of the company, and we used the results and their feedback to evaluate the degree of applicability and usefulness of our tool.

PREFACE

This thesis was submitted in partial fulfillment of the requirements for the degree of Master of Science in Computing Science of Utrecht University. It is the outcome of my work at the Software Improvement Group (SIG) over the past several months. The experience I gained during this internship is vast and multi-dimensional. It gave me insight into how a company works and handles real-life problems, while constantly improving software products according to the latest international standards.

I would like to thank Dr. Joost Visser for giving me the opportunity to work at SIG. Along with Dr. Haiyun Xu, they supported and guided me throughout the duration of my internship, while Jeroen Heijmans guided me in all the technical aspects and challenges that I faced. I would also like to thank Dr. Jurriaan Hage from Utrecht University, who stood by me and helped me with on-point comments, as well as useful ideas and strategies in approaching the different parts of my thesis. My thanks also go out to Dr. Wishnu Prasetya for the feedback and evaluation he has given me. Great appreciation also goes to many others - if not everybody - at SIG, for a lot of constructive conversations during coffee and lunch breaks and for creating a pleasant and friendly working environment.

Special thanks go to my friends for always being there for me. Without you my life would not be the same and all the experiences we shared will always be fondly remembered by me.

Finally, none of this would have been possible without the endless and unconditional love that I receive from my family. Your encouragement and belief in me (as well as your criticism when and where appropriate) has truly been inspiring and your support is what brought me to where I am. For this I am forever grateful.

CONTENTS

1

INTRODUCTION

1.1 CONTEXT

Companies and organizations rely more and more on software products. However, this also comes with greater risks: software can fail and even worse, software can be intentionally made to fail by attackers [2]. This creates the need to build security into the software product itself, and also to find ways to measure software security assurance [3] [4] [5]. This is the reason why many organisations hire consultancy companies with specialisation in assessment and improvement of information systems, to help them carry out and manage upgrading their products to the desireable security standards. One of the first steps in this process is the evaluation of the organisation's existing systems.

The International Organization for Standardization (ISO - `www.iso.org`), originally published in 1991 the ISO/IEC 9126 standard [6], which was revised in 2001. In this standard, they define six software quality characteristics, one of which is *Functionality*, with one of its sub-characteristics being *Security*. In 2011, ISO issued the updated standard for software product quality ISO/IEC 25010 [1] and one major change was to include *Security* as one of the main software product quality characteristics. According to ISO, the *Security* characteristic is defined as *"the degree to which a product or system protects information and data so that persons or other product systems have the degree of data access appropriate to their types and levels of authorization"*.

The Software Improvement Group (SIG) is an IT consultancy firm that bases its work on this and other ISO/IEC standards. However, as we will discuss in Section 2.1, the models provided by ISO are not specific and cannot be used as is to evaluate software product quality, complying with the requirements of SIG:

- The model has to be applicable within the duration of a short project (typically 6 weeks).
- The inputs for the model shoud be as factual as possible.
- The results of the model should enable root-cause analysis [7].

For this, SIG has proposed a new security product quality model that makes ISO/IEC 25010 operational [8].

1.2 PROBLEM DESCRIPTION

The SIG Security Model is described in detail in Section 2.2. The biggest challenge of applying the model is that it requires a lot of manual work. Currently, SIG has developed and uses a tool that supports the aggregation of the findings and obtaining an overall rating. However, as we will analyse in Section 2.3, there is no tool support for the other steps of the Security Model. The evaluation relies on getting information from the software architects of the system to be inspected, the documentation of the system and also a system demo given by the developers. But since security is a very delicate subject where a minor miss of a system component might jeopardize the security of the whole system, SIG cannot rely solely on these information sources to ensure that the results of their work are complete.

To have a complete picture, SIG manually looks into the source code. This procedure, not only requires a lot of time and manual labor, but furthermore it might lead to non-repeatable results. Furthermore, there is still the possibility of missed flaws and vulnerabilities of the product by the evaluator. With proper tool support, the evaluation can become more sound and complete, and the steps performed by the evaluator will be repeatable.

1.3 RESEARCH QUESTIONS

In this research we try to provide a solution to the problem stated above. The objective of the research is hence stated as follows:

> “*The objective of this study is to investigate possible improvements in the process of evaluating the software product quality in regards to security, using the SIG Security Model and integrating tool support in the steps of the evaluation process where its useful.*”

The aforementioned problem and the objective that needs to be reached form our main research question:

RQ: What tools can be used for enhancing or automating the evaluation process for software quality regarding security?

This main research question can be broken down into some more detailed sub-questions:

SQ1: *Which parts/steps of the evaluation process can be supported by tools?*
SQ2: *Which tools are currently available that can be used?*
SQ3: *Which tool(s) are more suitable for the purposes of SIG?*
SQ4: *If no tool is available for a certain step of the process, can we specify, design and implement one?*
SQ5: *How can we measure and evaluate the impact of tool support in the evaluation process?*

1.4 COMPANY PROFILE

In the context of this thesis, the author worked as an intern at the company: Software Improvement Group - SIG (`http://www.sig.eu`). The company presents itself as: "*a consultancy firm that provides impartial, objective, verifiable and quantitative assessments of risks related to your corporate software systems. Our analysis software allows us to measure quality, to lay bare the underlying architecture and to assess the risks related to a system.*"

The company provides a number of services related to consultancy. Among others, these are:

- *Application portfolio analysis*: gives insights into the cost and risks of application portfolios and identifies opportunities for optimisation;
- *Project risk estimation*: gives insights into IT project risks and increases the likelihood of project success;
- *Software risk assessment*: helps reduce or eliminate technical and operational issues by identifying software system risks at an early stage;
- *Software risk monitor*: monitoring systems under development and maintenance;
- *Security risk assessment*: helps prevent security incidents by identifying the root causes of weak spots in code, design and process.

The company consists of a professional team of consultants of two types: *"Technical Consultants"* who are responsible for evaluating the quality of a software system from a client company, and *"General Consultants"* who receive the evaluation result from the technical consultants, translate it into business risks or objectives and advise the client company accordingly.

To perform structured and complete evaluations, the technical consultants rely on the in-house built models. For example, to perform a security risk assessment, the evaluator applies the SIG Security Model. As we can see from the company's profile described above, software security assessment is one of the key services it provides and a research topic like the one we are dealing with, directly serves the needs of the company.

2 RELATED LITERATURE

In this part of the thesis we present all the related material that was studied in order to become familiar with the research area.

2.1 THE ISO STANDARD

The Software Improvement Group wants to base their models on international standards and best practices. For this reason, they follow the guidelines of the International Organization for Standardization (ISO - `www.iso.org`) and the International Electrotechnical Commision (IEC - `http://www.iec.ch`). ISO and IEC technical committees collaborate in fields of mutual interest, and they have issued international standards and technical reports regarding Software Engineering and Software product Quality, which also included *Security*. We examine these standards and reports in Sections 2.1.1 and 2.1.2.

2.1.1 *ISO/IEC 9126*

The ISO/IEC 9126 *Software engineering — Product quality* [6] was initially published in 1991 and then revised in 2001 which divided it into four parts: the quality model (ISO/IEC 9126-1), the external metrics (ISO/IEC 9126-2), the internal metrics (ISO/IEC 9126-3) and the quality in use metrics (ISO/IEC 9126-4). In the quality model, it defines six software quality characteristics for internal and external quality:

1. *Functionality*: The capability of the software product to provide functions which meet stated and implied needs when software is used under specified conditions.

2. *Reliability*: The capability of the software product to maintain a specified level of performance when used under specified conditions.

3. *Usability*: The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.

4. *Efficiency*: The capability of the software product to provide appropriate performance, relative to the amount of resources used, under the stated conditions.

5. *Maintainability*: The capability of the software product to be modified. Modifications may include corrections, improvements or adaptions of the software changes in environment, and in requirements and functional specifications.

6. *Portability*: The capability of the software product to be transferred from one environment to another.

The ISO also decomposes these characteristics into subcharacteristics. The subcharacteristic of *Security* falls under *Functionality.*

The internal metrics presented in ISO/IEC 9126-2 may be applied to a non-executable software product during its development stages and they provide users with the ability to measure the quality of the intermediate deliverables and thereby predict the quality of the final product. This allows the user to identify quality issues and initiate corrective action as early as possible in the software development life cycle. The external metrics presented in ISO/IEC 9126-3 may be used to measure the quality of the software product by measuring the behaviour of the system of which it is a part. The external metrics can only be used during the testing stages of the life cycle process and during any operational stages. However, the technical reports providing the metrics (ISO/IEC 9126-2 and ISO/IEC 9126-3) for the quality model, do not assign ranges of values of these metrics to rated levels or to grades of compliance. The reason for this is that these values are defined for each software product by its nature. Also, the list of metrics is described by the ISO/IEC standard itself to be non-exhaustive.

The internal metrics defined for the characteristic of *Functionality* are used for predicting if the software product in question will satisfy prescribed functional requirements and implied user needs. Specifically, the metrics defined for the *Security* subcharacteristic indicate a set of attributes for assessing the capability of the software product to avoid illegal access to the system and/or data. Four metrics are defined:

1. *Access auditability*: To measure how easily is the access login can be inspected.

2. *Access controllability*: To measure how controllable is access to the system.

3. *Data corruption prevention*: To measure how complete is the implementation of data corruption prevention.

4. *Data encryption*: To measure how complete is the implementation of data encryprion.

2.1.2 *ISO/IEC 25010*

The ISO/IEC 9126 was canceled and replaced by the ISO/IEC 25010 [1]. The ISO/IEC 25010 is part of the Systems and software Quality Requirements and Evaluation (SQuaRE) series of International Standards, which consists of the following divisions:

- Quality Management Division (ISO/IEC 2500n)

- Quality Model Division (ISO/IEC 2501n)
- Quality Measurement Division (ISO/IEC 2502n)
- Quality Requirements Division (ISO/IEC 2503n)
- Quality Evaluation Division (ISO/IEC 2504n)
- SQuaRE Extension Division (ISO/IEC 25050 - ISO/IEC 25099)

This new international standard revises the old 9126 and incorporates the same software quality characteristics with some amendments. One of these amendments is the addition of *Security* as a characteristic, rather than a subcharacteristic of *Functionality*. A breakdown of the Quality Model into its eight characteristics as defined by the ISO 25010 is presented in Figure 1.

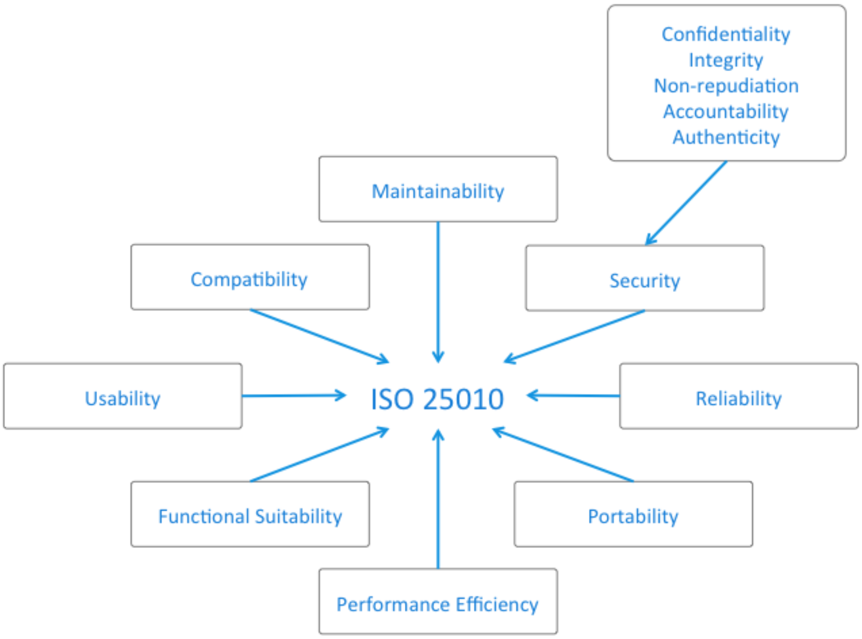


Figure 1: A breakdown of the ISO/IEC 25010 Quality Model into it's eight characteristics. The *Security* characteristic is also analysed into it's subcharacteristics.

The *Security* characteristic is defined as *"the degree to which a product or system protects information and data so that persons or other products or systems have degree of data access appropriate to their types of authorization"*. Furthermore, the *Security* characteristic is divided into five subcharacteristics which are defined as:

- Confidentiality: the degree to which a product or system ensures that data are accessible only to those authorized to have access.

- Integrity: the degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.
- Non-repudiation: the degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
- Accountability: the degree to which the actions of an entity can be traced uniquely to the entity.
- Authenticity: the degree to which the identity of a subject or resource can be proven to be the one claimed.

The quality model that is presented in ISO/IEC 25010, referes to the measurement reference model which is defined by ISO/IEC 25020 [9]. According to this model, the quality properties are measured by applying a measurement method, the result of which is a quality measure element. The quality characteristics and subcharacteristics can be quantified by applying measurement functions. The result of applying a measurement function is called a software quality measure. In this way, software quality measures become quantifications of the quality characteristics and subcharacteristics. This procedure is shown in Figure 2.

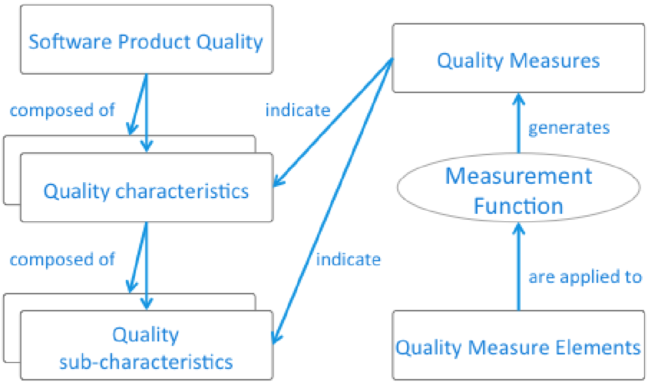


Figure 2: Software product quality measurement reference model, as defined in ISO 25020 [9]

However, the methodology described in ISO/IEC 25020 is independent of specific characteristics. In fact, ISO/IEC 25020 only provides a structure which can be applied in measuring quality, no matter what the quality aspects are. To operationalise the quality model of ISO 25010 and use the measurement model of ISO/IEC 25020, proper quality measure elements must be defined that can be measured to indicate the quality of certain characteristics or sub-characteristics. The Technical Report ISO/IEC 25021 [10] presents a description of the concept of quality measure elements and some consideration for using quality measure elements.

2.2 THE SIG SECURITY MODEL

The main goal of the SIG Security Model is to make the ISO/IEC 25010 operational. The outcome of the model is a star rating, expressed in 1 to 5 stars, and this gives an expression of the quality of the implementation. Even though the model tries to impose a structure and repeatability of all the findings, ultimately, the rating is a form of expert opinion. The star rating reflects the best security practices (or lack thereof) in a system. Namely, the stars are interpreted in the following way:

- ★☆☆☆☆ : several or many poor practices found in the system
- ★★☆☆☆ : some poor practices applied
- ★★★☆☆ : no poor practices, sufficient practices applied throughout the system
- ★★★★☆ : sufficient or best practices applied
- ★★★★★ : best practices applied throughout the system

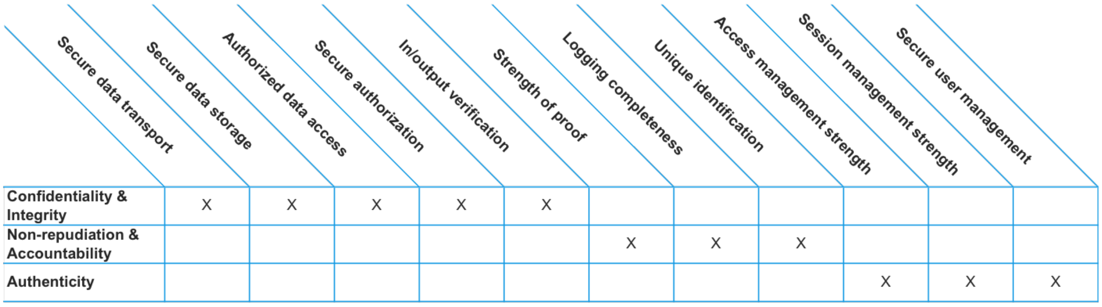
As we mentioned earlier in Section 2.1, *Security* is one of the eight quality characteristics defined by ISO/IEC 25010. Furthermore, ISO/IEC 25010 breaks down the security characteristic into five sub-characteristics: confidentiality, integrity, non-repudiation, accountability and authenticity.

In the SIG security model, they have combined the characteristics *Confidentiality* with *Integrity* and *Non-repudiation* with *Accountability*, since they share many attributes in their definitions in ISO/IEC 25010. In practice, they are often addressed together during software development. To operationalize the model and to make the subcharacteristics measurable, the SIG has defined eleven system properties which reflect how a typical software product addresses the desired security characteristics.

The mapping of the eleven system properties to the ISO/IEC 25010 security subcharacteristics is presented in Figure 3.

None of the 3 security sub-characteristics of the SIG Security Model is rated directly. Instead, all of them are divided in system properties, each of which is rated. For the *Confidentiality & Integrity* security sub-characteristic, the SIG model introduces the system properties:

1. Secure Data Transport



	Secure data transport	Secure data storage	Authorized data access	Secure authorization	In/output verification	Strength of proof	Logging completeness	Unique identification	Access management strength	Session management strength	Secure user management
Confidentiality & Integrity	X	X	X	X	X						
Non-repudiation & Accountability						X	X	X			
Authenticity									X	X	X

Figure 3: Mapping of system properties to the ISO/IEC 25010 security sub-characteristics, as proposed by [8]

.

2. Secure Data Storage
3. Authorised Data Access
4. Secure Authorisation
5. In/Output Verification

For the *Non-repudiation & Accountability* security sub-characteristic property, the model introduces the system properties:

1. Strength of Proof
2. Logging Completeness
3. Unique Identification

And for the *Authenticity* security sub-characteristic, the system properties are:

1. Access Management Strength
2. Session Management Strength
3. Secure User Management

In order to reach the final star rating, the system needs to be evaluated step by step:

1. The first step is to break down the system into the eight system elements, as defined in the model. These system elements are:
   a) System processes: running applications that are part of the system. They are logical processes and if the same process runs on multiple computers it is considered one process. Some examples are an application server and a desktop application.

b) User types: groups of users, both human and machine. Different types have different access rights in the system. Some examples are the system administrator and an end user.

c) Other systems: systems with which the system under investigation interacts. Most commonly these are third party software, hence they cannot be researched in the same detail as the system under investigation. Most common examples are an email server and an LDAP server.

d) Data stores: places where data is persisted within the system. Typical examples are databases and a filesystem.

e) Functions requiring non-repudiation: functions that can be performed with the system and need to be recorded and proven to have taken place. Examples are financial transactions and user login.

f) Communications channels: for all communication between system processes and any of the users, other systems and data stores there is a communication channel. For example user interaction with the application and HTTP requests from the web browser to the web application.

g) Sensitive data: determine where sensitive data are stored and via which communication channels they are transmitted. Which data are considered sensitive depend on the application, and its discussed with the customer. Typical examples are passwords and credit card numbers.

h) Authentication mechanisms: mechanisms that allow the users of the system to identify themselves. Examples are username/password, token machines and biometric data.

To illustrate these system elements, we present an example that is used in the documentation of the SIG Security Model. The example system is a web store, which allows users to purchase items. In addition, there is a mobile application in which a user can view the items on sale, but no purchase is possible via this application. The breakdown of this system into its elements as they are described above, is shown in Figure 4.

To achieve this breakdown, the consultants at SIG first have an interview session with the technical team that designed and implemented the system. They also look at the system documentation, and ask for a system demo. However, the interview method might not be complete because it relies on human recollection and the system under evaluation can be quite large. Therefore, for an accurate review and security assessment of a system, the consultants do not rely solely on this information, and go through the source code to identify and locate these elements.

2. Once the system elements are identified and located, the consultants go through the source code once more, to perform a system exploration. This step gives them

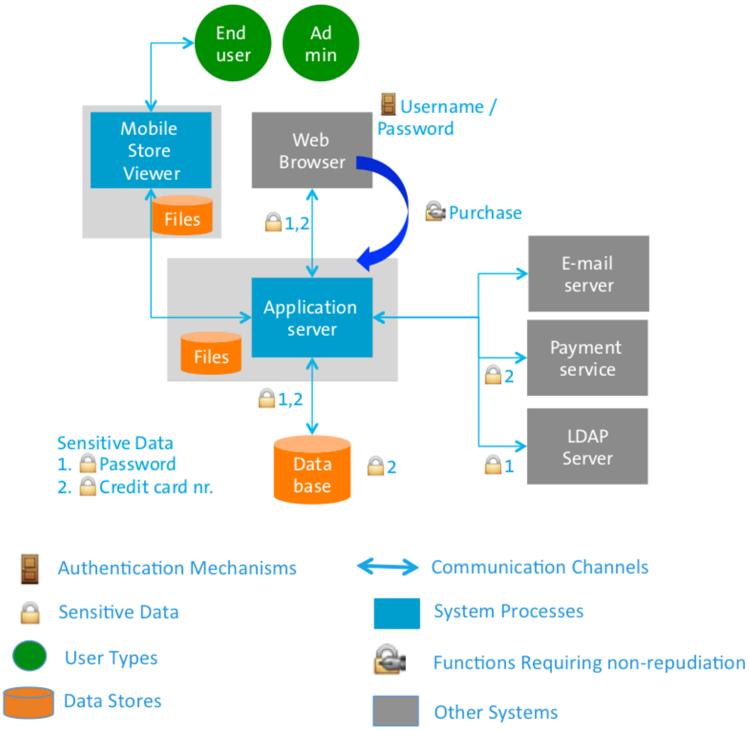


Figure 4: A visual breakdown of the example system as described in Section 2.2, taken from [8]

a general data flow picture of the system, which will be useful in the following steps. The main objective here is to locate references and usage points of the system elements discovered in the previous step.

3. The next step is to perform the security evaluation according to the eleven system properties shown in Figure 3. Instead of rating the properties directly, they are divided into sub-properties, that are rated. Then, the relevant system elements for each sub-property are identified, and then the system elements are rated for a sub-property. All findings are logged and documented, in order to allow for traceable results.

4. When all elements are rated, the result goes through an internal peer review. If another consultant has a different opinion on the evaluation process, then the consultants discuss and if necessary perform the security evaluation again, in order to ensure the most accurate results for the customer.

5. Finally, the ratings for each system element are aggregated to an overall rating for the entire system. In order to reflect the security principle of the *Weakest Link* [11], the SIG model suggests the use of the power mean [12] with a value of $p = -2$. The formula for the power mean M is:

$$M_p(x_1, \ldots, x_n) = \left(\frac{1}{n} \sum_{i=1}^{n} x_i^p \right)^{1/p}$$

   By using the power mean, the lower values have more impact on the final rating and it is more distinctive than using the *minimum* function, which also reflects the weakest link principle, but would disregard all other properties.

Having rated the system, the technical consultant with a general consultant validate the results with the customer.

2.3 POTENTIAL TOOL SUPPORT

According to the breakdown for applying the security model given above, we identify the following steps of the process that can benefit from tool support:

1. Extracting the system elements from the source code
2. Performing system exploration (data flow)
3. Performing code review for finding security vulnerabilities
4. Aggregating the findings and obtaining an overall star rating (this tool was already developed by SIG)

Therefore, apart from the last option, we had the other three to study and see if we can either find tools that can fit into the SIG model, or to design one in the case that no such tool is found.

Our work for the topic of elements extraction is presented in Chapters 4 and 5 of this thesis, while the validation of our work and some further discussion is presented in Chapters 6 and 7.

Regarding the topic of system exploration, there are many tools that perform data flow analysis in homogeneous systems. This unfortunately is not satisfying enough for SIG, since a quick browsing through the portfolio database, one can find over 70 different technologies involved in system projects and no system was composed by only 1 technology. A similar work was done by a former student of Universiteit Utrecht and intern at SIG and can be found in [13]. Upon consideration, we decided not to follow this approach and focus on the other two options.

Our findings from our research regarding the tools for code review are presented in detail in Chapter 3.

3 TOOLS FOR CODE REVIEW

From our research we found that there are quite a few tools, open-source and commercial, that perform code review. The tools we reviewed were: AppPerfect Java Code Test [14], FindBugs [15], CodePro Analytix [16], HP Fortify SCA [17], Veracode Static Analysis [18], Code Analysis for Managed Code [19], BugScout [20], Checkmarx [21], CodeSecure [22], Coverity SAVE [23], IBM Security AppScan Source [24] and Insight Defects [25]. Some support only one language (e.g. FindBugs, AppPerfect, CodePro Analytix), while others support a big variety of languages (e.g. HP Fortify, VeraCode, CheckMarx).

3.1 TOOL INFORMATION

The first thing that we would like to investigate is how much the tools correlate and overlap with the SIG security model. If there is not enough overlap of a tool with the security model, then the tool will not have a big contribution to SIG. To do this, we went through the documentation notes of every tool in order to find their list of rules.

The most commonly used reference for weaknesses and vulnerabilities is the Common Weakness Enumeration [26] ID numbers. The Common Weakness Enumeration (CWE) is a formal list of common software weaknesses that can occur in the architecture, design, code and implementation of software that can lead to exploitable security vulnerabilities. It 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; and to provide a common baseline standard for weakness identification, mitigation, and prevention efforts. It is maintained by MITRE [27], a not-for-profit organization that operates research and development centers sponsored by the government of the United States.

Every year, MITRE in collaboration with the SANS Institute [28] and other security experts publish a list with the top 25 most widespread and critical errors that can lead to serious vulnerabilities in software. We consulted this list (CWE/SANS Top 25 Most Dangerous Software Errors [29]) and we mapped it onto the SIG Security model. This mapping is shown in Table 1. From this table we can see that 2 CWE ID's are not mapped to the SIG Security Model. The first one is CWE with ID 676. According to its description, *"The program invokes a potentially dangerous function that could introduce* *a vulnerability if it is used incorrectly, but the function can also be used safely"*. An example of a potentially dangerous function is `strcpy` in C. This particular function can lead to buffer overflows, which is covered by the SIG Security Model in the In/Output Verification system property. However, other potentially dangerous functions can lead to other problems that fall under other system properties. This makes it hard to map this CWE to a single system property - theoretically it could correspond to all of the properties. The other CWE that was not mapped is CWE 829. The description given for this CWE is *"The software imports, requires, or includes executable functionality (such as a library) from a source that is outside of the intended control sphere"*. If a vulnerable library is found, this is indeed a risk. However, the model does not cover vulnerabilities in parts of the system that are not maintained by the developers team itself, i.e. they are not taken into account when looking at the implementation and the source code of third-party libraries is not analysed.

Having gained this experience, we then proceeded to map the tools on the SIG Security Model. Not all the vendors disclosed that information, and some of them refused to do that even after we contacted them. For the ones that we had the information, we mapped their advertised coverage to the SIG security model. This mapping is summarised in Table 2.

To illustrate at some degree the amount of rules that correspond to this mapping, we expressed it in a scale of ✓ to ✓✓✓. For 1 to 5 rules that correspond to a system property we used ✓, for 6 to 10 rules we used ✓✓ and for more than 10 we used ✓✓✓. However, this illustration only shows the coverage that the tool advertises related to that property. To properly evaluate this coverage, we must have hands-on experience with the tools and test them.

Apart from the mapping to the SIG model, in the table we included one more group of columns with other interesting properties:

- The column *"Scan uncompiled code"* shows whether the tool allows for the analysis to be done in source code without the need for compilation. With the exception of Veracode, all the other tools support this feature. Veracode scans the binary code instead of the source code, and while this might give some different insights to the vulnerabilites of the system, it's not suitable for the purposes of the SIG, since the audit is performed on the source code.

- The column *"Custom Rules"* indicates whether the tool provides the capability of extending its own rule set with our own custom rules. This capability can be very useful in systems with peculiar distinctive features.

CWE ID	CWE Name	Confidentiality & Integrity					Non-repudiation & Accountability			Authenticity		
		Secure Data Transport	Secure Data Storage	Authorised Data Access	Secure Authorisation	In/Output Verification	Strength of Proof	Logging Completeness	Unique Identification	Access Management Strength	Session Management Strength	Secure User Management
22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')			✓	✓							
78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')					✓						
79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')					✓						
89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')					✓						
120	Buffer Copy without Checking Size of Input					✓						
131	Incorrect Calculation of Buffer Size					✓						
134	Uncontrolled Format String					✓						
190	Integer Overflow or Wraparound					✓						
250	Execution with Unnecessary Privileges			✓	✓							
306	Missing Authentication for Critical Function									✓		
307	Improper Restriction of Excessive Authentication Attempts									✓		
311	Missing Encryption of Sensitive Data	✓	✓									
327	Use of a Broken or Risky Cryptographic Algorithm	✓	✓									
352	Cross-Site Request Forgery (CSRF)										✓	
434	Unrestricted Upload of File with Dangerous Type					✓						
494	Download of Code Without Integrity Check					✓						
601	URL Redirection to Untrusted Site ('Open Redirect')					✓						
676	Use of Potentially Dangerous Function											
732	Incorrect Permission Assignment for Critical Resource			✓								
759	Use of a One-Way Hash without a Salt		✓									
798	Use of Hard-coded Credentials		✓									
807	Reliance on Untrusted Inputs in a Security Decision				✓					✓		
829	Inclusion of Functionality from Untrusted Control Sphere											
862	Missing Authorization			✓								
863	Incorrect Authorization				✓							

Table 1: The CWE/SANS Top 25 [29] mapping on the SIG Security Model

	SIG Model											Other properties		
	Confidentiality & Integrity					Non-repudiation & Accountability			Authenticity					
	Secure Data Transport	Secure Data Storage	Authorised Data Access	Secure Authorisation	In/Output Verification	Strength of Proof	Logging Completeness	Unique Identification	Access Management Strength	Session Management Strength	Secure User Management	Scan Uncompiled Code	Custom Rules	Latest Version
AppPerfect Java Code Test					✓							✓	✓	13.0.0 2013
CodePro Analytix		✓	✓	✓	✓✓				✓✓			✓	✓	7.0.0 2010
FindBugs			✓	✓	✓✓				✓			✓	✗	2.0.2 2012
HP Fortify SCA		✓	✓✓	✓✓	✓✓✓		✓		✓✓	✓		✓	✓	2013
VeraCode Static Analysis		✓✓	✓✓	✓	✓✓✓				✓✓	✓		✗	✓	13.5 2013
CheckMarx	✓	✓✓	✓	✓	✓✓✓				✓✓✓	✓		✓	✓	6.2.8 2013

Table 2: System property coverage by the available tools

- Finally, the column *"Latest Version"* shows which is the latest version and when it was released. In this area, it is crucial to have a tool that is up to date since vulnerabilities are discovered daily.

3.2 TOOL EVALUATION

After we populated the list of tools for source code review, the next step would be to get some hands-on experience with them. For this purpose, we contacted the vendors and asked for an academic trial license. Unfortunately, only one of the vendors provided us with the opportunity to start the procedure of obtaining a license. However, testing only one tool would not give us a sufficient point of reference, so we decided to search for other sources and studies that performed this or a similar task. Our research lead us to work done by the Center for Assured Software (CAS) of the National Security Agency (NSA) and by the National Institute of Standards and Technology (NIST) [30] in the United States.

3.2.1 *The SAMATE and SATE projects*

One of NIST's projects is SAMATE (Software Assurance Metrics And Tool Evaluation) [31]. According to NIST, the SAMATE project *"is dedicated to improving software assurance by developing methods to enable software tool evaluations, measuring the effectiveness of tools and techniques, and identifying gaps in tools and methods"*. For this purpose, they developed tool specifications, test plans, and test sets. They also organize the Static Analysis Tools Exposition (SATE - `http://samate.nist.gov/SATE.html`).

3.2.1.1 *The CVE dictionary*

The MITRE organization, apart from the CWE database that we presented in Section 3.1, maintains also the Common Vulnerabilites and Exposures (CVE) [32]. The CVE is a dictionary of common names (i.e., CVE Identifiers) for publicly known information security vulnerabilities. Before it was created in 1999, most information security tools used their own databases with their own names for security vulnerabilities. This led to difficulties in determining whether different databases were referring to the same problem, creating potential gaps in security coverage. Furthermore, the tool vendors did not use standardized metrics and measures for the vulnerabilities they detected.

CVE's common, standardized identifiers provided the solution to these problems. According to [32], CVE is:

- One name for one vulnerability or exposure

- One standardized description for each vulnerability or exposure
- A dictionary rather than a database
- How disparate databases and tools can "speak" the same language
- The way to interoperability and better security coverage
- A basis for evaluation among tools and databases
- Free for public download and use
- Industry-endorsed via the CVE Editorial Board and CVE-Compatible Products

A typical entry in the CVE states which production system, and in which version, has a vulnerability or exposure along with a small description. For example, for CVE-2013-6407 (`http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-6407`) the description is: "*The UpdateRequestHandler for XML in Apache Solr before 4.1 allows remote attackers to have an unspecified impact via XML data containing an external entity declaration in conjunction with an entity reference, related to an XML External Entity (XXE) issue*". CVE provides a list of references confirming the vulnerability, as well as solutions (in this case a link to a patch fixing this vulnerabiltiy).

3.2.1.2 *The* Juliet *test suite and the SRD Database*

The National Institute of Standards and Technology (NIST) built a repository of software bugs to help application developers find weaknesses in their programming code. They call this the SAMATE Reference Dataset (SRD). According to [33], "*The purpose of the SAMATE Reference Dataset (SRD) is to provide users, researchers, and software security assurance tool developers with a set of known security flaws. This will allow end users to evaluate tools and tool developers to test their methods*". This dataset is a large-scale effort from many contributors and intends to encompass a wide variety of possible vulnerabilities, languages, platforms, and compilers. The test cases are freely available.

One of the test suites in the SRD is the Juliet test suite [34]. Juliet is a collection of C/C++ and Java programs with known flaws. It was originally released in 2010, and with its current version (v1.2, 2013) it comprises more than 60 thousand test cases in C/C++ and more than 25 thousand in Java. Each test case is very small and most of them are synthetic (i.e. created for use as test cases). The test cases are based on the Common Weakness Enumeration (CWE). The C/C++ test cases contain examples for 118 different CWEs, while the Java test cases cover 112. Each test case focuses on one type of flaw, but in addition to that, for every flaw in a test case one or more non-flawed methods or functions that perform a similar task are provided.

Juliet test cases can be analysed as a whole, by individual CWEs or by individual test case. This allows the users to investigate which static analysis tools are most effective at finding flaws that are important to them and the project they have at hand. The structure of the test cases allow for easy interpretation of the results. The flawed methods contain the keyword *"bad"* in their name, thus the classification of findings as True Positives or False Positives can be done easily. Furthermore, each test case clearly states which flaw it contains, so the user can easily identify False Negatives as well.

3.2.1.3 *SATE IV*

The Static Analysis Tool Exposition (SATE - `http://samate.nist.gov/SATE.html`) was designed to advance research in static analysis tools that defects in source code relevant to security. The first iteration of SATE was in 2008, and the latest completed iteration was SATE IV in 2012. The goals of SATE are:

- to enable empirical research based on large test sets,
- to encourage improvement of tools,
- to speed adoption of tools by objectively demonstrating their use on real software.

As stated in [30], the primary idea behind SATE is to understand how security-oriented static analyzers perform. The tool makers that participate in SATE are asked to run their tools on the test data provided to them, and then the results are analysed and reviewed by researchers of SAMATE. The exposition had 2 tracks: C/C++ and Java. The test data in both tracks included production open source programs based on entries in the CVE database, and the Juliet test suite. The CVE-selected test cases included a vulnerable version with publicly reported vulnerabilites in the CVE database, as well as a fixed, newer version where some or all of the CVEs were fixed.

In SATE IV, there were 2 participant teams in the Java track and 6 in the C/C++ track (one team participated in both tracks). However, 1 team from each track analyzed only the CVE-selected test cases and not the Juliet test cases. In total, 28 tool runs for the CVE-selected programs were analysed. This counts tool outputs for vulnerable and fixed versions of the same CVE-selected program separately. Also, 5 tool runs for the C/C++ Juliet test cases and 1 run for the Java Julet test cases were analysed.

Due to the vast amount of warnings for the CVE-selected cases in the tool reports, only a random sample of less than 3% was analysed. The sampling was based on findings by security experts and based on CVEs. One conclusion drawn from the analysis was that tools mostly find different weakneses, since over 2/3 of the weaknesses were reported by one tool only. The researchers attributed this to three reasons. One is the limited participation, in particular only two tools were run on the Java test cases. Another reason is that tools look for different weakness types. Finally, while there are many weaknesses in large software, only a relatively small subset may be reported by tools. The biggest overlap in the tool findings, as well as the highest proportion of related warnings was for some well known and well studied categories, mainly in the group of input validation (buffer errors, XSS, path traversal).

3.2.1.4 *SATE V*

At the time of the writting of this thesis, the latest iteration of SATE was ongoing. The analysis of the results begun in September 2013, while the experience workshop was announced to take place on March 14 2014. The NIST report for SATE V is expected in October 2014. The main changes from SATE IV are:

- Tool outputs and the detailed analysis of tool warnings will not be released in order to encourage wider participation.
- Teams should run their tools within the Software Assurance Marketplace (SWAMP - `http://www.cosalab.org/`). SWAMP is a research facility funded by the Department of Homeland Security in the United States of America. For SATE V, they hosted the infrastructure to support the testing.
- Teams should provide a Coverage Claims Representation (CCR - `http://cwe.mitre.org/compatible/ccr.html`) of the weaknesses their tool can find. CCR is a means for software analysis vendors to convey to their customers exactly which CWE-identified weaknesses they claim to be able to locate in software.

3.2.2 *The CAS Static Analysis Tool Study - Methodology*

In order to address the growing lack of software assurance in the Department of Defense in the USA, the National Security Agency's Center for Assured Software (CAS) was created in 2005. CAS regularly conducts scientific studies that measure and rate the effectiveness of static analysis tools in a repeatable manner. The CAS Static Analysis Tool Study Methodology [35], contrary to the approach taken by SATE, is based solely on the artificial test cases of the Juliet test suite.

In the Methodology proposed by CAS, every tool is executed on the test cases of the Juliet suite. Each result is then classified as a True Positive, False Positive or False Negative. The False Negative classification is not actually a tool result. As we described in Section 3.2.1.2, each test case contains one True Positive, so for every test case that no True Positive was reported by the tool, a False Negative result is added. To perform the analysis of the tool results, CAS then uses the metrics of *precision*, *recall* and *F-score* [36][37]. We will further discuss these metrics in Section 4.3 of this thesis.

CAS also uses another feature of the Juliet test suite that we described in Section 3.2.1.2 to analyse the tool results. As we described above, for every flaw in a test case one or more non-flawed methods or functions that perform a similar task are provided. To use this, the CAS Methodology introduces the metrics of *Discriminations* and *Discrimination Rate*. The former is a binary metric. For every test case, if the tool reported the flaw (True Positive) but not the non-flawed code (False Positive), then it receives 1 *Discriminations*, otherwise it receives 0. This way, the methodology separates the tools that do simple pattern matching, from tools that perform more complex analysis. The definition of the *Discrimination Rate* over a set of test cases, is:

$$\mathit{Discrimination\ Rate} = \frac{\#\mathit{Discriminations}}{\#\mathit{Flaws}}$$

To help understand the areas in which a given tool performs better or worse than others, CAS created the Weakness Classes. The Weakness Classes are defined using CWE entries that contain similar weaknesses. Table 3 shows these Weakness Classes with an example weakness that corresponds to each class.

Weakness Class	Example Weakness (CWE entry)
Authentication and Access Control	CWE 620: Unverified password change
Buffer handling (C/C++ only)	CWE 121: Stack-based buffer overflow
Code quality	CWE 561: Dead code
Control flow management	CWE 362: Race condition
Encryption and randomness	CWE 328: Reversible one-way hash
Error handling	CWE 252: Unchecked return value
File handling	CWE 23: Relative path traversal
Information leaks	CWE 534: Information leak through debug log files
Initialization and shutdown	CWE 415: Double free
Injection	CWE 89: SQL injection
Malicious logic	CWE 506: Embedded malicious code
Miscellaneous	CWE 480: Use of incorrect operator
Number handling	CWE 369: Divide by zero
Pointers and reference handling	CWE 476: Null pointer dereference

Table 3: The Weakness Classes of the CAS Static Analysis Tool Methodology, as presented in [35]

.

Using this methodology, CAS did a study in 2010 and published the results in [38]. All the results were published anonymously. In this study, they chose 9 tools, both commercial and open source, with 7 tools participating in each track (C/C++ or Java). The main results of this study were:

- Tools are not interchangeable
- Tools perform differently on different languages
- Complementary tools can be combined to achieve better results
- Each tool failed to report a significant portion of the flaws studied. On average, a tool covered 8 Weakness Classes and 22% of flaws in Weakness Classes

Not surprisingly, similar conclusions were reached in the SATE IV workshop that we presented in Section 3.2.1.3.

4 GENERAL DESCRIPTION

From our research, we didn't encounter any tool (commercial or otherwise) that would fit the purposes of extracting the system elements defined in the SIG Security Model. Hence, we decided to design and implement such a tool ourselves.

4.1 REQUIREMENTS

After discussing with members of the technical team and researchers at SIG, we set up some requirements that the tool should fulfill:

1. The purpose of this tool is to be used by SIG employees while performing Security Risk Asessments. Therefore, it is important that we use the current infrastructure of the company, so that our tool can be easily adopted.

2. We must also make sure that the tool will be easily configurable to fit the specifics of the current project at hand.

3. Furthermore, a very important feature is to keep the tool as language-agnostic as possible. The SIG has dealt with a huge variety of projects in many different technologies, so this will be a crucial feature. Of course, we will have to use some language-specific techniques at some points, but we must make sure to keep this to a minimum, and also ensure that in these cases our tool can easily be extended to support more technologies.

4. The tool will have as input the source code of the project, and as output information about the findings in a way that helps the evaluator locate them quickly and easily.

4.2 VALIDATION

Also, we discussed and decided how to validate the tool during development and also evaluate the tool's performance:

1. To be able to build and test a tool during development, we need to define a ground truth. This must be a system of average size, that has as many as possible (if not all) of the system elements defined in the security model. This system will be manually analyzed and it's system elements will be manually extracted. We will then proceed to build a tool that attempts to extract these elements automatically.

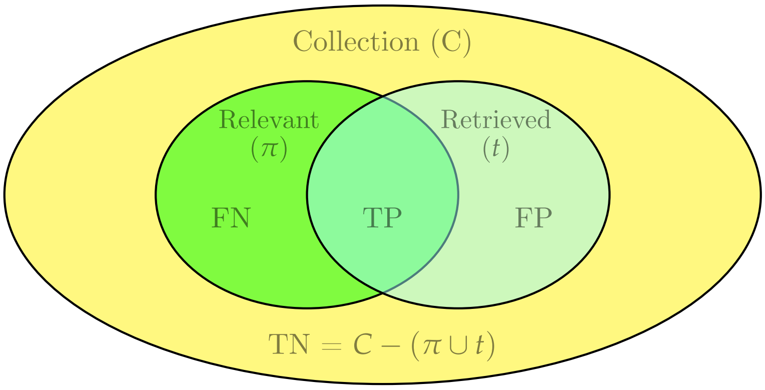


Figure 5: Metrics used to evaluate accuracy in the field of Information Retrieval. In our case, *Collection* is the set of all the possible elements, *Relevant* is the set of the system elements that we are interested in and *Retrieved* is the set of the system elements found by our tool. (FN = False Negatives, TP = True Positives, FP = False Positives, TN = True Negatives)

2. Upon reaching a sufficient coverage of the system elements for our ground truth, we must then evaluate its performance on different systems. This way we can draw solid conclusions about the performance and usefulness of our tool and we can properly evaluate it.

4.3 EVALUATING THE FINDINGS

To evaluate the findings of the tool, we must use metrics that are applicable to our case. For this, we looked to the area of Information Retrieval [36][37]. Let C be the collection of all possible system elements. From this collection, only a (sub)set π with $\pi \subseteq C$ is relevant to us, and our tool extracts a set t with $t \subseteq C$. Our goal of course is to increase the overlap of π and t, as much as possible. The grouping of findings to *Relevant* and *Retrieved* creates four possible combinations, that [36] summarises in Table 4 and shown in Figure 5.

	Relevant	Non Relevant
Retrieved	True Positives (TP)	False Positives (FP)
Not Retrieved	False Negatives (FN)	True Negatives (TN)

Table 4: Possible classification of the collection of system elements

With this classification, the two main metrics of Information Retrieval can be defined:

"Recall (R) is the fraction of relevant documents that are retrieved"

$$R = \frac{|TP|}{|\pi|} = \frac{|TP|}{|TP| + |FN|} \tag{1}$$

"Precision (P) is the fraction of retrieved documents that are relevant"

$$P = \frac{|TP|}{|t|} = \frac{|TP|}{|TP| + |FP|} \tag{2}$$

Using these two metrics, we can evaluate the overall usefulness of our tool.

4.4 THE STATIC ANALYSIS TOOLKIT

As mentioned earlier, SIG assesses the quality of the software products used by their client companies. The big diversity of technologies they encounter (over 70 programming languages) has created the need for them to build a tool to analyse these systems. They call this system the Software Analysis Toolkit, or SAT, and it provides a framework that allows building analysis tools for different languages efficiently. The SAT already supports more than 50 programming languages and it can generate, for example for the Java language, a graph of the system, broken down to method level. This graph is enriched with metrics for every element of the maintainability quality standard of the ISO 25010 that was described earlier. Metrics like Lines Of Code (LOC) and McCabe's complexity [39] are automatically calculated.

To address the first requirement (from Section 4.1) of our tool, we decided that we should base our tool on the SAT. By doing so, we also comply with the fourth requirement, since we will provide the root directory containing the modules of the system to be analysed, and then use the SAT to create the graph of the files and directories along with some other information, like the programming language. We can then develop some layers on top of that, where the system elements that are described in the SIG Security Model are extracted from the source code and reported .

4.5 THE MONITOR CONTROL CENTER

To comply with the validation choices that we made (Section 4.2), we must select an appropriate system as our ground truth. On the one hand, using the Ground Truth Method approach might introduce some bias in the process of developing the tool, and also this method is not automatic and hence time-consuming. On the other hand, it's one of the most accurate ways of interpreting results, since the system elements will be manually extracted by an expert in the field. In order to increase the accuracy of the results and also reduce the amount of time consumed during the manual element extraction, we decided that it would be best if we used a system that was built by SIG. Of course this system should also have the properties described in the requirements, i.e. it should be of average size and contain as many system elements as possible.

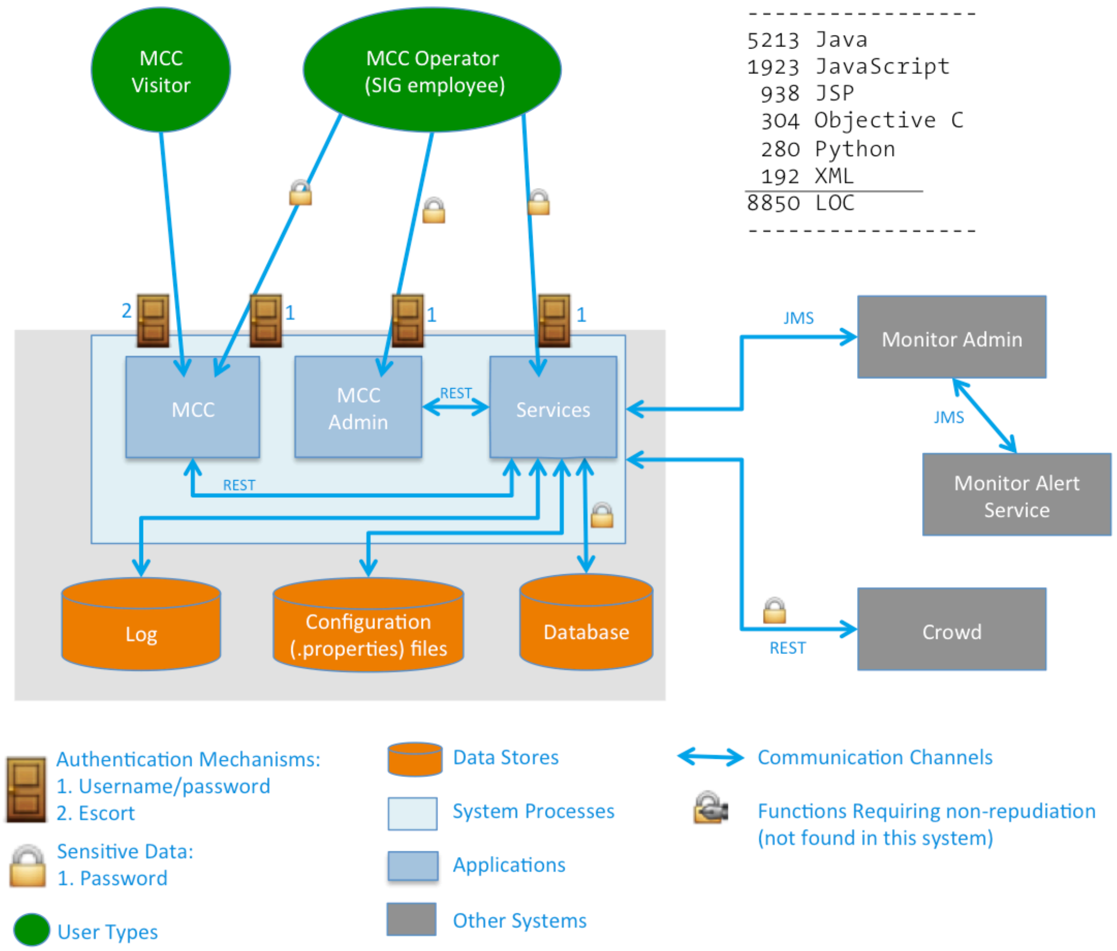
Upon consideration, we decided that an ideal system for our purposes would be the Monitor Control Center (MCC). One of the services that SIG provides to its customers is monitoring the development of their systems over a long period of time. The clients send snapshots of their code regularly (usually weekly) and the SIG continuously assesses the development and evolution of the product. The goal of the MCC is to improve the overall customer experience by decreasing the reaction time of SIG to a new development and also draw the attention to noteworthy events.

The MCC is developed using mainly Java, but also JavaScript, JSP, Objective C, Python and XML, and in total it is comprised of almost nine thousand lines of code. We decided that this should be sufficient to be used as our ground truth, as long as it contains a good variety of the system elements that we will be looking for. We then proceeded to analyse it and break it down to it's system elements, with the help of an experienced technical consultant. This breakdown is shown in Figure 6.

The MCC is meant to be used only by SIG employees, and this is the only user type that is found in the system. The user type "MCC Visitor" that is shown in Figure 6 is only there for the purpose of completeness. This is to show that occasionally the MCC is demonstrated to clients, but in these cases the clients are always escorted and the information presented on the MCC is anonymous. The MCC Operator must authenticate himself using a username and password to have access to the three applications of the single system process.

The core application of the MCC is "*Services*", and through this the system process communicates with the other systems. These are the "*Monitor Admin*" and the "*Monitor Alert Service*" and "*Crowd*", which is a third-party centralized identity management tool for web apps. Also, "*Services*" communicates with the datastores found in the system, namely the system Log, the configuration files (that have the extension `.properties`) and the system Database.

In Figure 6 we can see all the communication channels, and also where sensitive data are transmitted. For our system, the only sensitive data are passwords, and they are used in three places: when a user authenticates himself, when the database is accessed, and when the "*Crowd*" tool is used. In this system, there are no actions that require non-repudiation. Judging by the findings of the breakdown, we decided that the MCC



```
-----------------
5213 Java
1923 JavaScript
 938 JSP
 304 Objective C
 280 Python
 192 XML
8850 LOC
-----------------
```

Figure 6: Breakdown of the Monitor Control Center to its system elements

contains enough system elements to be considered our ground truth case.

5 TOOL DEVELOPMENT

Having established our ground truth, we then proceeded to develop our tool. We followed the pattern-based approach, i.e. we investigated which patterns or keywords are most commonly used for each of the system elements that we are looking for.

Of course, the biggest challenge that we had to overcome was to keep these patterns as language-agnostic as possible.

5.1 GENERAL TOOL DESIGN

As mentioned in the previous chapter, our tool was developed on top of the SAT. To use the SAT, the user must first specify mainly three things:

- The source directory, where the system's source code is located.
- The source context, i.e. the language of the files to be analysed.
- Which visitors will be used. As mentioned earlier, the SAT builds a graph of the system, and then each node is enriched with the results of different metrics. Therefore we will build graph visitors, and our objective will be to find the system elements.

To make the tool easily configurable, we decided that this information will be given through a configuration file. This way the user can tweak it to the specific needs of the project at hand easily, without having to compile the code over and over again. In this configuration file, the user is also asked to choose an output folder, for the results of our tool.

Also, apart from this information, in the configuration file we included the regular expressions that will be used to find the system elements. This way, the evaluator can tweak them and of course extend them or add new ones, making the tool more useful and accurate over time.

Our tool first builds the graph using the SAT, and then proceeds with enriching the nodes using the visitor that the user has chosen. When this is finished, we then finalize all the observations discovered by the visitor by collecting them from the nodes and writing them to a file, in a way that can be read and used efficiently by the evaluator. The output for each finding is of the form:

```
- <Found element> type : <Element classification according to the tool>
- <Found element> name : <Token that triggered the finding>
- Found in : <File that contains the token> , line: <line number>
```

5.2 DATA STORES

There are 4 main kinds of datastores that we focused our efforts upon:

- Reading/writing from/to files
- Accessing a SQL database
- Logging actions
- System configuration files

5.2.1 *Accessing files*

We started by examining the different ways in which languages handle I/O operations, and tried to discover a pattern that perhaps was common in some of them. Not surprisingly, there is a big variety of ways for performing I/O operations. Some examples are shown in Table 5. Some languages use *stream readers*, while others just use a simple *open* command.

Language	I/O operations examples

Java	`new FileInputStream("file.xmpl");`
	`InputStream inStr = getResourceAsStream("file.xmpl");`
	`ResourceBundle rb = ResourceBundle.getBundle("file.xmpl");`

C	`FILE *fp; fp = fopen("file.xmpl","r");`

C++	`ofstream myfile ("file.xmpl");`
	`ifstream myfile; myfile.open ("file.xmpl");`

Python	`f = open('file.xmpl', 'w')`

Haskell	`f <- readFile "file.xmpl"`
	`f <- openFile "file.xmpl"`

PHP	`$file = fopen("file.xmpl");`

Table 5: Examples of I/O operations in different languages

From Table 5, we can see that there isn't a pattern or keyword that is common and can be used to capture all the ways of performing I/O operations in a single language, let alone a pattern that would work for different languages. One thing that is common in most of the examples shown is that the filename is given as a string in brackets and double quotes, and also usually in these cases the filename is of the form `<filename>.<extension>`. Even though looking for this pattern would surely produce a lot of False Positives, we decided to include it in a "generic", language-agnostic visitor as a fallback. The regular expression that we finally used is shown in (3). An initial implementation for this, restricted the file extension to up to 4 characters in an attempt to limit the False Positives, but by validating it with our ground truth, we saw that we missed occurances of the file `config.properties`, which is crucial for the security assessment. Hence, we decided to lift that restriction.

`(w+.[a-zA-Z0-9]+)` (3)

Of course, this generic visitor is not sufficient, so we decided that on this occasion, it is necessary to build language-specific visitors. Since our ground truth system is built in Java (the other technologies we described in Chapter 4 are used for "gluing" things together), we decided to build a visitor that covers all the I/O operations in Java. From our research, all the possible ways of performing any kind of I/O operation in Java are the ones shown in Figure 7.

```
new FileInputStream
new FileOutputStream
new FileReader
new FileWriter
new File
getResource
getResourceAsStream
getBundle
getBundleAsStream
```

Figure 7: A complete list of the ways to perform I/O operations in Java

To cover all of these cases, we created the regular expression shown in (4). To deliver as precise and helpful information as possible, we manipulated the finding so that we report back only the filename, instead of reporting back the triggering keyword (for example, in the expression `new FileInputStream("file.xmpl");` the triggering keyword is `FileInputStream`, but the tool reports only the filename `"file.xmpl"`).

`(File.*|getResource.*|getBundle.*)` (4)

5.2.2 *Accessing a SQL database*

Our ground truth system uses the Hibernate ORM (Object/Relational Mapping) [40] to perform the SQL queries on the database. In fact, using an ORM is the most common way to persist data in many technologies. In Hibernate (as in most other ORM libraries/tools) the user provides the queries in normal SQL syntax, in strings. Therefore, we created the regular expression shown in (5), that looks for SQL keywords that have to do with the database tables, i.e. SELECT, INSERT, UPDATE, ALTER TABLE, DROP TABLE. Again, we decided to manipulate the findings to report back directly the table name, instead of the triggering keyword.

$$\texttt{(.*)(SELECT|INSERT|UPDATE|ALTER TABLE|DROP TABLE)(.*)} \tag{5}$$

5.2.3 *Logs*

For logging purposes, the MCC uses the Apache Log4j [41] tool. However, in every technology, every logging tool uses the keyword `log` when it's used. Our main task then, was to filter out other tokens that might match this keyword. We ended up with the regular expression shown in (6). Apart from that, while reviewing the results we noticed that many hits were caused by the statement for importing the Log4j package (for example, `import org.apache.log4j.Logger;`). We then decided to ignore the findings that contained `org.apache`, so that we reduce the "noise" of results, and lead the evaluator only to the places where logging takes place.

$$\texttt{(.*(?!login|logo|logic|logged)log.*)} \tag{6}$$

5.2.4 *Configuration Files*

System configuration files are very common in Java. There are different sources for the user to pass the configuration to the system. Some of the most common ones are shown in Table 6. Our task for the configuration files is only to list them, so that the evaluator knows where they are located, which is done relatively easy. Our ground truth system uses the `.properties` configuration files, so our visitor went through the filesystem and looked for all the `.properties` files, and then reported them back as configuration files, together with their name and location. Since no further processing of the findings is required in this case, this visitor can be easily extended to look for other configuration sources as well.

Configuration source	File extension
Properties Files	`.properties`
XML documents	`.xml`
Windows INI files	`.ini`
Property list files	`.plist`

Table 6: Some of the most common ways to define the system configuration in Java

5.2.5 *Results*

With these regular expressions in place, we then run our tool on the MCC and classified the findings as True Positives or False Positives. The classification per keyword is shown in Figure 8.

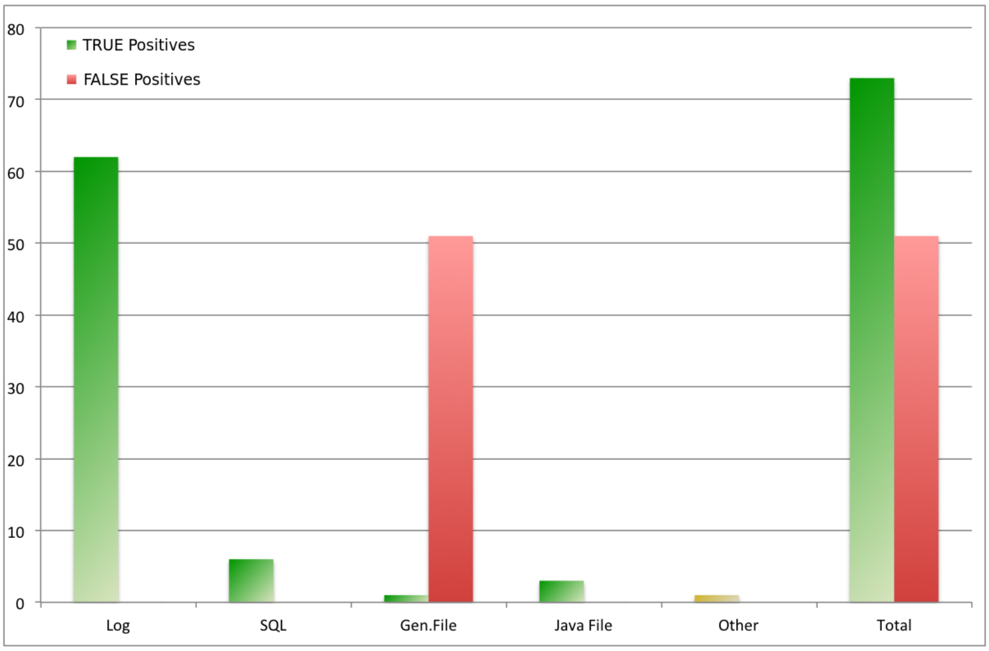


Figure 8: Classification of Datastore findings on the MCC as True Positives or False Positives, per RegEx: Log (5.2.3), SQL (5.2.2), Generic File and Java File (5.2.1), Other and Total)

From this figure we can see that our filtering worked well, as we only have False Positives for the generic file regex. For this RegEx, in the MCC, our tool had 1 True finding and 51 False findings. As we expected, this RegEx is not accurate at all, but on the other hand, the 1 True finding was not found by the Java file RegEx, and since for a complete security assessment *every* file could potentially threaten the whole system, we decided not to discard this RegEx. Also, in the figure there is a column labeled "Other". Our tool found one file called `default_photo.png` that was loaded. Strictly speaking, this is indeed a file being loaded by the system, but it is not in the scope of the elements that we are looking for, therefore we decided to classify it as "Other findings".

Using this report, we then proceeded with calculating the metrics defined in Chapter 4, namely *Precision* and *Recall*. In total, not counting the out-of-scope finding, our tool returned 51 False Positives and 83 True Positives, which means the *Precision* is 66.9%.

Given the fact that all the False Positives were caused by the one RegEx that we were expecting to produce a lot of False Positives, we were satisfied by this precision.

For *Recall*, we can't use the strict definition that was given previously in Chapter 4. The reason for this is that our tool reports back the lines of the findings. To strictly measure *Recall* using that definition, we would have to manually go through every line of code and classify it as relevant or not, and then compare this list with the tool report. Instead, we decided to measure *Recall* at a more "abstract" level, so we only check how many of the datastores shown in Figure 6 (i.e. the orange boxes) are found by our tool. Since our tool found all of the datastores, using this "lighter" definition of *Recall* our tool scored 100%. However, this does not guarantee that our tool has found *all* of the places where the use of these datastores takes place.

The results of the configuration files visitor are not shown in Figure 8. The reason for this is that for configuration files our visitor simply lists the `.properties` files and therefore all the results from this visitor are true positives. Furthermore, all the `.properties` files of the system are listed, so *Recall* is 100%. For the ground truth system, this visitor listed 36 configuration files.

5.3 AUTHENTICATION MECHANISMS

The next category of system elements that we worked on, was Authentication Mechanisms. Even though there are many different tools and frameworks that can be used in various technologies, the keywords that are used are not different from framework to framework and from technology to technology. This allows us to build a generic, language-agnostic visitor. We started looking in our ground truth for the keywords `username`, `password` and `auth` to get an initial impression of how authentication is done in our system. Upon investigation and research, the final regular expression that we ended up with is shown in (7). Our tool classified the findings as "Authentication", and listed it with the activating keyword and the file and line number that it was discovered.

```
(.*(?!author)(login|logged|auth|user|pass|credential).*)          (7)
```

We then proceeded to run our tool to collect all the findings for authentication mechanisms, and classified them in True Positives or False Positives. This classification is shown in Figure 9. Again, we separated the finding per keyword, so that we can evaluate them individually.

From Figure 9 we can see that unlike what we had with the Datastores, almost all keywords produce False Positives. Unfortunately it was not possible to filter them out, without missing out on other True Positive findings. Furthermore, in the figure we also

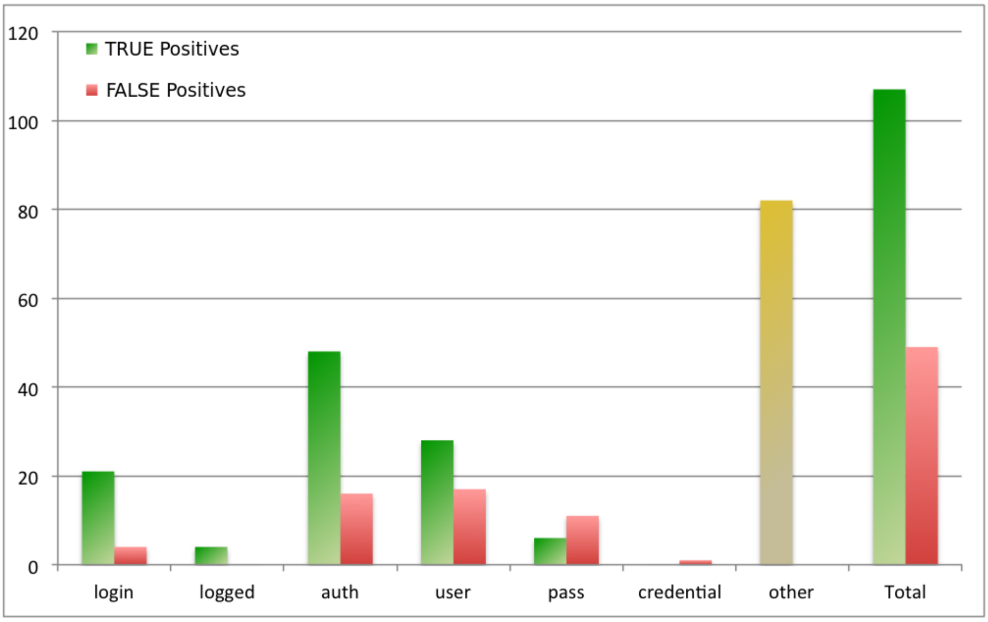


Figure 9: Classification of Authentication Mechanism findings on the MCC as True Positive or False Positive, per RegEx (5.3) (login, logged, auth, user, pass, credential, Other and Total)

inlcuded again a column labeled "Other". These findings were indeed true authentication findings, but they referred to the authentication between the *Services* application and the *Crowd* tool. After discussing these findings with a technical consultant, we decided that these findings were out of scope, since it's an internal authentication mechanism and not very interesting while performing the security assessment of the system.

In total, again ignoring the findings that we classified as out of scope, our tool returned 49 False Positives and 107 True Positives, giving us a *Precision* of 68.6%. For *Recall* we followed the same practice like before, and used the "lighter" version that we described for the Datastore findings. There are 3 places of authentication in the MCC, where the operator uses each of the 3 applications. Our tool reported findings in all 3 of these places, so the *Recall* is again 100%.

5.4 SYSTEM PROCESSES, COMMUNICATIONS CHANNELS AND SENSITIVE DATA

From our results so far, it is clear that the MCC has 3 applications in 1 system process. It is also clear that some users authenticate themselves to have access to these applications, and we discovered the *Crowd* tool from the authentication findings. The definition of the "Communications Channels" according to the SIG Security Model was *"for all communication between system processes and any of the users, other systems and data stores there is a communication channel"*. Therefore, we can use the results we already have to connect the elements we discovered so far.

The immediate communication channels that are discovered, are the ones connecting *"Services"* to the 3 datastores, and also to *"Crowd"*. Also, the potential users - that could belong in one or more groups - can connect to the 3 applications of the system process. The connections between the *"MCC"* and *"MCC Admin"* applications and the *"Services"* application are also found. Furthermore, using the findings from the authentication mechanism visitor, we can see on which channels there is transmission of sensitive data (in our case, passwords). Combining all these information, we can draw a picture of the discovered elements by our tool so far. This picture is shown in Figure 10.

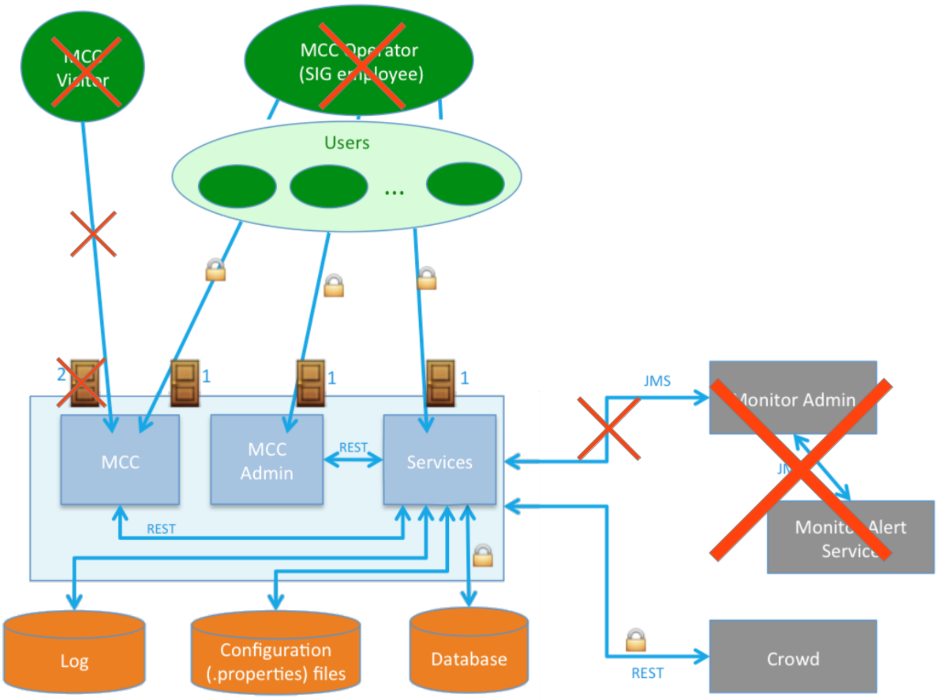


Figure 10: The discovered system elements on our ground truth system by our tool, by using the reports on datastores and authentication mechanisms. Marked with red X are the elements that were shown in Figure 6, but not discovered by our tool

Comparing this picture with the one that we manually created while investigating our ground truth, we can see that our findings are able to recreate an almost complete picture.

5.5 USER TYPES, NON-REPUDIATION AND OTHER SYSTEMS

Unfortunately the MCC was designed to be used by only 1 user type, namely the MCC operator. Therefore, there is no evidence in the source code for us to investigate in order to encapsulate them in the form of a visitor for our tool. Moreover, there are also no functions in this system that require non-repudiation. Thus, the only area that we could look into to improve our tool would be the "Other Systems".

However, finding other 3rd party systems cannot be easily done from inspecting the source code, most of the relevant code will be in the 3rd party system, and not in the one we have at hand. Basically, the only way to actually find these 3rd party systems, is to find the communication channels that connect them to one of the system processes. Usually the evidence that we can use to trace the 3rd party systems are the authentication mechanisms, like the case of the *"Crowd"* system that we found in the MCC. Nonetheless, this is not limiting us in creating our framework for this tool. Considering the time limitations we had for this project, we expected it was possible that we wouldn't be able to explore all the system elements.

Since we already had an almost complete picture of the MCC from our findings so far, and given the hurdles described above, we decided to proceed with the next phase of our project, namely the large-scale testing and evaluation, and reserve ourselves to improve upon this area in the future if time allows it.

6 TESTING OUR TOOL

To properly evaluate our tool, we decided to ask from technical consultants to use it, validate the findings, and give an overall opinion. This way we can have a clear picture of the general applicability of our tool and its results. We defined a testing plan so that our experiments would be as precise as possible, and presented the technical consultants with the steps that they should follow to use the tool and what was expected of them in the end.

6.1 TESTING PLAN

The purpose of these experiments is to evaluate the effect of using the Elements Extraction tool. For these experiments, we ask a Technical Consultant to use the tool under the following plan:

- The technical consultant must have experience in applying the SIG Security Model.
- The test case to be used will be a real company project for which a security assessment has been made, using the SIG Security Model. Also, the technical consultant performing this experiment, was involved in that assessment. This has both positive and negative side-effects, that are thoroughly discussed in Chapter 7.
- The time invested in this experiment will be 1 hour. In this hour we will also mark the progress and findings of the first 30 minutes, and compare these findings to the total findings of the whole hour.
- The evaluator will follow the steps provided to them for using the tool.
- In the experiment report, we will classify the tool as Useful, Potentially Useful, or Not Useful. In any case, a full justification will be given.

The steps that were provided to the evaluators are:

1. Edit the configuration file to match the search criteria. This configuration file is found in `elementsutils` and the fields to be edited are:
   a) The source directory
   b) The output path
   c) The source context

   d) Which visitor will be used

2. For this experiment, we need to run the tool in 2 different source contexts:
   a) `java`
   b) `properties`

3. For the `java` source context, we want to run 2 visitors:
   a) `datastorejava`
   b) `authmechgeneric`

4. For the `properties` source context, we want to run 1 visitor:
   a) `datastoreproperties`

5. After the tool is run with all the above configurations, the tool findings will be in the output folder in 3 files:
   a) `allDataStores.txt`
   b) `allPropertiesFiles.txt`
   c) `allAuthMechs.txt`

   The evaluator must then go through these files and classify each finding as True Positive or False Positive.

6. Using these findings the evaluator should be able to draw a picture similar to the MCC breakdown shown in Figure 6 (this picture was shown to the evaluators during the experiment). Furthermore, the evaluator can also draw the communication channels between the sysem elements that can be derived by the tool findings, and also pinpoint where passwords (Sensitive Data) are going through a communication channel.

6.2 TOOL EVALUATION

6.2.1 *Project A*

The first project that was used for our experiments was a software system for a telecommunications company of medium size, i.e. 25 thousand Lines of Code, with 22.6 thousand lines of java code. The evaluator needed about 6 minutes to configure the tool and run the requested visitors in the requested modes. The findings were:

1. Datastores: Zero findings.

2. Configuration (`.properties`) files: Zero findings.

3. Authentication Mechanisms: 946 findings in 91 (out of the total 154) files.

The evaluator then proceeded to classify the files as True Positives or False Positives.

General comments for the experiment:

1. The whole experiment (configuration + evaluation) was finished in 35 minutes. The evaluator explained that he would not go through the findings one-by-one to classify them. Instead he would classify the files that were reported back from our tool, and classify those, since this is how the tool would be used in a real evaluation. Out of the 91 files, 17 were classified as False Positives and 74 were classified as True Positives, which gives us a *Precision* of 81.3%.

2. The lack of findings in datastores and configuration files was not correct. In fact, it is a False Negative, since the system does have a datastore. The project that was used is an android application and therefore uses the android-specific framework for storing files. After we described to the evaluator how the tool searches for datastores, he was not surprised that the tool missed this.

3. For a comment on *Recall*, the evaluator also followed the somewhat lighter version that we described in the tool development chapter. Since he was involved in the assessment of this project, the evaluator was in a position to tell us which elements were missed by the tool. Of course, the major one was the datastore that was mentioned above. Besides this, the evaluator said that our tool missed a 3rd party tracking service, but it found the connection to the backend of the client.

4. The evaluator liked the fact that the tool reports the line of every finding, because this helped him classify the findings as True Positive or False Positive fairly quickly. Nevertheless, he explained that grouping the resulting files by certain triggers would be more helpful, since this would assist him to build a good hierarchy of the system.

5. When asked to draw a picture of the system based on the findings, the evaluator said that since he was involved during the original assessment of this system by the company, he already had a pretty complete and accurate picture of it in his mind. He suggested that it woudl not be fair to try and draw this picture based on the tool findings because of this.

6. Before expressing an overall opinion for the tool, the evaluator asked for a feature that makes this tool better than a series of `grep` queries. We explained that the tool's functionality could be easily extended by adding or editing regular expressions in the configuration file, so with every use on different systems, new triggers can be added for a more complete search. This accumulation of triggers over time, will make the tool more and more accurate.
   With this in mind, and even though the tool had a major false negative for the

datastore that was used and the 3rd party tracking service, the evaluator was overall positive on the tool. When asked for a general opinion on the tool, the evaluator said that it is Potentially Useful.

6.2.2 *Project B*

The second project was a software system for the Dutch government and it was much larger, i.e. in total 116 thousand lines of code, with 85 thousand lines of java code in 1478 java files. The evaluator needed about 8 minutes to configure and run the tool as described by our experiment instructions. The findings were:

1. Authentication Mechanisms: 760 findings in 89 files
2. Datastores: 2848 findings in 404 files
3. Configuration files: 104 files

The evaluator then proceeded to classify the files as True Positives or False Positives. At the 30 minutes mark, the evaluator was halfway through the authentication mechanism findings.

General comments for the experiment:

1. Out of the 89 files reported by the tool for containing authentication mechanisms, the evaluator reviewed 63 of them (70.8%) due to the time limitations. From these 63, 29 were classified as False Positives and 34 were classified as True Positives, so the *Precision* was 54.0%.

2. The evaluator then proceeded in reviewing the findings for datastores. The tool reported findings in 404 files. The evaluator quickly found that the reason for this was that almost every file was using the system log. In order to get a picture of datastores other than logs, the evaluator disabled the log regex from the configuration file of our tool and run the datastore visitor again. With this new configuration, our tool reported findings in 82 files. Out of these 82 files, 2 were classified as False Positives, and the other 80 as True Positives, giving us a *Precision* of 97.5%.

3. When asked to make a comment on *Recall* by comparing the findings of our tool to the findings of the original evaluation of the system that he performed, the evaluator said that the findings were on a different level. The findings reported by our tool were in a lower level, and needed to be further processed by the evaluator to get the higher-level findings. Of course our tool reports lines of findings, which can then be composed upon evaluation to get the higher-level picture of the system elements, but this requires some further processing by the evaluator.

4. The evaluator said that he would prefer if the results were clickable, i.e. from the tool report to click on the findings and the corresponding file would be opened on the specific line of the finding. This functionality was supported by tools like OpenGrok[1]. When asked why we didn't use a tool like this, we explained that during designing the tool we decided that we should use the company's existing framework for the tool to be more easily adapted in the future.

5. For an overall opinion, the evaluator said that he would prefer if he can run multiple visitors with one configuration, and also make the search more adaptive. For example, to disable the log regex as mentioned above, the evaluator had to replace the regex with a nonsense string. It would be better if in the configuration file there was an option to disable the regex in a better way. However, he said that the tool gave him a different angle to see the source code. In assessing such a big system in a very limited time, the evaluator mentioned that he follows a "design-based" review, not an exact file-to-file source code review, because in that timeframe it would be impossible. Our tool lead him to files that were overlooked during the assessment and that were interesting. As a general opinion, this evaluator also said that the tool is Potentially Useful.

6. When asked to draw a picture of the system based on the findings, his response was similar to the other evaluator. However, he said that one of the potential uses of our tool would be to validate the picture that he would get during the technical interviews with the developer team of the system.

[1] http://opengrok.github.io/OpenGrok/

7 DISCUSSION

7.1 ANSWERS TO RESEARCH QUESTIONS

As stated in Section 1.3, *the objective of this study is to investigate possible improvements in the process of evaluating the software product quality in regards to security, using the SIG Security Model and integrating tool support in the steps of the evaluation process where it's useful.*

We believe that this objective has been met through the different phases of this thesis. We broke our main research question into 5 subquestions, the answers to which are summarised in Table 7.

Having answered these 5 subquestions, we believe that our main research question is also answered. The main research question was: *"What tools can be used for enhancing or automating the evaluation process for software quality regarding security?"* and these tools have been described, presented and discussed in this thesis.

7.2 RESULTS AND VALIDITY

To evaluate our tool we performed 2 case studies. We chose these evaluations to be performed by experienced technical consultants, on projects that they had performed a security risk assessment in the past. With this choice we were able to evaluate also the False Negatives of our tool, since the technical consultants could compare the findings of the tool to the findings of their previous evaluation.

However, we recognize the following limitations to the validity of our conclusions:

- The fact that we decided to have the technical consultants use the tool on projects that they were involved in the past, led them to their understandable unwillingness to draw a picture of the system that we required from them according to step 6 of the steps described in Section 6.1, since they already knew this picture from their previous assessment.
- During the tool development, *Recall* was impossible to be measured in the detail required by its definition. This led us to use a lighter, more abstract definition for it, that is described in Chapter 5. This was also an issue when we performed the 2

Research subquestion	Answer
Which parts/steps of the evaluation process can be supported by tools?	This subquestion is answered by our literature study, specifically the breakdown of the SIG Security Model, in section 2.2
Which tools are currently available that can be used?	The research we performed lead us to a number of tools that are presented in Chapter 3. These tools are source code scanners that can be used to detect vulnerabilities regarding the software security.
Which tool(s) are more suitable for the purposes of SIG?	To answer this subquestion we first went through the rules list of all the tools we found (or at least the ones that shared this information when we contacted the vendors). This mapping is shown in Table 2. We then tried to obtain trial licenses for the tools we had found to test and evaluate them, but unfortunately this was not possible, so we presented some evaluations made by NIST in Section 3.2.
If no tool is available for a certain step of the process, can we specify, design and implement one?	For the step of extracting system elements, we couldn't find a tool that suited the needs and the purposes of SIG. In Chapters 4 and 5 of this thesis we describe the requirements of such a tool, the design, and finally the implementation of it.
How can we measure and evaluate the impact of tool support in the evaluation process?	Since we couldn't obtain licenses for the scanning tools we found, we could only measure the impact of the tool that we built. Our evaluation process is described in Chapter 6, and it included 2 large-scale studies, and analyses performed by 2 experienced technical consultants.

Table 7: Answers to research subquestions

case studies. However, based on the opinion of the technical consultants and their comparison from their own previous evaluation of the system, we believe that we mitigated this limitation to the validity of our conclusions.

- The effectiveness of our tool is based on the regular expressions we created. With these regular expressions, we tried to cover all the sensible keywords that can be used for performing the tasks we looked at. However, we have no guarantee that these keywords are used by all developers. For example, developers that are more comfortable using their own native language instead of English can create different keywords (e.g. the word *password* in Dutch is *wachtwoord*).

From our evaluations, we got positive feedback from the technical consultants. Also, the False Negatives were analysed, and some solutions for them were explained. For example, in *"Project A"* in Section 6.2.2 we had one major False Negative for the datastore. That was due to the fact that this system was using a specific framework to connect to its datastore, and it's a prime example of why a tool like the one we developed cannot be 100% language-agnostic. This framework can be analysed and then encapsulated in some triggering keywords that can be added to the ones that we already included in our tool.

The second False Negative was a 3rd-party tracking service. According to the technical consultant that performed the evaluation, this service basically tracks the visits to a particular website. There isn't any authentication performed when this service is activated, and it's not exactly a datastore, but it keeps a record of all the visitors, so some data is persisted. We believe that it's a reasonable miss by our tool, but one that must be investigated so that it is not missed by future versions of our tool.

Overall, the *Precision* of our tool was in the region of 68% on our ground truth, 81% for Project A and 78.6% for Project B (54.0% for the authentication mechanisms, and 97.5% for the datastores). The precision for *Project A* and for the datastores of *Project B* was satisfactory, but the precision for the authentication mechanisms of *Project B*, at a first look seems low. However, as Figures 11, 12, 13 and 14 illustrate, the reduction rate of our tool is big enough to make up for this low precision. For *Project A*, our tool reported back 946 findings. Even if we consider that there is only 1 finding in a Line Of Code (LOC) (which in reality it is a big overestimation), out of the 22.6 thousand lines of java code our tool reported only a small fraction, as shown in Figure 11. Our tool findings were spread over 91 files, which is about 60% of the total java files of the system (see Figure 12). For *Project B*, our tool reported 3608 findings which if we again consider only 1 finding for every LOC, out of the 85 thousand lines of java code, this is only 4.1% of the system as depicted in Figure 13. In total, there were findings in 171 files, only 11.6% of the java files of the system (see Figure 14). With this reduction rate, the evaluator has only a small fraction of the system to investigate. Considering this factor, even the precision of 54% that we had for the authentication mechanisms is acceptable, if the *Recall* of the tool is good.

For the *Recall* we cannot give a precise number, because of the limitations that we described above. For the system that was chosen as the ground truth, all the elements that were searched by our tool were discovered, but again we are only judging this from a higher, more abstract level. For *Project A*, our tool discovered the authentication to the backend of the system, but missed the datastore of the system and the 3rd-party tracking service. For *Project B*, the evaluator was reluctant to make a direct comparison of the findings of our tool to the high-level elements of the system we analysed, for the reasons we explained in Section 6.2.2.

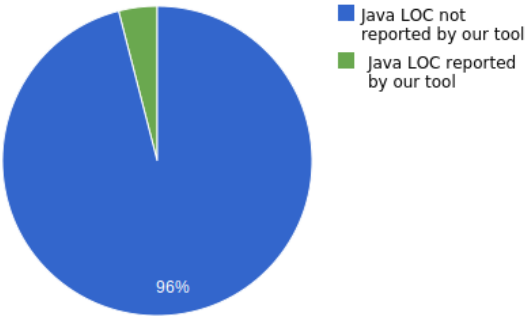


Figure 11: Reduction rate of our tool on Java LOC, for Project A

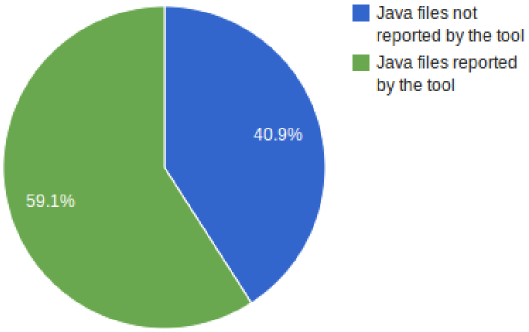


Figure 12: Reduction rate of our tool on Java files, for Project A

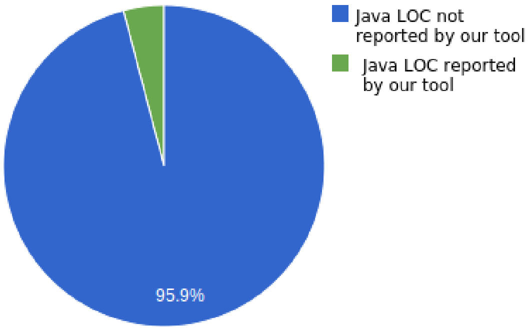


Figure 13: Reduction rate of our tool on Java LOC, for Project B

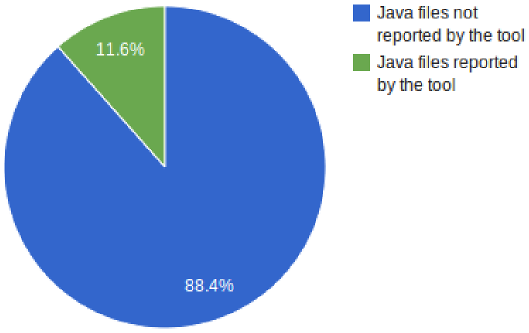


Figure 14: Reduction rate of our tool on Java files, for Project B

Looking back at the requirements that we set for our tool in Section 4.1, we see that all are met, to the greatest possible degree. By choosing to use the SAT, we meet our first requirement. All of the technical consultants at SIG are using the SAT every day, so understanding exactly how our tool works will be no problem for them, thus they can adopt it easily.

The other requirements we set are also met. Using the configuration file, the user of our tool can choose the visitors he wishes to use, and also which language (source context) he wants to analyse. The regular expressions that are used by our tool are also found in the configuration file, so the user can manipulate them, or add more to fit the project at hand.

Regarding the requirement for our tool to be language-agnostic, we kept it as much as it was possible. We explained in Section 5.2.1 and showed in Table 5 that to ensure a good performance of our tool, we must use language-specific triggers in our regular expressions to find the locations for File I/O. This need was also emphasised by the False Negative of our tool regarding the datastore of *Project A* during the testing phase. We created a java-specific regular expression (shown in (4)) in order to improve our tool's performance, but of course our tool should be enriched with other language-specific regular expressions to cover comprehensively other languages. However, we also included a fallback, generic regular expression (shown in (3)) and also all the other regular expressions are language-agnostic since they use only generic triggers, so we consider that this requirement was also met.

7.3 RELATED WORK

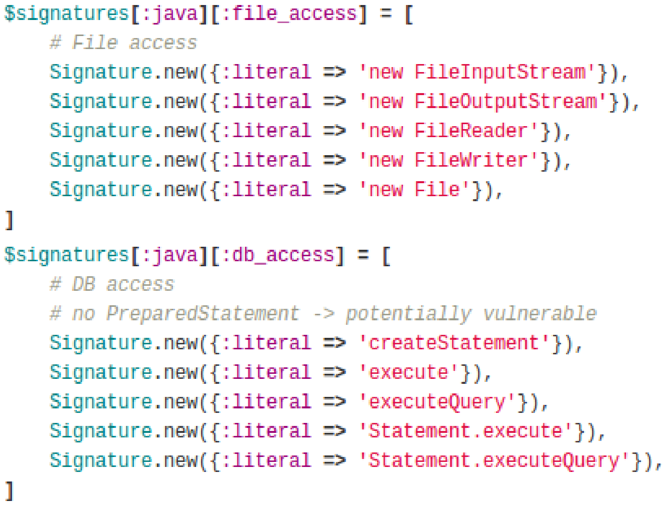
During the last parts of our implementation, we discovered a tool called Watchtower [42]. This tool is *"a Static Code Analysis tool designed to assist security auditors who are tasked with performing manual code reviews. It offers a robust alternative to grep for finding matches on literal and regex-based strings within a project. It can thus be used to locate (for example) dangerous functions calls that are made within an application"*.

Even though this tool isn't designed to do the same tasks that our tool does, we decided to run this tool on our ground truth, and compare the findings with our tool, since there should be some overlap - at least in the case of datastores or files. Our motivation was to perform a comparison of our tool with a similar one, and perhaps get inspiration and see places where our tool could be improved. Watchtower reported back 46 findings, out of which 17 were located in the testing code. From the other 26, 8 were irrelevant to our tool since they didn't involve any of the system elements that we are looking for, 14 were False Positives and 7 were True Positives. All 7 of the True Positives were found because of the `execute` keyword, and they were regarding database access.

To explain the findings, we looked into the regular expressions that are built-in the Watchtower tool. The ones that are relevant to our tool are shown in Figure 15.

From this picture we can see that regarding file access, compared to our tool, Watchtower doesn't cover all the cases that we cover. Some examples are `getResource` and `getBundle`, that are common practice in java. Furthermore, for the database access, it is not looking for any SQL keywords, which we believe would be very useful. The 7 True Positives that were reported back from Watchtower were also found by our tool.

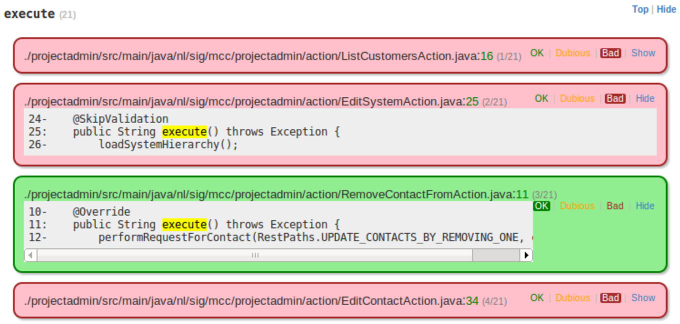
The main advantage of Watchtower over our tool would be its report. To present the findings, watchtower creates an HTML file in HTML5, that shows each finding with a code snippet so that the user can get a quick picture of the context. Also, the findings are grouped by the keywords that triggered them and each finding is presented in a box



```
$signatures[:java][:file_access] = [
    # File access
    Signature.new({:literal => 'new FileInputStream'}),
    Signature.new({:literal => 'new FileOutputStream'}),
    Signature.new({:literal => 'new FileReader'}),
    Signature.new({:literal => 'new FileWriter'}),
    Signature.new({:literal => 'new File'}),
]
$signatures[:java][:db_access] = [
    # DB access
    # no PreparedStatement -> potentially vulnerable
    Signature.new({:literal => 'createStatement'}),
    Signature.new({:literal => 'execute'}),
    Signature.new({:literal => 'executeQuery'}),
    Signature.new({:literal => 'Statement.execute'}),
    Signature.new({:literal => 'Statement.executeQuery'}),
]
```

Figure 15: The regular expressions that are built in Watchtower and are relevant to our tool

that has the options to mark it as "OK", "Dubious", "Bad", or Hide it. A small sample of the tool output is shown in Figure 16.



execute (21)

Top | Hide

./projectadmin/src/main/java/nl/sig/mcc/projectadmin/action/ListCustomersAction.java:16 (1/21) OK | Dubious | Bad | Show

./projectadmin/src/main/java/nl/sig/mcc/projectadmin/action/EditSystemAction.java:25 (2/21) OK | Dubious | Bad | Hide

```
    @SkipValidation
    public String execute() throws Exception {
        loadSystemHierarchy();
```

./projectadmin/src/main/java/nl/sig/mcc/projectadmin/action/RemoveContactFromAction.java:11 (3/21) OK | Dubious | Bad | Hide

```
    @Override
    public String execute() throws Exception {
        performRequestForContact(RestPaths.UPDATE_CONTACTS_BY_REMOVING_ONE,
```

./projectadmin/src/main/java/nl/sig/mcc/projectadmin/action/EditContactAction.java:34 (4/21) OK | Dubious | Bad | Show

Figure 16: A sample of the report produced by Watchtower

Even though Watchtower is superior in presenting the findings than our tool, by using the SAT we can use functionalities that Watchtower doesn't have. For example the SAT can distinguish between production code, test code and even generated code, so our tool is only running on the production code which is the objective.

8 CONCLUSION AND FUTURE WORK

In this thesis we studied and analysed a model for performing security assessments on software products, that was based on the ISO 25010 standard. Our task was to identify the places where automation and tool support would enhance or improve the assessments using this model.

8.1 CONTRIBUTIONS

With our work, we pinpointed all the places in performing a security assessment of software products using the SIG Security Model that can be enhanced or supported by tools. We then proceeded in further investigating two of these places, and we collected a lot of information for one and designed and built a tool for the other.

The information that was collected during the first phase of our research can be used as a point of reference for future work in this field. We listed a lot of code scanners, with some of them considered to be the state of the art, and we present them together with reports and further information of their features, and also organisations and conferences that have attempted to compare and validate them.

In the second phase of our work, we identified the need for a tool for extracting system elements and specified the requirements that such a tool should have. We also defined an evaluation protocol for this tool by presenting a ground truth method to validate our findings. Then we proceeded with designing, implementing and testing our tool and finally performed 2 large-scale studies to test and evaluate the generality and applicability of our tool in systems other than our ground truth. The results we had were satisfying and promising, and both of the technical consultants that performed the evaluations expressed positive comments about our tool.

We consider that our initial objective was reached, and that this thesis has an actual contribution to both the academic and to the business environment, and also it created the foundations for future research on this topic.

8.2 FUTURE WORK

Certainly, the work of this thesis can be continued and improved. The first improvement that comes to mind would be to look further into the system elements that were not covered by our work, namely the User Types, the Functions Requiring Non-Repudiation and the Other Systems. For this to be possible, a new system that has these elements must be used as a ground truth method to search and validate the results.

Of course, other system elements like Sensitive Data were also not directly covered by our tool. There needs to be some research in this area in order for a specific visitor for these data to be built. This visitor should not only be constrained in looking for passwords, but other forms of sensitive data like pin numbers or bank account and credit card numbers. Additionally, to mitigate one of the limitations we described in Chapter 7, we suggest further research to be done in the area of finding the keywords used in the regular expressions in different languages.

Furthermore, the reporting of our tool needs to be improved. One thing that we noticed during the large-scale case studies is that both of the evaluators didn't pay much attention to the first 2 fields of each finding, namely the classification of the finding (e.g. log, file, database, etc) and the keyword that triggered the finding or the file name. Instead they only looked at the third field that had the information on where to find this result. Perhaps an improvement on the reporting would be to skip the first 2 fields altogether, and report only the filepath and the line number of each finding in comma-separated values (CSV) format, since this would allow quick and easy importing in a spreadsheet that would be convenient to the technical consultants. Also, the filepath could be presented as a kind of hyper-link that opens the corresponding file when clicked.

The first part of our work could also be improved. We didn't get the chance to have a hands-on experience with the code-scanning tools that we found so we had to rely on the reports of 3rd party studies, like the ones by NIST. Ideally, the obstacles that we encountered can be overcome and then we can compare these tools first-hand and give our own reports.

Finally, further research should be done to investigate potential tool support on step 2 of applying the security model that was presented in Section 2.2, namely the system exploration step. This tool will probably be a data flow analysis tool, but it should also be able to detect cross-language flow. In any case proper research must be done to specify the exact needs and requirements so that a tool can be found or designed.

BIBLIOGRAPHY

[1] ISO.org, *ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models*, 2011.

[2] ``OWASP top 10 - 2013. The ten most critical web application security risks,'' tech. rep., OWASP The Open Web Application Security Project, 2010.

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

[4] M. Howard and S. Lipner, *The Security Development Lifecycle.* Redmond, WA, USA: Microsoft Press, 2006.

[5] G. McGraw and J. Routh, ``Software [in]security: Measuring software security,'' *informIT*, 2009. http://www.informit.com/articles/article.aspx?p=1357183.

[6] ISO/IEC, *ISO/IEC 9126. Software engineering -- Product quality.* ISO/IEC, 2001.

[7] ``Understanding root cause analysis,'' tech. rep., BRC Global Standards, 2012.

[8] H. Xu, J. Heijmans, and J. Visser, ``A practical model for rating software security,'' in *7th International Conference on Software Security and Reliability (SERE 2013)*, 2013.

[9] ISO.org, *ISO/IEC 25020:2007 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Measurement reference model and guide*, 2007.

[10] ISO.org, *ISO/IEC 25021:2007 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Quality measure elements*, 2007.

[11] M. Gegick and S. Barnum, *Securing the Weakest Link.* Cigital, 2005.

[12] P. Bullen, *Handbook of Means and Their Inequalities.* Mathematics and Its Applications, Springer, 2003.

[13] T. Polychniatis, J. Hage, S. Jansen, E. Bouwers, and J. Visser, ``Detecting cross-language dependencies generically,'' *2011 15th European Conference on Software Maintenance and Reengineering.*

[14] AppPerfect, *AppPerfect Java Code Test.* http://www.appperfect.com/products/java-code-test.html.

[15] University of Maryland, *FindBugs.* http://findbugs.sourceforge.net/.

[16] Google, *CodePro Analytix.* https://developers.google.com/java-dev-tools/codepro/.

[17] Hewlett Packard, *HP Fortify Static Code Analyzer.* http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812.

[18] Veracode, *Veracode Static Analysis.* http://www.veracode.com/products/static.

[19] Microsoft, *Code Analysis for Managed Code.* http://msdn.microsoft.com/en-us/library/3z0aeatx.aspx.

[20] Bugaroo, *BugScout.* https://www.buguroo.com/en/products/bugscout/.

[21] Checkmarx, *Checkmarx.* http://www.checkmarx.com/.

[22] Armorize, *CodeSecure.* http://www.armorize.com/codesecure/.

[23] Coverity, *Coverity Static Analysis Verification Engine (Coverity SAVE).* http://www.coverity.com/products/coverity-save.html.

[24] IBM, *IBM Security AppScan Source.* http://www-03.ibm.com/software/products/us/en/appscan-source.

[25] Klocwork, *Insight Defects.* http://www.klocwork.com/solutions/insight-defects/.

[26] The MITRE Corporation, *Common Weakness Enumeration.* http://cwe.mitre.org/.

[27] MITRE, *The MITRE Corporation.* http://www.mitre.org/.

[28] SANS (SysAdmin, Audit, Networking, and Security), *SANS Institute.* http://www.sans.org/.

[29] SANS, *CWE/SANS TOP 25 Most Dangerous Software Errors.* http://www.sans.org/top25-software-errors/.

[30] A. Delaitre, V. Okun, and E. Fong, ``Of massive static analysis data,'' in *7th International Conference on Software Security and Reliability (SERE 2013)*, 2013.

[31] NIST, *Software Assurance Metrics And Tool Evaluation - SAMATE.* http://samate.nist.gov/.

[32] The MITRE Corporation, *Common Vulnerabilities and Exposures.* http://cve.mitre.org/.

[33] National Institute of Standards and Technology (NIST), *NIST SAMATE Reference Dataset (SRD).* http://samate.nist.gov/SRD/index.php.

[34] T. Boland and P. E. Black, ``Juliet 1.1 c/c++ and java test suite,'' *IEEE Computer*, vol. 45, no. 10, pp. 88--90, 2012.

[35] Center for Assured Software, National Security Agency, *CAS Static Analysis Tool Study - Methodology*, 2011.

[36] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* New York, NY, USA: Cambridge University Press, 2008.

[37] S. A. Alvarez, ``An exact analytical relation among recall, precision, and classification accuracy in information retrieval,'' Tech. Rep. BC-CS-02-01, Computer Science Department, Boston College, 2002.

[38] C. Willis, ``Cas static analysis tool study overview,'' in *11th annual high confidence software and systems conference*, National Security Agency, 2011. http://cps-vo.org/node/1152.

[39] T. McCabe, ``A complexity measure,'' *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308--320, 1976.

[40] Hibernate, *Hibernate ORM.* http://hibernate.org/orm/.

[41] Apache Software Foundation, *Log4j.* http://logging.apache.org/log4j/2.x/.

[42] Chris Allen Lane, *Watchtower Static Code Analysis.* https://github.com/chrisallenlane/watchtower.