Automated security review of Java card applets with static code analysis

Lie, Roberto E.A.

Award date:
2012

Automated Security Review of Java Card Applets with Static Code Analysis

Roberto Eliantono Adiseputra Lie

August 2012

Abstract

Code review is a tedious and error-prone process that can be expensive when manually done. Various tools have been developed to aid this process, mainly by using static analysis. Specific application domains such as smartcards have different unique characteristics that are not yet handled by available tools.

The main idea of this study is to bring the knowledge formalized in various knowledge catalogs into the static analysis tools. In order to do so, several static analysis tools have been evaluated and two knowledge catalogs have been analyzed. From the study it was found that in general it is feasible to utilize the knowledge from the catalogs for the tools. This can be done by either creating new rulesets and/or modifying the code of the tools, depending on the tools capabilities. For knowledge catalogs that are more abstract such as principles or guidelines, the knowledge has to be translated first into more concrete rules.

Acknowledgements

My heartfelt thanks to my supervisors: Alexander Serebrenik, Cees-Bart Breunesse, Erik Poll and Marc Witteman for all their support, guidance and feedback during the course of my master's project. Also, to the other members of the assessment committee: Ruurd Kuiper and Sandro Etalle, for their time and effort to review this work.

I would also like to thank Alec Moss, Joy Marie Forsythe and Matias Madou from HP Fortify; Dirk Giesen from Parasoft; and, Maty Siman, Moni Stern and Asaph Schulman from Checkmarx for helping me obtain the appropriate permissions to use their products.

I am grateful of all the information I received from many researchers across the information security, program analysis and Java Card communities. Special thanks to Eric Vétillard and Jean-Louis Lanet who have kindly answered my questions about their projects.

Last but not least, I would like to thank my friends and family, whose love and support have made it possible for me to come to this point.

Contents

List of Tables

List of Figures

List of Listings

CHAPTER 1

Introduction

1.1 Background and Problem Description

Security is an important aspect of software quality [BITS, 2012]. Since the emergence of *software security* in the late 1990s, various security best practices have been identified and integrated in the software development life cycle [McGraw, 2008]. One of the most common best practices is code security review [McGraw, 2006, 2008]. By incorporating code security reviews in the software development life cycle, bugs that introduce security vulnerabilities and other potential threats can be caught and remedied or mitigated early.

Code review, however, can be tedious, error-prone and costly when manually done [Chess & West, 2007; Evans & Larochelle, 2002; McGraw, 2008]. Consequently, many different tools have been developed to assist and automate (parts of) the code review process by doing static code analysis, *e.g.* ITS4, Coverity Static Analysis, HP Fortify Static Code Analyzer and IBM Rational AppScan Source Edition [Baca et al., 2008]. Although some of the tools are developed for analyzing programs written in a specific programming language, many of them support multiple languages.

The static code analysis tools work in various degrees of sophistication. Some of them, especially the earlier generations such as ITS4, simply do basic lexical analysis of the code by preprocessing and tokenizing the source, followed by matching the resulting token stream against a set of vulnerability constructs. Later generations of code analysis tools, *e.g.* HP Fortify Static Code Analyzer and IBM Rational AppScan Source Edition, take the semantics of the program into account and include more sophisticated techniques such as code and data flow analyses, producing better analysis results, both in completeness and accuracy [McGraw, 2008]. In order to have a better result without being overly complex, some tools might focus on specific classes of applications. For instance, Armorize CodeSecure only deals with web applications.

The popularity of smartcards has been continuously growing in the last few years [Ko & Caytiles, 2011; Vétillard & Marlet, 2003; Witteman, 2002]. Smartcards are cards that are equipped with a microprocessor, memory modules and suitable interfaces for off-chip communications–generally reinforced with various temper-resistance protections, allowing to do secure computations and storage in the card. A more thorough discussion about smartcards can be found in Section 2.6. In particular, smartcards are widely deployed in the banking (*e.g.* debit, credit and cash cards) and wireless (*e.g.* SIM cards) market [Rankl & Effing, 2003; Vétillard & Marlet, 2003].

Due to the nature of their usages, it is of a high importance to ensure the security of the applications written for smartcards. Unfortunately, being an embedded device, a smartcard is also vulnerable to different classes of attacks such as side-channel and fault injection, which eventually affect how the program should be written in order to ensure security. Consequently, this also means that general-purpose static code analysis tools might not be sufficient to detect security problems in smartcards applications (*aka.* applets).

On the other hand, efforts have been made by parties such as smartcard vendors, industrial consortia and solution providers to identify the common threats specific to smartcards and come up with development guidelines, programming patterns and principles (*e.g.* [AFSCM, 2012; Witteman, 2012]) to avoid or minimize the presence of vulnerabilities in the code. One way to ensure compliance with these guidelines is, naturally, by doing code reviews.

1.2 Project Goal and Scope

The aim of this project is to investigate to what extent static code analysis can be used to verify compliance with existing development guidelines and programming patterns in order to have automated security reviews of smartcard applications. In particular, the study focuses on static code analysis of Java Card applets. Java Card technology is chosen for two main reasons: it has a wide-spread use in the smartcard industry, signifying a notable impact; and, it is based on Java whose grammar is simple enough that it will not shift the focus of the study to resolving difficulties due to language complexity (*cf.* [Vétillard & Marlet, 2003]). The development guidelines and programming patterns used in this study are the AFSCM Cardlet Development Guidelines v2.2 [AFSCM, 2012] and Riscure Secure Application Programming Patterns [Witteman, 2012].

1.3 Methodology

The study is conducted in several stages. First, for each rule or pattern from the development guidelines, the information required to detect it in the code is identified. Then, the possibility and ways of obtaining such information from a static analysis are analyzed. Literature review and static code analysis tools survey are done in order to have an overview of various approaches that are already available. Finally, using the information obtained in the previous stages, a mapping between rules and the reviewed tools capabilities are created.

1.4 Related Work

This work belongs to the field of using static analysis tools for security review of Java Card applets. Static code analysis tools that specifically test Java Card Applets code for security issues are still rare. After an extensive search, only two of such tools are found. The first one is a tool that is developed by Trusted Logic, a company providing secure software for smart cards, terminals and consumer devices. The tool, capable of verifying compliance to defined portability and security policies, is briefly described in [Vétillard & Marlet, 2003]. In a personal communication, Vétillard stated that the tool is still being maintained for internal use but has not been deployed publicly. The other tool is Smart Analyzer, which is a tool developed by Jean-Louis Lanet's Smart Secure Devices Team at the Université de Limoges. According to Lanet, the tool is still under development

and will be commercially available when it is ready. A short description of the tool is available in [SSD Team, 2011]. Both tools analyze the bytecode of the Java Card applets.

All major static code analysis tools, which are described in Chapter 3, do not contain any Java Card specific rules. However, there is an extension of FindBugs that checks the absence of unhandled security critical exceptions and the temporal safety and correctness of the arguments of Java Card API calls, which is described in [Almaliotis et al., 2008]. A more recent work of the same authors is [Loizidis et al., 2011], which extends the analysis capabilities for multi-applet Java Card applications setting.

All the previously described tools were essentially built by experts in the field, incorporating their knowledge gained from experience to their specific tools. This project attempts to do this in a different manner, namely by taking available knowledge catalogs and using the knowledge contained within to enrich the knowledge-bases of general-purpose static code analysis tools.

1.5 Organization of this Report

This report is organized as follows: Chapter 1, this chapter, introduces the problem, the project goal, the approach of the study, the related work and the structure of the report. Chapter 2 provides an overview of important background knowledge related to this study. A survey of static code analysis tools is provided in Chapter 3. The analyses of [AFSCM, 2012] and [Witteman, 2012] are provided in Chapters 4 and 5, respectively. Finally, the report is concluded with some remarks in Chapter 6 and several appendices to complement the main discussion. A glossary and list of abbreviations are available in the back matter to facilitate readers with limited familiarity on the subject.

CHAPTER 2

Preliminaries

This chapter describes important background knowledge that will be referred to in later discussions. Readers who are familiar with the topic can skip this chapter and consult the glossaries whenever necessary. The background knowledge is presented in several sections. First, Section 2.1 and Section 2.2 introduce the main ideas of information and software security. A brief overview of program verification and analysis, and their applications in security are subsequently presented in Sections 2.3 to 2.5. Section 2.6 follows with an introduction to smartcards. The most popular platform of smartcards, the Java Card, is then described in Section 2.7, followed by a discussion of various kinds of attacks on smartcards in Section 2.8. Finally, the chapter is concluded with the descriptions of the AFSCM Cardlet Development Guidelines v2.2 [AFSCM, 2012] and Riscure Secure Application Programming Patterns [Witteman, 2012] in Section 2.9.

2.1 Information Security

Information security deals with the safeguarding of information or data[1]. In particular, a key aspect of information security is to preserve the *confidentiality*, *integrity* and *availability* of information. These properties are commonly referred by their abbreviations: CIA. [Wylder, 2003] defines the CIA properties as follows:

1. *Confidentiality*: The prevention of unauthorized use or disclosure of information.
2. *Integrity*: Ensuring that information is accurate, complete and has not been modified by unauthorized users or processes.
3. *Availability*: Ensuring that users have timely and reliable access to their information assets.

[De Leeuw & Bergstra, 2007] provides a comprehensive history of the information security field, highlighting various historic milestones such as: the emergence of cryptology as a discipline during the Renaissance, the breaking of German military codes during World War II, viruses and worms on the Internet, and the privacy debate. In this section, we limit our discussion to the part closely related to software security.

In the early days of computing, securing data was simple as computers were basically single-user, could not communicate with each other and only trusted people had physical access to operate

[1]There is a subtle difference between information and data. Information generally refers to interpreted data. This distinction is, however, not important in our context of security and thus the terms will be used interchangeably. Unless otherwise stated, any reference to either *data* or *information* should be understood as data and/or information.

the computers. Data would remain secure as long as the physical security of the computers were protected [Salus, 1998].

As computers became multi-user and internet was introduced, security started to be a concern. Many security incidents happened due to malicious software such as viruses, which commonly spread via the network. Earlier efforts tried to tackle this by securing the perimeters, by the means of, for instance, network security (setting a firewall, having an intrusion detection system, etc.) and various kinds of anti-virus software [de Leeuw & Bergstra, 2007; McGraw, 2008]. [Ruiu, 2006] provides an overview of the evolutionary trends in network attack and defense techniques, aimed to help predict the future of information security threats and defences.

2.2 Software Security

The next major step in the efforts was set in the late 1990s, when a new paradigm known as software security was introduced. Software security aims to build software in such way that it continues to function correctly under malicious attack. This discipline evolved upon the realization that a large number of malicious attacks exploit software defects [Chess & West, 2007; McGraw, 2004, 2008].

The idea of building security into software started to get closer attention in 2001, when Viega & McGraw wrote the seminal book of software security [Viega & McGraw, 2001]. This was then followed by several other publications such as [Howard & LeBlanc, 2002; Howard & Lipner, 2006; McGraw, 2004, 2006; Schumacher et al., 2005].

2.2.1 Bugs vs Flaws

There are two kinds of software defects: implementation bugs and design/architectural flaws. A bug is a mismatch between implementation and specification [Møller, 2003]. In some cases, this mismatch will introduce vulnerabilities, which are exploitable by malicious parties. A flaw is a deeper-level problem that is also reflected in the design of the software [McGraw, 2006]. For example, incorrect implementation of a particular encryption algorithm is a bug while not encrypting confidential information is a flaw. According to McGraw [2006], bugs and flaws contribute equally to software security problems. Table 2.1 lists a few examples of bugs and flaws.

Table 2.1: Examples of Bugs and Flaws (adapted from [McGraw, 2006])

Bugs	Flaws
Buffer overflow	Compartmentalization problems in design
Race conditions	Privileged block protection failure
Unsafe environment variables	Error-handling problems
Unsafe system calls	Insecure audit log design
Incorrect input validation	Broken or illogical access control

2.2.2 The Three Pillars of Software Security

There are three pillars of software security according to McGraw: *applied risk management*, *software security touchpoints* (best practices) and *knowledge*. We can achieve a reasonable, cost-effective software security program by applying the three pillars in a gradual, evolutionary manner and in equal measure [McGraw, 2006].

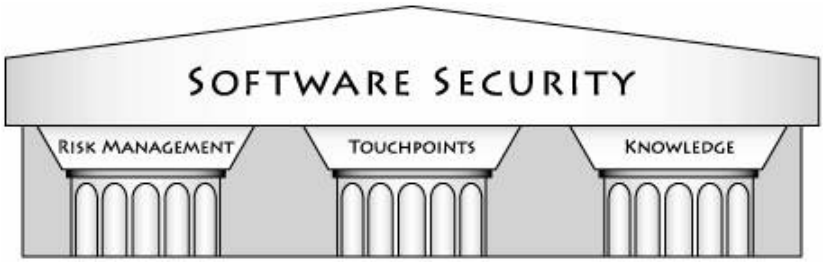

Figure 2.1: The three pillars of software security: risk management, software security best practices and knowledge [McGraw, 2006]

Security is about managing risk [Townley, 2005]. The same holds true for software security. In order to achieve secure software, it is necessary to track and mitigate risk as a full lifecycle activity and conduct a risk analysis at the architectural level (*aka.* threat modeling or security design analysis). This applied risk management constitutes the first pillar of software security.

Security best practices create the second pillar of software security. McGraw identified seven most common best practices and ranked them in their order of effectiveness [McGraw, 2006]. These best practices, dubbed as the software security touchpoints, are: *code review*, *architectural risk analysis*, *penetration testing*, *risk-based security tests*, *abuse cases*, *security requirements* and *security operations*.

The third pillar of software security involves gathering, encapsulating and sharing security knowledge that can be used to provide a solid foundation for software security practices. By utilizing this knowledge, we can make sure not to repeat any mistakes that have been made in the past.

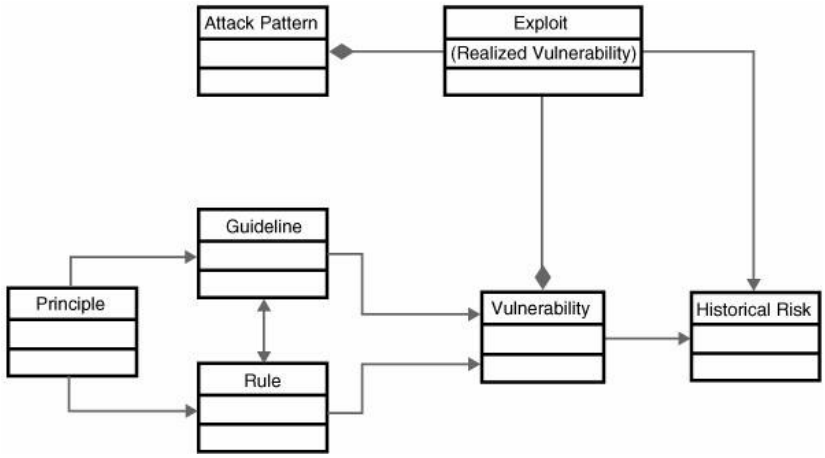

Figure 2.2: The software security catalogs and their interrelations [McGraw, 2006]

McGraw identifies seven knowledge catalogs, into which software security knowledge can be organized, as shown in Figure 2.2. These seven knowledge catalogs are: *principles*, *guidelines*, *rules*, *vulnerabilities*, *exploits*, *attack patterns* and *historical risks* [McGraw, 2006].

From the previous description in Chapter 1, we can see that this work deals primarily with the second and third pillars of software security. In particular, it focuses on automating the code review, the most effective touchpoint (second pillar), using the knowledge from the knowledge catalogs (third pillar). The following Section 2.3 discusses the basic idea of how the code review process can be automated.

2.3 Program Verification and Analysis

The goal of code review is to have bug-free software. In order to have bug-free software, we have to guarantee the correctness of the implementation of the software specification. Program verification attempts to do this methodologically. Verifying correctness of a program is one main application of the field known as program analysis.

Program verification deals with *checking* properties of the possible behaviors of a complex computer program. In contrast, program analysis focuses on *detecting* the program's properties [Cousot, 2005; Møller, 2003]. While this indicates that program analysis has a larger scope than verification (*i.e.* the answer of a verification can be derived from the respective analysis result), program analysis is not necessarily 'harder' than verification [Møller, 2003]. In practice, there is a large overlap between verification and analysis [Møller, 2003] and, therefore, the terms are often intermixed in use.

The mathematical model of the set of all possible executions of a program in all possible execution environments is known as the program's *concrete semantics* [Cousot, 2005]. If an execution is represented by a curve showing the evolution of the vector $x(t)$ of the input, state and output variables of the program as a function of the time t, this concrete semantics can be graphically represented by a set of curves. A simplified example is shown in Figure 2.3.

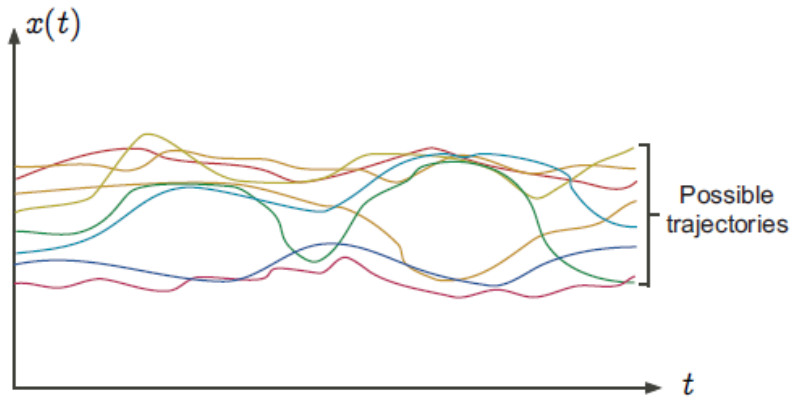

Figure 2.3: An example of a graphical representation of a program's concrete semantics. Each trajectory in the graph represents a possible execution of the program where $x(t)$ is a function over time t of the input, state and output variables of the program. [Cousot, 2005]

The concrete semantics of a program is an infinite mathematical object and hence not computable. Furthermore, it follows from Rice's theorem [Rice, 1953] that non-trivial questions[2] on the concrete program semantics are undecidable in general. Consequently, program verification/analysis is impossible, unless at least one of these following conditions is satisfied: [Cousot, 2005; Monniaux, 2011]

1. User interaction is required (the verification/analysis is not fully automatic);
2. The class of programs is constrained enough;
3. The memory is finite;
4. There is a finite number of program steps; or,
5. False/uncertain answers or incomplete results are allowed.

Consider that we are going to verify whether a program satisfies certain properties. The program's execution reaches an *erroneous state* whenever the specified properties are not satisfied. The *safety properties* of a program express that no possible execution in any possible execution environment can reach an erroneous state. A *safety proof* consists in proving that the intersection of the program's concrete semantics and the set of erroneous states (illustrated as the forbidden zones in the graphical example shown in Figure 2.4) is empty [Cousot, 2005]. Another interesting type of properties is the *liveness properties*. Informally defined, a liveness property stipulates that *good things* do happen (eventually) [Lamport, 1977]. Formal definitions of both safety and liveness properties can be found in [Alpern & Schneider, 1985]. In [Schneider, 1987], Schneider claims that every property can be decomposed into safety and liveness properties. While this claim holds for many properties, it has been shown that there are other properties that cannot be expressed in the safety/liveness framework *e.g.* information flow over covert channel [Gärtner, 2002; Rushby, 1993; Zakinthinos, 1996].

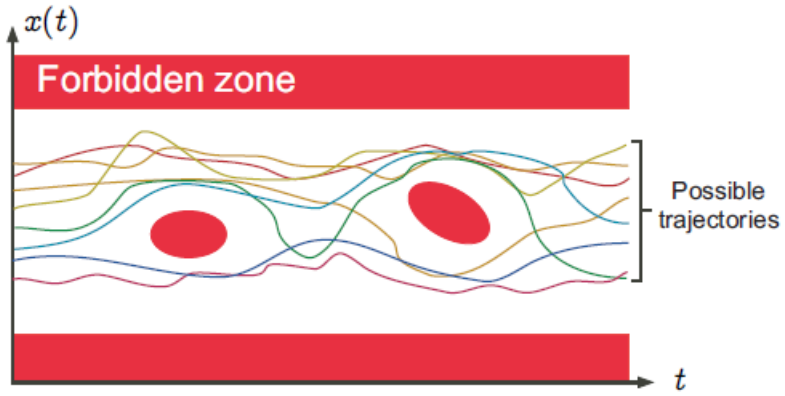

Figure 2.4: A simplified graphical representation of the safety property of a sample program. The trajectories illustrate all the possible execution paths of the particular program. The forbidden zones cover all the program's possible erroneous states. In this example, the program is safe as no trajectories intersect with the forbidden zones. [Cousot, 2005]

The program's semantics can be 'collected' either dynamically or statically:

1. *Dynamic approach*: The concrete semantics is obtained by executing (or simulating) the program with all the possible inputs in all its possible execution environments. As enumerating all possible inputs and possible execution environments is generally intractable

[2] Trivial means that the answer is always true or always false [Trevisan, 2009].

except for certain restricted classes of programs, in most cases only a subset of the concrete semantics is considered, resulting in the problem of *absence of coverage*. Consequently, a proof cannot be obtained with this approach as there might be some erroneous trajectories of the semantics that are not considered. In other words, a correctness verification (with the positive result indicating the program is correct) using this approach can result in a *false positive*, where an incorrect program will be classified as correct. On the other hand, this approach will never result in a *false negative* (*i.e.* when the verification indicates an incorrect program, the program will indeed be incorrect). An illustration of a simplified example of this approach can be seen in Figure 2.5 [Cousot, 2005].

2. *Static approach*: The concrete semantics is obtained by analyzing the program without running it. This can be done, for instance, by the means of abstract interpretation, that is by considering an *abstract semantics*, which covers the concrete semantics of the program (see Figure 2.6). By defining an abstract semantics in such way that it includes all possible concrete cases, this approach provides a full coverage and thus a proof of correctness can be obtained. The abstraction used in this approach is considered *sound* (or *correct*) if it covers all possible cases of the concrete semantics (*i.e.* the abstract semantics is a superset of the concrete semantics) [Cousot, 2005; Cousot & Cousot, 1992]. In most literature, the term program verification/analysis usually refers to this *static program verification/analysis*.

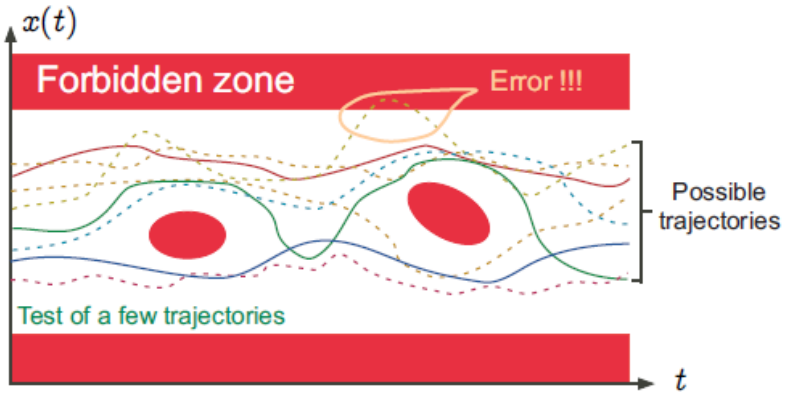

Figure 2.5: A graphical representation of a property testing that only covers a subset of the possible trajectories. The trajectories drawn in dashed lines are the trajectories that are not tested and those drawn in solid lines are the ones that are tested. In this case, the topmost erroneous trajectory is not detected as it is not in the test, resulting in a false negative report of errors. [Cousot, 2005]

There are three main approaches of (static) program verification: theorem proving, model checking and static code analysis [Henzinger, 2006]. These approaches are described briefly in the following sections.

2.3.1 Theorem Proving

The idea of formally proving the correctness of a program originated as early as 1947 by Goldstine & von Neumann in their series of reports "Planning and Coding of Problems for an Electronic Computing Instrument" [Goldstine & von Neumann, 1947] by the use of assertions as formal constraints on the behavior of a software application, supplemented with mathematical proofs for the assertions whose truths are not very obvious. The idea of using assertions was then

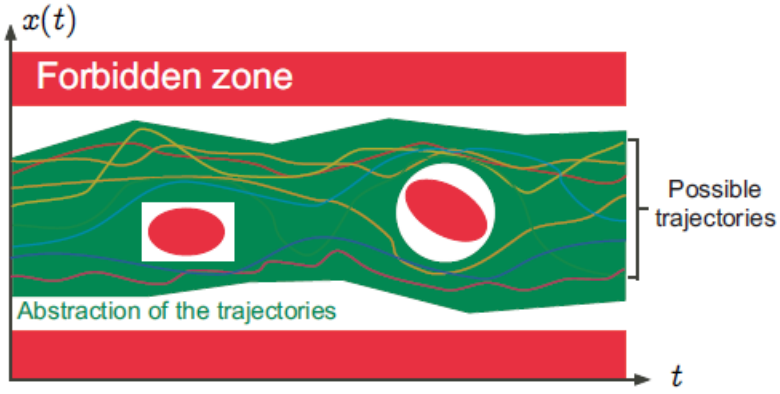

Figure 2.6: A graphical representation of abstract interpretation. The abstraction of the trajectories covers all possible trajectories and thus if the abstract semantics is safe (does not intersect the forbidden zone, as is shown in this example), so is the concrete semantics. [Cousot, 2005]

advocated by Turing in his talk "Checking a Large Routine" [Turing, 1949], which he presented at a conference in Cambrige in 1950 [Malloy & Voas, 2004].

The notion of asserting program correctness, however, was forgotten for two decades until Floyd published a paper on "Assigning Meaning to Programs" in 1967, which is now usually cited as the origin of program correctness work. A later important proposal on this subject was [Hoare, 1969]. Instead of having a monolithic approach that walks through a program from its entry point, utilizing loop invariants to jump through the loops, Hoare uses laws of inference to reason about programs. This idea was subsequently extended by Dijkstra [1975] in the concept of predicate transformers.

Methods of Theorem Proving

To formally assess the correctness of a program, we can employ *invariant assertions* or *pre/post conditions*. The invariant assertion approach defines correctness as an invariant formula which must be verified to hold throughout the program execution (or a part of it, *e.g.* a loop). This approach has it roots in the work of Floyd [1967]. The pre/post condition approach considers two different formulae: the precondition, which is assumed to hold at the beginning of a program execution, and the postcondition, which should hold at the end of the program execution. The program is deemed correct if the semantics of the program establishes the postcondition given the precondition. The calculus of this approach is described by Hoare [Hoare, 1969]. Dijkstra extended Hoare's idea in the concept of *predicate transformers*, which starts with a post condition and uses the program code to determine the pre condition required to establish the post condition [Dijkstra, 1975].

The Rice's theorem [Rice, 1953] implies that it is not possible to have fully automated theorem provers. A common way to overcome this problem is by having interactive theorem provers, which require human intervention to derive non-trivial theorems [Ouimet, 2008].

2.3.2 Model Checking

As a consequence of the Rice's theorem [Rice, 1953], fully automated correctness proving is impossible. Clarke et al. proposed another approach to automatically verify program correctness [Clarke et al., 1983]. Instead of working on a full-blown programming language, his approach involves the construction of an abstract model in the form of finite state automata and the construction of specification formulas [Clarke et al., 1983, 1986]. This approach is known as *model checking*. The concept of model checking was simultaneously researched by Sifakis in his thesis presented in 1979, which was later published in [Sifakis, 1982]. Sifakis's work was the basis of the first model checker CESAR, described in [Queille & Sifakis, 1982].

In model checking, the set of reachable states of the model is explored to ensure that the specification formulas hold. In the case that the specification formula is an invariant assertion, it will check the entire state space to ensure that it holds in all states. Due to the nature of having to exhaustively check the entire state space, model checking does not scale very well. This problem is commonly known as the *state space explosion problem* [Clarke et al., 2001]. There are numerous researches trying to overcome this problem, as discussed in [Clarke et al., 2001]. However, the need to perform a search of the state space remains an inherent limitation of verification by model checking [Ouimet, 2008].

2.3.3 Static Code Analysis

The term *static code analysis* (or simply *static analysis*) refers to any process of assessing code without executing it [Chess & West, 2007]. The idea of static analysis can be traced back to as early as 1957. Then, static analysis was used for the purpose of program optimization in the Fortran compiler. In 1978, Cousot defended his PhD thesis "Iterative methods for fixpoint construction and approximation of monotone operators on lattices, programs semantics-based analysis" (in French), which initiated a series of work in abstract interpretation, an important concept in the program analysis field. In the static analysis approach, program correctness is verified utilizing the facts obtained by the performed analysis. Static analysis originated in the early days of programming languages for the purpose of code optimization. It is, therefore, unsurprising that it has become an integral part of compiler development.

During compilation, program code is first passed to a *lexer* (*lexical analyzer*, also known as *scanner*), resulting in a sequence of *tokens*. The process is then continued with syntactic analysis done by the *parser*, which creates a *syntax tree* based on the grammar of the programming language.

From the syntax tree, we can get the sequence of program statements. It is then possible to create a *control flow graph*, which is a directed graph in which each node represents a basic block and each edge represents the flow of control between basic blocks [Harrold et al., 2005]. A *basic block* is a sequence of consecutive statements which has a single entry point at the beginning and a single exit point at the end [Backus et al., 1957; Harrold et al., 2005].

By having the control flow graph, we have the flow information of the control of the program. Another important flow information is the data flow. Data flow information is obtained by doing a *data flow analysis*.

Data flow analysis is performed in two steps. In the first step, the desired facts are collected using algorithms commonly known as the gen/kill algorithms. A gen/kill algorithm is an efficient algorithm that passes over every statement in the code and checks if one or some of the variables in the statement should be added to the gen (generate) set or the kill set. The condition of adding the variables into the sets depends on the kind of performed data flow analysis. The output of the algorithm is the gen sets and the kill sets of the program. The number of the sets depends on the number of statements in the program. These sets are subsequently used to set up and solve equations in the second step of data flow analysis.

There are two different ways of how equations in data flow analysis can be set up:

1. *Forward analysis*: Equations are set up by transferring information from the initial statement to the final statement.
2. *Backward analysis*: Equations are set up by transferring information from the final statement to the initial statement.

More concretely, the scheme for setting up the equations in forward analysis is:

$$entry_\ell = \begin{cases} \emptyset & \text{if } \ell \text{ is an initial statement} \\ \bigcup exit_{\ell'} & \text{where } \ell' \text{ is an ancestor statement of } \ell \end{cases}$$

$$exit_\ell = entry_\ell \setminus kill_\ell \cup gen_\ell$$

On the other hand, the scheme for backward analysis is:

$$entry_\ell = exit_\ell \setminus kill_\ell \cup gen_\ell$$

$$exit_\ell = \begin{cases} \emptyset & \text{if } \ell \text{ is a final statement} \\ \bigcup entry_{\ell'} & \text{where } \ell' \text{ is an ancestor statement of } \ell \end{cases}$$

There are various applications of data flow analysis and whether forward or backward analysis is needed depends on the application. [Nielson et al., 1999] discusses this in more detail.

Consider the following program (written in the WHILE programming language [Nielson & Nielson, 1992]) as an example:
`[x:=1]`^{1}`; [y:=3]`^{2}`; [x:=2]`^{3}`; (if [y>x]`^{4} `then [z:=y]`^{5} `else [z:=y*y]`^{6}`); [x:=z]`^{7}

If we want to determine the live variables (variables that might still be used at a later point of a program), we have the following rules for the gen/kill algorithm [Wögerer, 2005]:

1. For the *gen* set: If a statement ℓ is an assignment then gen_ℓ consists of all variables of the right hand side. If the statement is a condition then gen_ℓ consists of all variables of that condition. Otherwise, the set is empty.
2. For the *kill* set: If a statement ℓ is an assignment then $kill_\ell$ consists of the variable in the left hand side. Otherwise, the set is empty.

Applying the gen/kill algorithm in the first step, we have:

ℓ	$kill_\ell$	gen_ℓ
1	$\{x\}$	$\emptyset$
2	$\{y\}$	$\emptyset$
3	$\{x\}$	$\emptyset$
4	$\emptyset$	$\{x, y\}$
5	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$

Noting that we need to do backward analysis, in the second step we get the following equations:

$$entry_1 = exit_1 \setminus \{x\} \qquad exit_1 = entry_2$$
$$entry_2 = exit_2 \setminus \{y\} \qquad exit_2 = entry_3$$
$$entry_3 = exit_3 \setminus \{x\} \qquad exit_3 = entry_4$$
$$entry_4 = exit_4 \cup \{x, y\} \qquad exit_4 = entry_5 \cup entry_6$$
$$entry_5 = (exit_5 \setminus \{z\}) \cup \{y\} \qquad exit_5 = entry_7$$
$$entry_6 = (exit_6 \setminus \{z\}) \cup \{y\} \qquad exit_6 = entry_7$$
$$entry_7 = \{z\} \qquad exit_7 = \emptyset$$

Solving the equations, we have the result of the live variable analysis:

$$entry_1 = \emptyset \qquad exit_1 = \emptyset$$
$$entry_2 = \emptyset \qquad exit_2 = \{y\}$$
$$entry_3 = \{y\} \qquad exit_3 = \{x, y\}$$
$$entry_4 = \{x, y\} \qquad exit_4 = \{y\}$$
$$entry_5 = \{y\} \qquad exit_5 = \{z\}$$
$$entry_6 = \{y\} \qquad exit_6 = \{z\}$$
$$entry_7 = \{z\} \qquad exit_7 = \emptyset$$

The ways to solve data flow analysis equations are described in detail in [Nielson et al., 1999].

Another example of the application in software security is for tracking taintedness of data. Data are considered tainted when they are obtained from the user and have not been validated, and thus might pose security risks. In taint analysis, the gen set will contain values that are obtained from an untrusted source. The values that have been validated are put into the kill set. The analysis can be done in either forward or backward fashion [Young & Pezze, 2008].

Further Reading

The idea of program verification is discussed thoroughly in [Fetzer, 1988]. Theorem proving, model checking and static analysis are relatively mature subjects and there is a large body of literatures covering these areas. [D'Silva, 2009] provides a historical account of program verification. The model checking and theorem proving approaches are discussed in more detail in [Ouimet, 2008]. There are various approaches in static analysis. [Harrold et al., 2005; Nielson et al., 1999; Pistoia et al., 2007; Wögerer, 2005] provide an overview of some of the important ones.

2.4 Bug Finding

When a program becomes large and complex (*e.g.* is multi-threaded, contains various complex data structures) or when resources (*e.g.* time and computing power) are limited, program verification might not be feasible. An alternative approach is to try to detect bugs by analyzing the code and utilizing prior knowledge of possible bugs. As it is simply not possible to know all kinds of possible bugs beforehand, this approach may result in a *false negative*, *i.e.* no bugs are detected while actually there are some. Depending on the knowledge and how it is applied, this approach may also result in *false positive*, where correct code is deemed as bug. While this approach does not guarantee correctness, it can be good enough in practice when properly done.

As discussed previously, there are two different approaches to analyze code, statically or dynamically. Dynamic program analysis suffers from the problem of coverage as its completeness depends heavily on the tested program execution and it is generally not feasible to exhaustively test every possible execution path (except for very restricted programs). It is, therefore, unsurprising that static analysis is used instead in many bug finders.

A bug finder receives program code as an input. It will then do static analysis on the code, collecting necessary information. The gathered information will then be compared with the stored knowledge of possible bugs, usually in the form of rules, giving the analysis result. The analysis result typically contains the descriptions of the found possible bugs and where the offending code locations are.

From the above description of how a bug finder works, it is fairly obvious that it combines well with code review. In code review, human reviewers will read the code and apply their knowledge to spot bugs in the code. The commonly accepted best practice is to do code review with the help of bug finding tools [Chess & West, 2007; McGraw, 2008]. The whole process now becomes as shown in Figure 2.7. Note that while commonly a code review is done with source code as input (as depicted in the picture), in some cases lower-level code such as bytecode might be used instead.

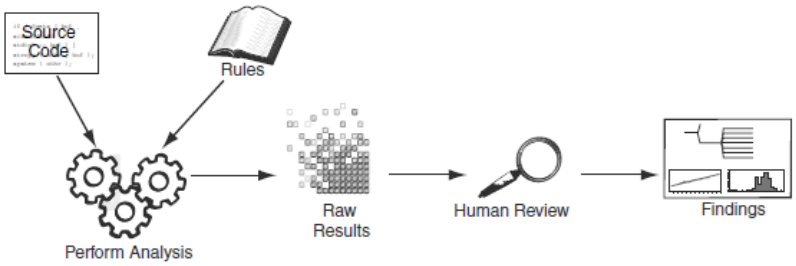

Figure 2.7: Code review with bug finding tools [Chess & West, 2007]

This review process automates some parts of review and, therefore, helps reduce human reviewers' work. It might also reduce overlooked bugs which sometimes happen in manual code review as bug finders will apply their knowledge consistently and objectively throughout the code [Hovemeyer & Pugh, 2004]. Furthermore, the involvement of human reviewers can reduce the occurrence of false positive and/or false negative, leading to a better overall review result.

2.4.1 Bug Finding Tools

Various tools have been developed for bug finding purposes. One of the best known early bug finders is Lint [Johnson, 1978]. Lint uses heuristics to find a variety of common errors in C programs. Many of the checks performed by Lint, such as uses of uninitialized variables, are now integrated into compilers [Hovemeyer & Pugh, 2004]. More modern bug finders include LCLint [Evans et al., 1994], Jlint [Artho & Biere, 2001] and FindBugs [Hovemeyer & Pugh, 2004].

2.5 Verification and Bug Finding in Software Security

There is more to security than ensuring that all software requirements are fulfilled in the implementation. In particular, secure software should do only what it is supposed to do, nothing else. [Whittaker & Thompson, 2003] illustrates this with a diagram as shown in Figure 2.8.

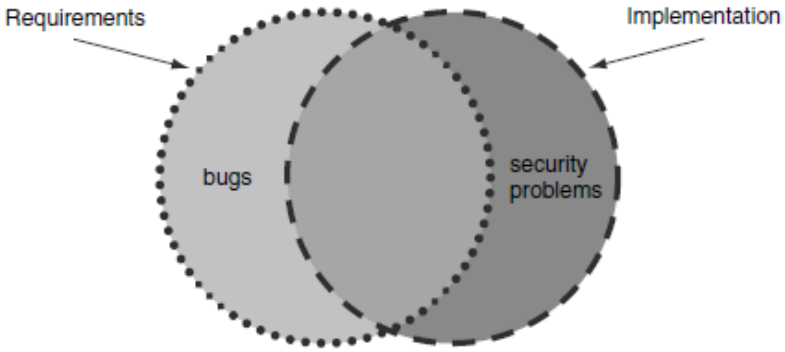

Figure 2.8: The relation between requirements, implementations, (functional) bugs and security problems [Whittaker & Thompson, 2003]

Numerous researches have been done to bring program verification into the realm of security. [Chess & West, 2007] contains a discussion of the notable ones. However, as noted in [Chess & West, 2007], program verification does not scale very well in practice. Even though a lot of progress has been made in the area of program verification, the "cheaper" bug finding approach is still more commonly used due to its being less demanding in resources.

The first bug finding tool known in literature that was specifically built for detecting security issues was ITS4 [Viega et al., 2000]. It uses simple lexical analysis to find dangerous function calls in programs written in C or C++. Other early security bug finders such as Flawfinder [Wheeler, 2007] and RATS [Chess, 2009] also employ a similar approach, using only basic lexical analysis and a database of vulnerable constructs. More modern tools such as Fortify [Fortify, 2012b] and Coverity [Coverity, 2012] use various static analysis techniques to collect information about the analyzed program. While not originally built to identify security issues, bug finding tools like FindBugs [Hovemeyer & Pugh, 2004] have been enriched with security-related rules in their later developments.

A discussion of several popular bug finding tools that use a static analysis approach can be found in Chapter 3. Other literature that discusses and/or compares bug finders and other related tools include [Ayewah & Pugh, 2008; Chess & West, 2007; Feiman & MacDonald, 2010; Rutar et al., 2004; Wagner et al., 2005; Ware & Fox, 2008]. [Chess & West, 2007] contains a list of various benchmarks to analyze the performance of the tools.

2.6 Smartcards

A smartcard is a card (generally in a business card form factor or smaller) containing a microcontroller with a microprocessor, volatile and non-volatile memory, allowing on-card computing[3] [Rankl & Effing, 2003]. The smartcard technology evolved from an early effort to replace magnetic-stripe technology, which was deemed to be not secure and thus not suitable for storing confidential data. By having computing capability, it is possible to enforce access control on-card, safely guarding stored information from unauthorized access. Smartcards have since found fairly broad applications ranging from electronic tickets to identity cards. A historical account on smartcards can be found in [Rankl & Effing, 2003]. Smartcard technology has been standardized mainly in ISO-7816, which will be the base of our discussion in this section. [DIN, 2010] provides pointers to other card standards.

2.6.1 Hardware

There are two main variants of smartcards, the *contact smartcards* and *contactless smartcards*. As the name implies, they differ on the way they interact with the card readers (*aka.* terminals, Card Acceptance Devices (CADs)). In order to use a contact card, a user will need to insert the card into the reader. On the contrary, for a contactless card, the user can simply hold the card in close proximity to the reader, without any need of contact between the card and the reader. A third variant that combines both contact and contactless interface is also available, known as *dual interface smartcards* or *combicards*.

2.6.1.1 Contact smartcards

A contact smartcard can be easily identified by its characteristic chip contact stamp. Several examples of this contact stamp are shown in Figure 2.9.

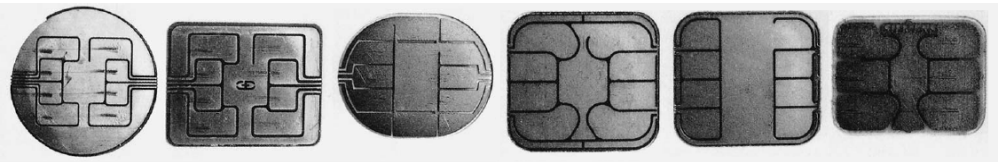

Figure 2.9: Various contact stamps of smartcards [Rankl & Effing, 2003]

A smartcard chip contains eight or six contact fields. In the eight-contact configuration, two of the contacts are reserved for future use. The contacts are typically numbered C1 to C8 from the top-left to the bottom-right, as shown in Figure 2.10. The function of each contact is described in Table 2.2.

[3] In practice, the term is also used to refer to memory cards, which have no computing capabilities. We will, however, exclude this from our discussion.

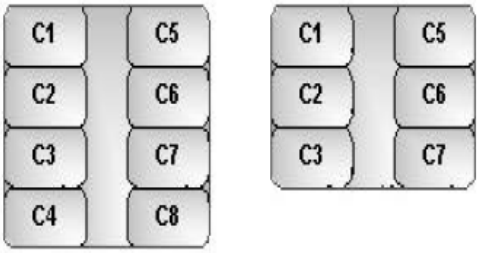

Figure 2.10: Smartcard electrical contacts configuration, showing both eight and six-contact variants. [Zanero, 2002]

Table 2.2: Function description of smartcard electrical contacts [Zanero, 2002]

Position	Abbreviation	Function
C1	Vcc	Supply Voltage
C2	RST	Reset
C3	CLK	Clock Frequency
C4	RFU	Reserved for future use
C5	GND	Ground
C6	Vpp	External programming voltage
C7	I/O	Serial input/output communications
C8	RFU	Reserved for future use

The smartcard is powered through the Vcc and GND contacts of the chip. The RST contact allows a hard restart of all processes. The CLK contact provides an external clock signal to the chip as smartcard processors usually do not have internal clock generators. The Vpp contact exists historically to provide a higher voltage for programming and erasing the EEPROM, but is now rarely used due to the advent of charge pumps that are built in newer chips. The I/O contact is used for communication between smartcards and their terminals. The two RFU contacts serve no functional standard purposes in ISO-7816 and are often omitted to save manufacturing costs [Zanero, 2002] or used for features such as a USB interface [Deshmukh, 2011].

2.6.1.2 Contactless smartcards

Unlike contact smartcards, contactless smartcards are usually not so easily recognizable as the chips are hidden inside the cards. Instead of using electrical contacts, contactless smartcards use electromagnetic induction to provide power from the terminal to smartcard as well as for communication between them.

2.6.1.3 Chip Architectures

Regardless of the type, smartcards share similar chip architectures. The basic smartcard chip architecture is shown in Figure 2.11.

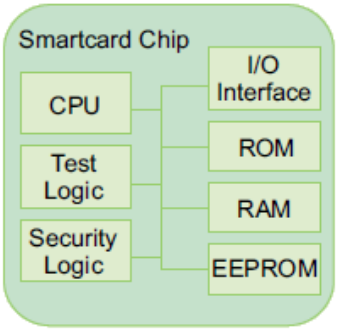

Figure 2.11: Basic smartcard chip architecture [Witteman, 2002]

The *CPU* of smartcard is traditionally an 8-bit microcontroller, but more powerful 16 and 32-bit chips are being used increasingly recently [Bezakova et al., 2000]. It is the heart of the chip and does all or most of the computational work. A cryptographic coprocessor is sometimes included to improve performance.

The memory of a smartcard consists of the ROM, RAM and EEPROM. The *ROM* is the permanent (non-rewritable) memory of the chip and generally contains the essential parts of the operating system and some other basic functions (varying depending on the manufacturer, *e.g.* self test procedures, encryption algorithms). The *RAM* is rewritable, fast and volatile (*i.e.* the content is lost when there is no power) and thus serves as the CPU's scratch pad memory for temporary data such as session keys and stack data. The available RAM of smartcard is usually very limited, around 1–2 kB. The *EEPROM* is also rewritable. It is generally slower than the RAM but non-volatile and, therefore, is used for storing application data. It might also be used to store updates to the operating system or any other pre-installed data in ROM. The EEPROM of a smartcard is typically around 8–72 kB, although larger sizes exist.

Other components of the smartcard are the I/O Interface, the Security and Test Logic, and the Data Bus. The *I/O Interface* handles the communication function of the smartcard by receiving commands from a terminal and sending back responses using a serial communication protocol. The physical security of the smartcard is protected by the *Security Logic*, which checks the environmental conditions for possible threats to the security of the smartcard. The *Test Logic* is a verification function to test all internal circuits for defects during manufacture. It is only used during the production process of the smartcard. Finally, the components are connected with the *Data Bus*, which allows information to be exchanged between functions.

2.6.2 Communication

A smartcard operates in a master-slave relationship with its terminal, with the terminal as master and the card as slave. This means that communication with the card is always initiated by the terminal. The card will then respond accordingly to the commands from the terminal. This also means that the card will never send any data without an external stimulus.

There are 15 defined protocols for smartcard data transmissions, as summarized in Table 2.3. For practical reasons, generally not all options will be implemented by smartcards. The most commonly used are the T=0 and T=1 protocol, with T=0 being the most popular [Zanero, 2002]. More information about the protocols can be found in [Jurgensen & Guthery, 2002; Rankl & Effing, 2003].

Table 2.3: Summary of smartcard transmission protocols (as specified in ISO/IEC 7816-3) [Rankl & Effing, 2003]

Protocol	Description
T=0	Asynchronous, half-duplex, byte oriented
T=1	Asynchronous, half-duplex, block oriented
T=2	Asynchronous, full duplex, block oriented
T=3	Full duplex
T=4	Asynchronous, half-duplex, byte oriented, extension of T=0
T=5–13, 15	Reserved for future use
T=14	For national use

Messages exchanged during data transmissions are structured in Application Protocol Data Units (APDUs). These data units are designed to be independent from the transmission protocol. In other words, the content and format of an APDU must not change when a different transmission protocol is used and it should be possible to transmit them transparently using both the byte-oriented and block-oriented protocols.

There are two kinds of APDUs: Command APDUs (C-APDUs), which represent commands to the card, and Response APDUs (R-APDUs), which represent replies to the commands from the card. During transmissions, the transmission protocol converts the APDUs into their respective protocol-dependant Transmission Protocol Data Units (TPDUs). The structures of C-APDU and R-APDU are described in Section 2.6.2.1 and Section 2.6.2.2, respectively.

2.6.2.1 C-APDU Structure

A C-APDU consists of two parts: header and body, as shown in Figure 2.12. The header contains four elements. The first element is the class byte (CLA), which is used to identify applications and their specific command sets. The second one is the instruction byte (INS), which encodes the actual command. Only even values can be used for command identifiers because the odd value space of this byte is used by the T=0 protocol to control voltage. The next two bytes, P1 and P2, are used primarily to provide more information about the command selected by the instruction byte.

The body part of C-APDU contains the length of the data (Lc: length command), the content of the data and the length of the expected response data (Le: length expected). The Lc and Le are typically one byte values. However, if larger values are necessary, it is possible to use values up to 65536 by using three bytes, with the first byte being 0x00 as an escape sequence.

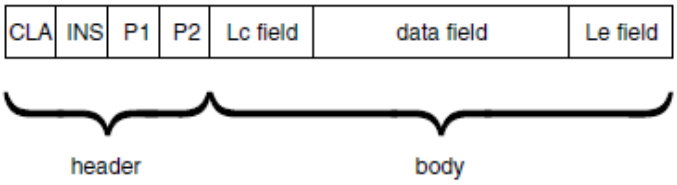

Figure 2.12: Structure of a C-APDU [Rankl & Effing, 2003]

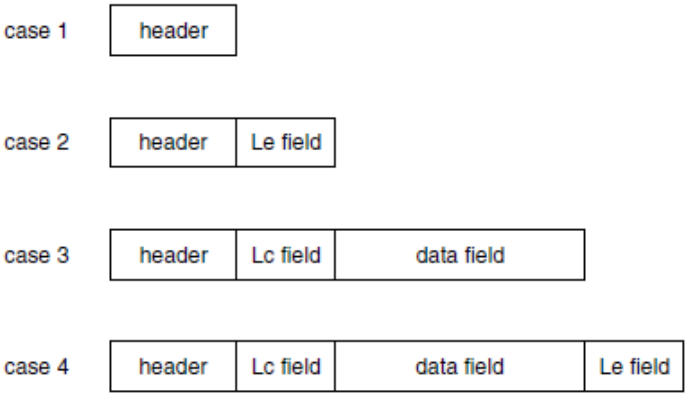

Figure 2.13: Four possible C-APDU cases [Rankl & Effing, 2003]

A command can contain no data or expect no data in response. In these cases, certain fields may be omitted, giving four possible cases (see Figure 2.13). In the first case, there is no command data and no response data is expected. In the second case, there is no command data, but response data with length Le is expected. The third case represents the case where command data is present but no response data is expected. Finally, the fourth case represents the case where command data is present and response data is expected.

2.6.2.2 R-APDU Structure

The R-APDU also consists of two parts: the body and trailer. The body contains the data field with the length of Le byte as specified in the C-APDU. This part might be omitted if no data is present (either because no response data is expected or due to some errors). The mandatory trailer part comprises of two single-byte status words SW1 and SW2, which are also known as the return code.

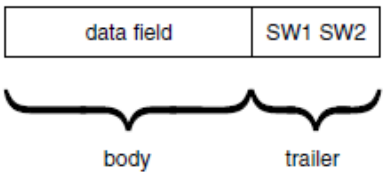

Figure 2.14: Structure of a R-APDU [Rankl & Effing, 2003]

2.6.3 Software

The earliest generation of smartcards uses custom programs written for specific purposes. Through a series of generalizations and extensions, this results in general-purpose operating systems that

are being used in modern smartcards. The most popular smartcard operating system is the Java Card, which is described in Section 2.7 [Zanero, 2002].

All modern smartcards also have complete hierarchical file management systems with symbolic, hardware-independent addressing. The file management systems are typically object-oriented. Each file consists of a header, which contains information about the layout and structure of the file and its access condition, and a body containing the actual data, which is linked to the header by a pointer.

There are two types of file in smartcards: Dedicated Files (DFs) and Elementary Files (EFs). A DF is basically a directory, which contains lower-level DFs and EFs. An EF is a regular file containing user data. The root directory of the smartcard file system is also known as the Master File (MF). When a smartcard contains several applications, files for each application are usually grouped into a single DF, which is sometimes also referred as the Application Data File (ADF).

2.7 Java Card

The Java Card platform is a variant of Java technology that is specifically targeted for smart cards and other devices with limited memory and processing capabilities. It allows multiple applications (also called applets or cardlets) written in Java dialect to be run in a single smartcard.

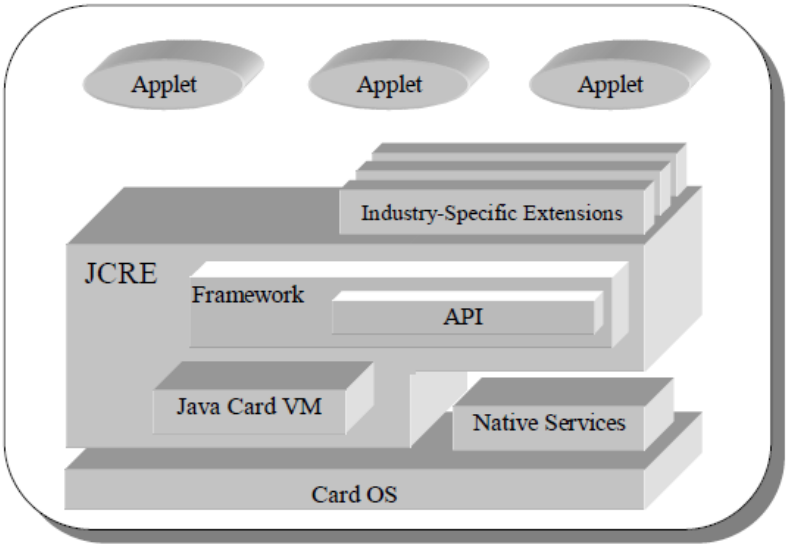

Figure 2.15: Java Card Technology Architecture [Sun Microsystems, 1998]

The Java Card platform (Figure 2.15) consists of the following components [Sun Microsystems, 1998]:

1. *Card OS*: The card operating system.
2. *Native services*: Perform the I/O, cryptographic and memory allocation services of the card.
3. *Virtual Machine (VM)*: Provides bytecode execution and Java language support, including exception handling.
4. *Framework*: The set of classes which implement the Application Programming Interface (API). This includes core and extension packages. Responsibilities include dispatching of APDUs, applet selection, managing atomicity and installing applets.

5. *API*: Defines the calling conventions by which an applet accesses the Java Card Runtime Environment (JCRE) and native services.
6. *JCRE*: Includes the Java Card VM, the framework, the associated native methods and the API.
7. *Industry extensions*: Add-on classes that extend the applets installed on the card.
8. *Applets*: Programs written in the Java programming language for use on a smart card.

For practical reasons, Java Card Virtual Machine (JCVM) is implemented as two separate pieces, one off-card and one on-card. The on-card JCVM executes bytecode, manages classes and objects, enforces separation between applications (firewalls), and enables secure data sharing. The off-card JCVM contains a Java Card Converter tool. The Java Card Converter tool performs verifications, preparations, optimizations and symbolic resolutions of the code, allowing the on-card JCVM to be more compact and efficient.

There are differences in the language specifications between the Java platform and the Java Card platform, due to the resource limitations of smartcards. The Java Card language differences from the standard Java, as described in [Sun Microsystems, 1998], are as follows:

- no multithreading (will be supported in the newer Java Card 3 platform), optional garbage collection
- only supports *byte*, *short*, *boolean* and, for 32-bit cards, *int*
- only single-dimensional arrays are supported
- more compact core classes, with extra classes to deal with specific smartcard peculiarities (APDU, PIN, etc.)
- use of shared, system-wide exception objects
- simpler `Object` class with `equals` as the only supported method

Each Java Card applet is assigned a unique Application Identifier (AID), which is used to identify a particular applet for selection in the Java Card multi-applets environment. Applet code typically includes:

1. An applet *constructor*, which will be instantiated by the `install` method. Usually a call to `Applet.register` is done here to register the applet with the JCRE to ensure future communication (when deciding where to route messages, the JCRE consults the list of registered applets);
2. An `install` method, invoked by the JCRE as the last step of the applet installation process;
3. A `select` method, which will be invoked when the applet is selected. This is especially useful for initialization;
4. A `process` method, which processes received APDUs, performs appropriate actions and returns responses to the terminal;
5. A `deselect` method, which will be invoked when the applet is deselected (*i.e.* another applet is selected); and,
6. An `uninstall` method, which will be invoked when the JCRE is preparing to delete the applet instance.

The applet life-cycle begins when the applet is downloaded to the card and the JCRE invokes the applet's `Applet.install()` method, which includes a call to `Applet.register()` to register the applet to the JCRE. Once the applet is installed and registered, it is available for selection

and APDU processing, but still inactive. When the JCRE is asked to select an applet, it calls the respective `select()` method. Any incoming APDU commands are passed to the selected applet for processing by invoking its `process()` method.

2.7.1 GlobalPlatform

The Java Card platform does not specify how applets should be managed. GlobalPlatform, an independent not-for-profit smartcards standardization organization, has created the GlobalPlatform Card Specification [GlobalPlatform, 2006] which tackles this issue. This section briefly describes the security domains and the applet installation procedure, which are referred extensively in the AFSCM guidelines [AFSCM, 2012]. More detailed descriptions of GlobalPlatform can be found in [Markantonakis, 2008].

Security Domains (SDs) are special on-card management applications that support security services such as key handling, encryption, decription, digital signature generation and verification for their providers' (Card Issuer, Application Provider or Controlling Authority) applications [GlobalPlatform, 2006]. It guarantees the isolation and security of each application. Each service provider has its own security domain and maintains full control over it; no other service provider can access it or eavesdrop on its transactions [Gemalto, 2008].

The applet installation is done in two phases: the *loading phase*, in which the applet is downloaded into the card, and the *installation phase*, in which the applet is instantiated [Gemalto, 2009].

The load phase consists of an INSTALL [for load] command and one or more LOAD commands. These commands are processed by the SD before further additional verification and processing by the card is done. Once the load phase is done successfully, the install phase takes place. Upon the receipt of the INSTALL [for install] command, the applet instance is created and usually followed by registering the applet instance to the card's registry. The installation is completed by making the applet selectable using the INSTALL [for make selectable] command. The whole process is shown in Figure 2.16.

2.7.2 SIM Toolkit

One major use of smartcards is for the Subscriber Identity Module (SIM) cards or the newer Universal Integrated Circuit Cards (UICCs), which are used to identify subscribers in the Global System for Mobile Communications (GSM) network. As has been discussed earlier, smartcards work in a master-slave configuration and the cards can never initiate commands. However, there is a need for SIM cards to be able to do so. A standard called SIM Application Toolkits (STKs) is developed for SIM applications. Among other things, the standard defines a set of *proactive commands*, which are commands initiated by the cards. In order to circumvent the master-slave configuration and allow the proactive commands, the reader (in this case commonly the phone) that supports the STK standard will poll the card periodically to see if there is any proactive command waiting to be executed.

[Mayes & Evans, 2008] provides a more thorough overview of STK and the usage of smartcards for mobile communications in general.

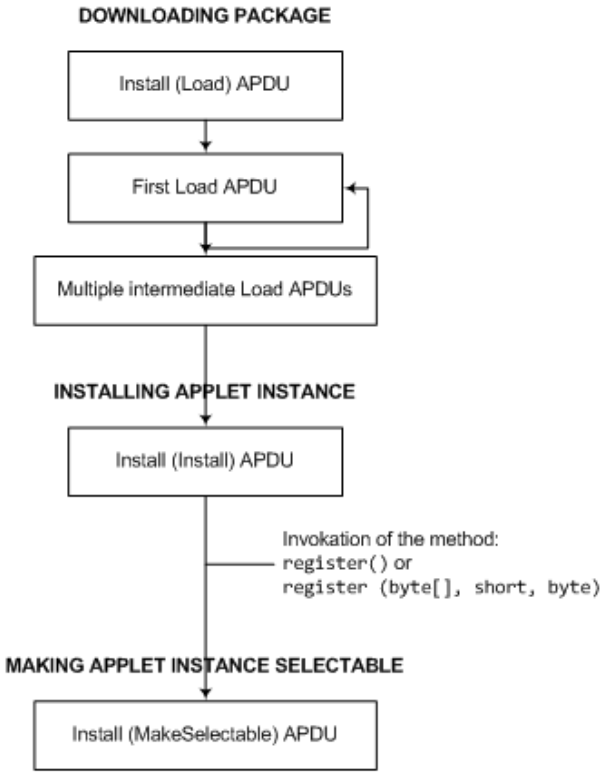

Figure 2.16: Applet installation process, based on [Gemalto, 2009]

2.8 Attacks on Smartcards

The main categories of attacks on smartcards are: [Ko & Caytiles, 2011; Witteman, 2002]

- *Logical attacks*, which exploit defects in the software implementation;
- *Physical attacks*, which analyze or modify the smartcard hardware; and,
- *Side channel attacks*, which use physical phenomena to analyze or modify the smartcard behaviour.

Logical attacks are common with any kinds of applications and essentially are what software security aims to tackle. Physical attacks require hardware countermeasures and are beyond our scope of discussion. Side channel attacks are not generally a problem in basic desktop applications but pose a high risk in embedded devices such as smartcards. Side channel attacks use physical phenomena such as power consumption, electromagnetic radiation and time (side-channel information) to analyze or manipulate the behavior of a smartcard chip. There are two major side channel attacks:

- *Side-channel analysis*: By analyzing observed side-channel information, it is often possible to deduce secret information. An example of this is the power analysis attacks, which utilize the fact that the power consumption of a device often correlates with the data and the operations processed by the device. Power analysis attacks have been successfully used in various settings to reveal secret information such as the keys of cryptographic algorithms.
- *Fault injection*: Smartcards are designed to operate in certain specified environment. It is often possible to change the behavior of a smartcard by manipulating the environmental

conditions. For example, the microprocessor of a smartcard is designed to operate from a stable voltage. By introducing well-tuned glitches in the power supply, it is possible to, for example, affect the flow of the program.

2.9 Knowledge Catalogs

This project reviews two knowledge catalogs: the AFSCM Cardlet Development Guidelines v2.2 [AFSCM, 2012], which will be simply referred as the AFSCM guidelines from this point and the Riscure Secure Application Programming Patterns [Witteman, 2012], which will be referred as the Riscure guidelines.

AFSCM is a consortium of French Mobile Network Operators (MNOs), service providers and technology vendors. Their cardlet guidelines are thus targeted for STK applets, although some of them are also applicable for Java Card applets in general. Riscure is a company specializing in the security analysis of smart cards and embedded devices.

The Riscure guidelines provide programming patterns to avoid side channel attacks. The AFSCM guidelines focus on defending against logical attacks. By utilizing these two knowledge catalogs, we can expect to have a broad coverage of software security knowledge to be implemented in the static code analysis tools.

CHAPTER 3

Survey of Static Code Analysis Tools

Various tools have been developed to statically analyze program code. This section describes several popular static code analysis tools that are capable of analyzing codes written in the Java programming language, including both commercial and open-source ones.

For the commercial tools, a particular tool is considered popular if it is included in the recent Gartner Magic Quadrant for Static Application Security Testing report [Feiman & MacDonald, 2010], which is compiled from customer surveys and interviews with various companies. Gartner Magic Quadrant for SAST does not include tools with a small user base.

For the open-source tools, the popularity is measured by the total number of downloads of such tools. In this survey, a cut-off threshold of 25 000 downloads is enforced. Additionally, any tool that has not been updated for more than 6 months as of March 2012 is excluded from review, unless there is evidence that indicate that the tool is still being actively developed.

All information about a tool is obtained from the tool's official documentations, unless otherwise stated. In the case that some information is not available in the documentation, the following alternative sources of information are consulted: the source code of the tool (if available), results of experiments and/or personal communications with the developers (in descending order of preference). To avoid unnecessary efforts, it is assumed that the official documentations shipped with the tools are correct and thus will not be cross-checked with the source codes, unless there is a suspicion that the information provided by a particular documentation is incorrect[1]. Whenever applicable, information about tools' capabilities are confirmed by running the tools against several test cases. In particular, the test suite developed by [Ware, 2008] is used. In all cases, when information about a tool is obtained from alternative sources, the particular source from which the information is obtained is indicated.

Table 3.1 provides an overview of all the surveyed static code analysis tools. More detailed discussions of each tool are available in the following sections.

[1] An example of this is the discrepancy of how rules are specified in the program's argument between the documentation and the code of the alpha release of PMD 5.0.0, which has been reported to the developers and subsequently fixed in the final release of the documentation.

Table 3.1: Overview of the surveyed SCA Tools

	HP Fortify SCA	Parasoft Jtest	Checkmarx CxSuite	PMD	FindBugs	Jlint
Version	3.5	9.2.3	6.20	5.0.0	2.0.1	3.1.2
Input	Source code	Source code	Source code	Source code	Bytecode	Bytecode
Languages	ASP(.NET) VB(.NET), C/C++, C#, Java, JSP, PHP, Python and 10 others	Java	Java, JSP, C/C++, C#, VB.NET, ASP(.NET), PHP and 5 others	Java, JavaScript, XML, XSL and JSP	Java	Java
Written In	Java	Java	C#	Java	Java	C++
Supported OS	Windows, MacOS X, Linux, Solaris, HP-UX, AIX, FreeBSD	Windows, Linux, Solaris, MacOS X	Windows	Windows, Unix, others (Java 1.5)	GNU/Linux, Windows, MacOS X, others (Java 1.5)	Windows, Unix
# Java Rules	325	1096	218	280	401	51
Custom Rules	Yes	Yes, structural only	Yes	Yes	Yes	No
License	Proprietary	Proprietary	Proprietary	BSD-style	LGPL	GPL
Structural	AST	AST	AST	AST	BCEL object	parse tree[2]
Control Flow	interprocedural, state machine	interprocedural[3]	interprocedural[4]	intraprocedural[5]	intraprocedural + call graph	interprocedural[6]
Data Flow	interprocedural, taint propagation	interprocedural, several predetermined analyses	interprocedural, influence analysis	intraprocedural, def-use tracking	interprocedural, self-implementable generic (forward & backward) + several specific analyses	interprocedural, value tracking

3.1 Commercial Tools

3.1.1 HP Fortify Static Code Analyzer

HP Fortify Static Code Analyzer (Fortify) is a set of software security analyzers that search for violations of security-specific coding rules and guidelines in a variety of languages [Fortify, 2012d].

There are six different analyzers in Fortify:

1. Data Flow: detects potential vulnerabilities involving tainted data (user-controlled input) being put to potentially dangerous use. This is done by examining the flow of data between

[2] Jlint does not explicitly create a parse tree, directly doing the analysis during parsing instead.

[3] Jtest does not expose the control flow information for user's direct access and only uses it internally for its data flow analyzer.

[4] According to the user guide, CxSuite builds a flow graph of the code. The analysis capabilities are, however, not explicitly specified. As the product is not open source, the capabilities are deduced from the available rules (whose source are accessible from CxAudit) and analysis results.

[5] PMD's control flow capability is integrated in the data flow module and not available separately.

[6] Jlint uses control flow information for its data flow and lock dependency analysis.

a source (site of user input) and a sink (dangerous function call or operation) using global, interprocedural taint propagation analysis.
2. Control Flow: detects potentially dangerous sequences of operations by analyzing control flow paths in a program.
3. Semantic: uses specialized logic to detect several potentially dangerous uses of functions and APIs at the intra-procedural level.
4. Structural: identifies violations of secure programming practices and techniques in the structure or definition of the program.
5. Configuration: finds mistakes, weaknesses, and policy violations in an application's deployment configuration files.
6. Buffer: detects buffer overflow vulnerabilities that involve writing or reading more data than a buffer can hold. It uses limited inter-procedural analysis to determine whether or not there is a condition that causes the buffer to overflow. If all execution paths to a buffer lead to a buffer overflow, buffer overflow vulnerability is reported along with the variables that could cause the overflow. If some, but not all, execution paths to a buffer lead to a buffer overflow and the value of the variable causing the buffer overflow is tainted (usercontrolled), then it will be reported as well and the dataflow trace showing how the variable is tainted will be displayed. The analyzer checks both stack-allocated and heap-allocated buffers [Fortify, 2012d].

The analyzers receive two inputs: rules and source code. Rules provide the information of the elements in the source code that may result in security vulnerabilities or are otherwise unsafe, which is necessary to perform the corresponding type of analysis. Fortify supports analyzing source code written in ASP.NET, VB.NET, C# (.NET), C/C++, Classic ASP (with VBScript), COBOL, CFML, HTML, Java, JavaScript/AJAX, JSP, PHP, PL/SQL, Python, T-SQL, Visual Basic, VBScript, ActionScript/MXML, XML and ABAP/4 [Fortify, 2012a].

Fortify offers a premium subscription service for Rulepacks, which are collections of rules developed by Fortify as well as third parties. These rulepacks are updated periodically and the Fortify SCA can be configured to automatically update its rulepacks whenever newer versions are available. Additionally, Fortify SCA is bundled with a Custom Rules Editor tool that allows creation of custom rules for all of the analyzers.

For structural analysis, the structural analyzer provides an AST (called structural tree) and a query language (structural tree query language) to perform complex matches against the structural tree. Custom rules are created by utilizing the query language to match forbidden constructs.

The control flow of analyzed code is modeled as a state machine by the control flow analyzer. Custom rules can be written by defining the transition rules and the error states.

To utilize Fortify SCA's taint propagation analyzer, custom rules can be defined by specifying a set of dataflow rules:

1. Dataflow Source Rule: to identify the points at which tainted data enters a program.
2. Dataflow Sink Rule: to identify points in a program that tainted data must not reach.
3. Dataflow Passthrough Rule: to describe how functions and methods propagate taint from their input to output.

4. Dataflow Entrypoint Rule: to describe program points that introduce tainted data to a program.
5. Dataflow Cleanse Rule: to describe validation logic and other actions that render tainted data either partially or completely cleansed.

Custom rules can also be written for checking configuration (Java properties files) and XML files by specifying patterns to find in the files [Fortify, 2012c].

Fortify is written in Java and runs on Linux, Windows, Mac OSX, Solaris, HP-UX, AIX and FreeBSD [Fortify, 2012b]. It is distributed under a proprietary commercial license. During our evaluation, a copy of Fortify version 3.5 with a special license for academic purposes was used.

3.1.2 Parasoft Jtest

Parasoft Jtest (Jtest) is an integrated solution for automating a broad range of practices proven to improve development team productivity and software quality, facilitating static analysis, peer code review process automation, unit testing and runtime error detection [Parasoft, 2012].

Jtest analyzes Java source code for violations of specified rules. Jtest has two types of rules:

1. Pattern-based: does local (only on a single source file) structural analysis of the code to find specific patterns that are encoded in the rule and triggers when the code matches the pattern.
2. Flow-based (named BugDetective): utilizes predefined data flow analyses to detect more complex bugs.

It is possible to create custom pattern-based rules via the provided RuleWizard tool. The rules can be created graphically (by creating a flow-chart-like representation of the rule) or automatically (by providing code that demonstrates a sample rule violation). The flow-based rules are customizable by setting the available parameters but it is not currently possible to create new flow-based rules [Parasoft, 2011].

Jtest is written in Java and runs on Windows, Linux, Solaris and Mac OSX. It is distributed under a proprietary commercial license. During our evaluation, a trial version of Jtest 9.2.3 was used.

3.1.3 Checkmarx CxSuite

Checkmarx CxSuite (CxSuite) is a source code analysis solution designed for identifying, tracking and fixing technical and logical security flaws from the source code, which provides a high degree of flexibility and configurability by supporting a wide range of vulnerability categories, OS platforms, programming languages and frameworks [Checkmarx, 2012a].

Checkmarx CxSuite analyzes program's source code to build an AST and a flow graph, which are then stored in a queryable structure. An extensive list of preconfigured queries for known security vulnerabilities are provided. Furthermore, it is possible to create additional queries (which are specified in a C-like language) using the CxSuite Auditor tool (CxAudit). The supported programming languages of the input are Java, JSP, JavaScript, VBScript, C/C++, C#, VB6, VB.NET, ASP(.NET), PHP, Apex and Ruby [Checkmarx, 2012b].

While not explicitly stated in the user manual, CxSuite is found to be capable to do data flow analysis as well. In particular, it can do an influence analysis of the data, as evident from our inspection of the available queries in the CxAudit. Furthermore, by testing the tool with the [Ware, 2008] test suite, it is deduced that CxSuite's analyses are global (interprocedural and interfile).

CxSuite is written in C# and runs on any Windows system with .NET Framework 2.0. It is distributed under a proprietary commercial license. A trial version of CxEnterprise 6.20 was used during our evaluation.

3.1.4 Other Commercial Tools

Several other commercial tools are also included in [Feiman & MacDonald, 2010]. They are, however, not reviewed in this study due to various reasons. A brief description of each tool and the particular reasons of their non-inclusion are provided in this section.

3.1.4.1 IBM Rational AppScan Source Edition

IBM Rational AppScan Source Edition (AppScan) is a static analysis security testing solution that enables users to identify vulnerabilities within the source code, review data and call flows, and identify the threat exposure of each of the applications [IBM, 2012].

AppScan is among the leading commercial products according to [Feiman & MacDonald, 2010]. Unfortunately, there is no evaluation version of the Source Edition available.

3.1.4.2 Veracode Source Code Security Analyzer

Veracode's source code security analyzer (Veracode) performs both dynamic (automated penetration test) and static (automated code review) code analysis and finds security vulnerabilities that include malicious code as well as the absence of functionality that may lead to security breaches [Veracode, 2012].

Veracode is offered in the form of software as a service and the trial version only analyzes code to find XSS and SQL injection vulnerabilities.

3.1.4.3 Coverity Static Analysis

Coverity Static Analysis (Coverity) helps developers find hard-to-spot, yet potentially crash-causing defects early in the software development life-cycle, reducing the cost, time, and risk of software errors [Coverity, 2012].

Coverity offers an evaluation version but limited to eligible business and academic institutions participating in the Coverity Academic Program only.

3.1.4.4 Klocwork

Klocwork Truepath, the static analysis engine that powers the family of Klocwork's tools, identifies critical security and reliability issues through a sophisticated whole program analysis of C/C++, Java and C# code [Klocwork, 2012].

A stripped down evaluation version of Klockwork (Solo) is available. However, it focuses only on Android and web applications.

3.1.4.5 GrammaTech CodeSonar

CodeSonar identifies programming bugs that can result in system crashes, memory corruption, and other serious problems. CodeSonar currently analyzes C/C++ code only and thus is not considered for review [GrammaTech, 2012].

3.1.4.6 Armorize CodeSecure

CodeSecure is a static source code analysis platform that leverages third generation software verification technologies to identify web application vulnerabilities throughout development [Armorize, 2012].

CodeSecure is excluded from review as it focuses only on web applications.

3.2 Open Source Tools

3.2.1 PMD

PMD scans Java source code and looks for potential problems like possible bugs, dead code, suboptimal code, overcomplicated expressions and duplicate code (as copied/pasted code can also mean copied/pasted bugs) [PMD, 2012b].

PMD works by checking the source code against a set of rules. The source code is first parsed using a JavaCC-generated parser, which creates an AST. PMD then hands the AST off to the symbol table layer, which builds scopes, finds declarations, and find usages. Additionally, PMD also has a data flow analysis layer that is able to build control and data flow graphs when the rules need data flow analysis. When evaluating the rule, the AST is traversed, consulting the symbol table and data flow information whenever necessary. PMD allows creation of new rules using either XPath queries or Java code. It is also possible to extend the analysis capabilities of PMD as it is open source software. While mainly intended to scan Java codes, PMD supports JavaScript, XML, XSL and JSP as well [PMD, 2012a].

PMD is written in Java and can be run on any platform with JRE 1.5 or later. It is distributed under the BSD-style license and version 5.0.0 was used during our evaluation.

3.2.2 FindBugs

FindBugs is a program to find bugs in Java programs. It looks for instances of *bug patterns* (code instances that are likely to be errors) [Hovemeyer & Pugh, 2012].

FindBugs uses static analysis to inspect Java bytecode for occurrences of the bug patterns, utilizing BCEL bytecode framework. Several approaches are employed by FindBugs to detect the bug patterns: bytecode scanning, control flow and data flow analysis. Various default bug patterns detectors are provided and the FindBugs plugin architecture enables the user to add custom detectors. As FindBugs is open source, it is also possible to extend its analysis capabilities [FindBugs, 2012; Hovemeyer, 2004].

FindBugs is written in Java and is platform independent. It is known to run on GNU/Linux, Windows and MacOS X platforms with JRE 1.5 or later. FindBugs is distributed under LGPL. The evaluation was done using FindBugs version 2.0.1.

3.2.3 JLint

Jlint checks Java code for bugs, inconsistencies and synchronization problems by doing data flow analysis and building the lock graph.

Jlint works by performing semantic verification on the Java class files. Jlint performs local and global data flow analyses, calculating possible values of local variables and catching redundant and suspicious calculations. Jlint utilizes global method invocation analysis to detect invocations of methods with possible null value of its parameter and the usage of the parameter in the method's body without checking for null. It also creates lock dependency graph for class dependencies and uses this graph to detect situations where deadlock might occur during multi-threaded program execution. Furthermore, Jlint is also able to detect possible race condition problems (concurrent access to the same variables by different threads).

The Jlint package also includes another tool, AntiC, which performs syntactic verification on C/C++ or Java source code. AntiC uses a hand-written scanner and a simple top-down parser to detect possible issues such as suspicious use of operators priorities, absence of break in switch code and wrong assumptions about constructions bodies [Knizhnik & Artho, 2011].

Both Jlint and AntiC do not currently have any customization features. However, it is still possible to extend the tools as they are distributed under GPL.

Jlint is written in C++ and does not use operating system dependent code in particular and therefore should be compilable and runnable on any system with a C++ compiler. A compiled version is available for Windows platform. Jlint version 3.1.2 was used during our evaluation.

3.3 Discussion

All the modern static code analysis tools reviewed in this chapter perform all the basic analyses–structural, control and data flow analysis–with varying amounts of built-in knowledge. The details of how the analyses are performed could not be obtained and compared for the commercial tools due to prior agreements. Furthermore, as this work mainly focuses on extending the tools, such detailed observation was not made for the open source tools as well.

In general, commercial products appear to be more polished than their open source counterparts, with better user interfaces and various IDE plug-ins, more built-in knowledge and supported programming languages, and more thorough documentation. However, due to their proprietary license, the extensibility of commercial products are also severely more limited than the open-source ones. The commercial tools provide the possibility to extend the built-in knowledge by creating custom rules but none of them enable the user to create custom analysis modules. As intermediate results are also not exposed, this might introduce a significant challenge when we try to add new knowledge from the knowledge catalogs.

The open source products offer more flexibility in customization. Among the analyzed open source tools, FindBugs is clearly the most developed. It offers a solid number of built-in rules and flexible customizable data flow analysis, which is not found in any other tools including the commercial ones. FindBugs, however, operates in bytecode. While this allows the tool to do the analysis more simply, some information might also be lost during the translation (*see* [Logozzo & Fähndrich, 2008] for more discussion about this). PMD is also a relatively mature product and operates in source code instead. It also provides an easy way of creating new rules that only use structural analysis using XPath expression language. Jlint has relatively more limited capabilities and the development has been slow. It also does not offer any out of the box customization features and requires changes to be performed directly on the code, which makes it not desirable for our purpose.

CHAPTER 4

Analysis of AFSCM's Rules

In this chapter, an analysis of the rule set in the AFSCM guidelines [AFSCM, 2012] is presented. In the analysis, references to the elements of the Java grammar are typeset as `terminal` and NonTerminal as described in Appendix B.

4.1 Overview

With respect to McGraw's classification of knowledge catalogs [McGraw, 2006], the AFSCM guidelines belong more appropriately into rules instead of guidelines. As rules are typically concrete enough, most of them are directly implementable into static analysis tools. That being said, not all the rules in the AFSCM guidelines can be implemented with static code analysis tools. For example, there are rules that deal with the specification documents of the applet (*e.g.* Rule 2 and 5), which are beyond the scope of code analysis.

Table 4.1 summarizes the analysis of AFSCM rules. It provides two main types of information: the level of automation possible and the difficulty of checking each rule.

Table 4.1: Overview of AFSCM rules

Rule	Description	Auto	Level	Remarks
1	Separate interface and data management	1	3	approximate, possible high false positive/low true negative
2	Define roles of available interfaces	0	—	not in code
3	Do not use shareable interfaces in applications with different SDs	2^+	1	EI: applications' SDs, *see also* rule 17
4	Do not put basic applications in packages containing certified applets	2^+	0	EI: application type, list of (packages with) certified applets
5	Define all possible entry points	0	—	not in code
6	Prevent denial of service			
6a	Do not instantiate objects outside `install()` and constructor	2	2	—
6b	Limit data heap resources usage	1^+	1^+	EI: limitation specification
6c	Limit telecom file system resources usage	1^+	1^+	EI: limitation specification
6d	Do not use infinite loop	1	3	possible false positive/negative
7a	Remove dead code	1	3	possible false negative

Continued on next page . . .

Table 4.1 – *Continued from previous page*

Rule	Description	Auto	Level	Remarks
7b	Remove debug information	1	1	approximate, possible false positive/negative
8	Ensure that file import versions are compatible with the target cards	2^+	0	EI: target card's capabilities
9	Use an incremented major version for the basic application with the same AID as an already-verified basic application with different exported methods signatures	2^+	1	EI: version numbers, previous version of the source file
10	Verify basic application bytecode with the tool from Oracle JCDK 3.0.3 or higher	2	0	—
11	Verify the cardlet installation parameters	×	—	checkable only after cardlet installation
12	Register applet instance at late as possible	2	2	—
13	Initialize menu entries in install	2	2	—
14	Implement uninstall method in Java Card 2.2.1 applets	2^+	1	EI: Java Card version
15	Ensure that a cardlet is not in run mode and not selected on a channel before deleted	×	—	checkable only during run time
16	Delete all created files in the deletion phase	1	3	possible false negative/positive
17	Restrict the use of the Javacard API			
17a	Do not use `java.rmi`	2	1	—
17b	Limit the use of `APDU` (*cf.* `process(APDU)`)	2^+	1^+	*see* rule 17c
17c	Limit the use of `Applet.process()`	2^+	1^+	EI: limitation specification
17d	Limit the use of `Applet.getShareableInterfaceObject()`	2^+	1^+	EI: limitation specification
17e	Limit the use of `JCSystem.lookupAID()`	2^+	1^+	EI: limitation specification
17f	Limit the use of `JCSystem.abortTransaction()`	2^+	1^+	EI: limitation specification
17g	Limit the use of `JCSystem.getPreviousContextAID()`	2^+	1^+	EI: limitation specification
17h	Limit the use of `JCSystem.getAppletShareableInterfaceObject()`	2^+	1^+	EI: limitation specification
17i	Limit the use of `MultiSelectable`	2^+	1^+	EI: limitation specification
17j	Do not use `OwnerPIN.setValidatedFlag(boolean)` in basic applications	2^+	1	EI: application type
17k	Limit the use of `OwnerPIN.reset()`	2^+	1^+	EI: limitation specification
17l	Limit the use of `OwnerPIN.update(byte[], short, byte)`	2^+	1^+	EI: limitation specification
17m	Limit the use of `OwnerPIN.resetAndUnblock()`	2^+	1^+	EI: limitation specification
17n	Limit the use of `Shareable`	2^+	1^+	EI: limitation specification
17o	Do not use `javacard.framework.service`	2	1	—
18	Restrict the use of the GlobalPlatform API	2^+	1	EI: application type
19	Restrict the use of the UICC API			
19a	Limit the use of `uicc.access.FileView`	2^+	1^+	EI: limitation specification
19b	Limit the use of `uicc.access.UICCSystem`	2^+	1^+	EI: limitation specification
19c	Limit the use of `uicc.access.bertlvfile`	2^+	1^+	EI: limitation specification
19d	Limit the use of `uicc.access.fileadministration`	2	2	—
19e	Limit the use of `uicc.toolkit`	2^+	1^+	defined in rule 20 and 21
20	Restrict the use of the SIM Toolkit commands	2^+	1^+	EI: limitation specification
21	Restrict the use of the SIM Toolkit events			
21a	Do not use `EVENT_MO_SHORT_MESSAGE_CONTROL_BY_NAA` in service provider application	2^+	1	EI: application type
21b	Do not use `EVENT_CALL_CONTROL_BY_NAA` in service provider application	2^+	1	EI: application type
21c	Do not use `EVENT_EVENT_DOWNLOAD_MT_CALL` in service provider application	2^+	1	EI: application type

Continued on next page . . .

Table 4.1 – *Continued from previous page*

Rule	Description	Auto Level		Remarks
21d	Do not use `EVENT_EVENT_DOWNLOAD_CALL_DISCONNECTED` in service provider application	2^+	1	EI: application type
21e	Do not use `EVENT_EVENT_DOWNLOAD_LOCATION_STATUS` in service provider application	2^+	1	EI: application type
21f	Do not use `EVENT_EVENT_DOWNLOAD_USER_ACTIVITY` in service provider application	2^+	1	EI: application type
21g	Limit the use `EVENT_EVENT_DOWNLOAD_BROWSER_TERMINATION`	2^+	1^+	EI: limitation specification
21h	Limit the use `EVENT_EVENT_DOWNLOAD_LOCAL_CONNECTION`	2^+	1^+	EI: limitation specification
21i	Limit the use `EVENT_EVENT_DOWNLOAD_BROWSING_STATUS`	2^+	1^+	EI: limitation specification
22	Do not use SIM API	2	1	—
23	Do not store applet object references in static fields	2	3	—
24	Do not rely exclusively on the object deletion feature	1	1	assist only
25	Do not use the scratch buffer for exchanges between applets	1	3	possible false negative/positive
26	Do not allocate arrays with dynamically calculated size	2	1	—
27	Allocate `FileView` objects during installation	2	2	subsumed by rule 30
28	Do not create file except during installation	2	2	—
29	Do not use recursion	2	2	—
30	Allocate all objects in installation phase or use singleton	2	2	—
31	Limit handset resource usage	1^+	1	EI: limitation specifications, *see* rule 20
32	Do not access or create files except application and other specifically allowed files			*see* rule 33
33	Restrict access to files, except: `3F00`, `2FE2`^{r}, `7F10/6F3A`^{rwu}, `7F10/6F3C`^{ru}, `7F10/6F44`^{ru}, `7F10/5F3A/4F30`^{ru}, `7F10/5F3A/4F3A`^{rwu}, `7F10/5F3A/4F09`^{rwu}, `7F10/5F3A/4F40`^{rwu}, `7F10/5F3A/4F60`^{rwu}, `7F10/5F3A/4F61`^{rwu}, `7F10/5F3A/4F50`^{rwu}, `7F10/5F3A/4F10`^{rwu}, `7F10/5F3A/4F22`^{rwu}, `7F10/5F3A/4F23`^{rwu}, `7F10/5F3A/4F24`^{rwu}, `7F20/6F07`^{r}, `7F10/6F14`^{r}, `7F10/6F46`^{r}, `7FFF/6F07`^{r}, `7FFF/6F3C`^{ru}, `7FFF/6F46`^{r}, `7FFF/6F80`^{ru}, `7FFF/6F81`^{ru}, `7FFF/6F82`^{ru}, `7FFF/6F83`^{ru}, `7FFF/6FCE`^{ru}, `7FFF/6FD0`^{ru}, `7FFF/6FD3`^{ru}	2^+	3	superscripts: r – read, w – write, u – under user control; *see also* rule 32
34	Use `ToolkitRegistry.registerFileEvent` on application-specific files only	2^+	3	EI: application-specific files list
35a	Create only a determined number of files	2	3	—
35b	Create files only in the installation phase	2	2	—
36	Create files only in ADF	2	3	—
37	Access ADFs and files using determined identifiers only	2	3	—
38	Use constants for AID values (except for application instance AID)	2	3	—
39	Do not access files controlled by the MNO	2^+	3	EI: files controlled by the MNO
40	Register file update events only on accessible files	2^+	3	EI: list of accessible files
41	Verify content written in phone book files	1	3	assist only
42	Do not resize files	2	1	—
43	Use exceptions to handle errors and exceptional situations	1	1	approximate, possible false negative/positive
44	Catch all exceptions in library code	1	3	—
45	Do not explicitly throw runtime exceptions	2	1	—
46	Throw and catch specific exception types only	2	1	—
47	Do not define application-specific exception types	2	1	—
48	Do not use platform specific APIs or libraries outside the scope of the platform certification TOE	2^+	1	EI: list of APIs in the scope of the platform certification TOE

Continued on next page . . .

Table 4.1 – *Continued from previous page*

Rule	Description	Auto	Level	Remarks
49	Do not use `ISOException(REPLY_BUSY)` for concluding event processing	2	1	—
50	Do not use hidden channels	1	1	partial, possible false positive/negative
51	Do not use objects that implement system interfaces shared by another application	1	3	—
52	Register events and initialize STK menu at the end of initialization phase	2	2	—
53	Do not use Java Card RMI	2	1	already covered by Rule 17
54	Restrict GP Privileges of basic application	x	—	not in code
55	Do not assign an access domain to a basic application that gives more right than needed	x	—	not in code
56	Always include a `default` case on `switch` statements	2	1	—
57	Do not use `int` type	2	0	—
58	Only accept commands and return status words that are valid according to ISO7816-4	2	3	—
59	Mask low-order CLA bits	2	3	—
60	Do not register call and SMS control events	2	1	—
61	Do not register proprietary events	2	1	—
62	Do not use `ViewHandler.compareValue`, `ViewHandler.copy`, `ViewHandler.copyValue` and `EditHandler.appendArray`	2	1	—
63	Protect all accesses to handlers by an exception handler	2	1	—
64	Declare the attribute of classes, fields and methods as privately as possible and final when possible	2	3	—
65	Name all constants and declare them as static final fields	2	3	—
66	Use identifier in function calls parameters	2	1	—

Rule: the corresponding rule number in the AFSCM guidelines

Description: a short description of the rule

Auto: the automation level that can be achieved for the rule
- 0: no automation is possible, needs manual review
- 1: semi-manual, manual review needed but some aid can be given
- 2: automated
- +: extra information needs to be provided for the analysis
- x: requires analysis other than static code analysis

Level: approximate difficulty of analysis
- 0: simple (pattern) matching or lexical analysis
- 1: syntactic analysis
- 2: control flow analysis
- 3: data flow or other more complex analysis
- +: maybe higher depending on the (extra) specification

Remarks: remarks about the rule
- EI: the extra information needed to be able to check the rule

There are nine main groups of AFSCM rules:

1. The first one is the management rules, which provides recommendations regarding the management issues such as documentation and applet lifecycle management. Being related to how things should be managed, many rules cannot be automatically checked from the source code. However, we can often provide information to aid human reviewers. Some of the rules deal with operations during runtime (after the applets are installed) and thus cannot be checked using static code analysis.
2. The second group are the rules concerning the usage of APIs and SIM Toolkit commands and events. Such uses can typically be found using syntactic analysis and thus all the rules in this section can be automatically checked. The rules in this group are the easiest to check compared to the other groups.
3. The third one is about memory management. The rules in this group are related mainly to object allocations, restricting their use according to specific conditions. While these are generally checkable, many of them require following the control and data flow, making the rules more difficult to check than those of the second group.
4. The fourth is about handset resources. While the rules are in the form of restrictions similar to the second group, they are dependent on limitations that are not specified in the guidelines and thus might be more difficult to check.
5. The fifth group deals with system management, which is mostly about file management. Dealing with files in JavaCard is not an easy task to do. This is also reflected in the difficulty of checking the rules of this group.
6. The sixth group contains rules about exception and error management. Simple syntactic analysis is enough in some cases but more advanced analyses are required in other cases.
7. The seventh one deals with interoperability that disallow the use of platform specific APIs outside the scope of TOE. Similarly to the second group, it is possible to check the rule using syntactic analysis. However, for this one, extra information about the TOE needs to be supplied to the analyzer.
8. The eighth rule group is about the interactions with external entities. Most of the rules can be checked automatically although there are rules that can only be checked partially.
9. Finally, the ninth group contains miscellaneous rules about development. Due to the variety of the rules, the difficulties of checking such rules also vary.

A discussion of each rule can be found in the following section, which is then followed by a discussion about the tools capabilities with respect to the rules.

4.2 Analysis of the Rules

The analyses presented in this section are grouped in subsections according to the AFSCM guidelines.

4.2.1 Management Rules

The rules in this section provide a number of recommendations regarding the design, the usage of data, resources and sensitive functions by the application, and the development environment. Furthermore, several recommended practices on how the application on the card should be managed, including which GlobalPlatform commands to use are also described.

Rule 1 *Interface management and Data management must be clearly separated.*

SEMI-MANUAL Separation between the handling of commands and the handling of content is a design decision that is not directly observable from the code. However, if this rule is satisfied, we should observe that there will be no operation using the data part of the APDU buffer in the `process()` except for method call parameters. Using this heuristic, we can aid the reviewer by saying that the rule is satisfied when such condition is satisfied. Otherwise, the reviewer has to check manually whether or not the usage affects clear interface/data separation. As the buffer might be copied to other variables during the process, we need to keep track of this and thus data flow analysis is necessary.

Rule 2 *The interactions between different interfaces must be clearly defined.*

MANUAL Defining the roles of different interacting interfaces is done at the design level and, in general, cannot be observed in retrospect from the resulting implementation.

Rule 3 *The shareable interface is forbidden for applications in different SD.*

AUTOMATIC Security domain is a property that is not defined in the code. Therefore, in order to be able to check this rule automatically, we need to supply this information to the analyzer. Once this information is available, we can use syntactic analysis to check whether or not the shareable interface is used. In particular, in order to use the shareable interface, an applet must extend the interface `javacard.framework.Shareable`. This is generally observable from the AST.

Rule 4 *Package design: A basic application shall never be in the same package as any certified applet.*

AUTOMATIC This rule is easily checkable utilizing syntactic analysis if the information whether or not the applet is a basic application and the list of package with certified applet are supplied.

Rule 5 *Entry points: All the different entry points shall be clearly defined at, for instance: APDU level, Toolkit level, OTA script level, Personalization level, Shared methods level.*

MANUAL This rule is about documenting the architectural design of the application.

Rule 6 *Denial of Service: Developers must avoid practices leading any denial of service as, for instance:*

(a) Banning any instantiation other than `install()` or applet's constructor,

(b) Limitation of use of data heap resources,
(c) Limitation of use of telecom file system resources,
(d) Banning infinite loop.

SEMI-MANUAL There are various practices that can lead to denial of service, with all kinds of mechanism and varying degrees of complexity. It is, hence, impossible to statically check for all of those practices completely. The given good practices examples are checkable to some degree. It is possible to check instantiations that are not in `install()` or applet's constructor by syntactic analysis supplemented by control flow analysis as the instantiations might be done in other methods that are called by `install()` or the constructor. Limiting the use of resources cannot be done without clear specification of the limitations. One possible approach to assist human reviewers is to list all use of those resources. Detecting the presence of (potential) infinite loops is a relatively hard problem, depending on the precision desired. In principle, we need to check whether or not the logical expression that controls the termination of the loop always evaluates to `false` (or the other way around, depending on the form of the loop). This logical expression might involve values from other variables and data flow analysis might be necessary in such case.

Rule 7 *(a) Dead code or (b) debug information must be deleted/removed from the code.*

SEMI-MANUAL Dead code can be detected by utilizing control and (path and context sensitive) data flow analysis. Detecting debug information in Java Card is rather difficult as there is no standard way of debugging. However, some heuristics can be done and the result can be presented to aid reviewers. For example, we can find any occurrence of the string `debug` in the code and list all calls to `ISOException` with variable parameter.

Rule 8 *Files imports: The file imports versions must correspond exactly to the versions (and method signatures) installed on the targeted cards. That excludes any custom component.*

AUTOMATIC This rule depends on the information about the versions (and method signatures) installed on the targeted cards and this information is not available in the code. Therefore, if we want to check this rule, this extra information must be provided to the analyzer. We can do a simple matching check when this information is available.

Rule 9 *Version number increment: A basic application with the same AID as an already verified basic application but with different exported methods signatures shall have an incremented major version.*

AUTOMATIC This rule is about versioning, which is not generally observable from the code. However, if we have the information about the version numbers and the source code of previously verified versions, we can automatically compare them to check if this rule is satisfied.

Rule 10 *ByteCode verification: In order to be protected against attacks on card resources, a basic application must be strictly in accordance with the Java Card specifications and must successfully pass the latest ByteCode verification tool from Oracle in JCDK version 3.0.3 or higher.*

AUTOMATIC This rule can be easily checked by running the respective ByteCode verification tool.

Rule 11 *The following tests methods must be used so as to verify the « cardlet installation parameters »:*

(a) *For a pre-installed applet, check the AID instance presence using the "Get status" command. The usage of proprietary commands is forbidden for this feature.*
(b) *For other applets, check the Status code returned from the install for install command.*

MANUAL This rule can only be enforced after the cardlet is installed.

Rule 12 *Applet instance registration should occur as late as possible.*

AUTOMATIC This rule can be checked by ensuring that call to `register()` is at the end of `install()` method or strictly before toolkit event registrations (calls to `ToolkitRegistry`'s methods), with the toolkit event registrations being the last operations in the `install()`. The checks can be done using syntactic analysis supplemented with interprocedural control flow analysis.

Rule 13 *Menu entries should be initialized in install.*

AUTOMATIC This rule is checkable by examining whether or not the call to `ToolkitRegistry.initMenuEntry()` is present in applet's `install()` method using syntactic and interprocedural control flow analysis.

Rule 14 *Implementation of the uninstall method in Java Card 2.2.1 applets is mandatory.*

AUTOMATIC This rule can be easily checked by detecting the presence of a non-empty `uninstall` method in the code using syntactic analysis.

Rule 15 *Before the cardlet deletion stage, the cardlet shall not be in run mode or selected on a channel in order to avoid memory saturation.*

MANUAL This rule is only checkable during runtime.

Rule 16 *All created files must be deleted in the application deletion phase.*

SEMI-MANUAL While this rule might not be checkable if the files are created dynamically (*i.e.* during runtime), it is possible to enforce the rule if there are only a determined number of files and they are created only in the installation phase (this restriction is enforced by Rule 35) by observing if there is a matching deletion command for each file creation. We can also let a reviewer check this and assist by listing all file creations and deletions in the code.

4.2.2 Usage of APIs and SIM Toolkit Commands and Events

The rules in this section impose restrictions to the use of APIs, commands and events.

Rule 17 *The cardlet must obey the restrictions in Table 4.2 on the use of the Java Card API.*

AUTOMATIC The rules that specifically forbid the usage of certain methods of certain classes/interfaces in certain packages of the Java Card API are easily checkable using simple syntactic analysis. However, checking the *limited* restrictions will require extra information on what kinds of limitations are in place to be supplied to the analyzer. The same limitation applies for the rule that is only applicable for the basic applications. Once this extra information is provided, the rules can be similarly checked using syntactic analysis.

Rule 18 *The cardlet must obey the restrictions in Table 4.3 on the use of the GlobalPlatform API.*

AUTOMATIC This rule is checkable using syntactic analysis, but requires extra information

Table 4.2: AFSCM: Restrictions on the use of the Java Card API

	Packages	Class/Interfaces	Method	Rules
(a)	`java.rmi`	ALL	ALL	Forbidden
(b)	`javacard.framework`	`APDU`	ALL	*cf.* `process(APDU)`
(c)		`Applet`	`Process()`	Limited
(d)			`getSharableInterfaceObject`	Limited
(e)		`JCSystem`	`lookupAID()`	Limited
(f)			`abortTransaction()`	Limited
(g)			`getPreviousContextAID()`	Limited
(h)			`getAppletShareableInterfaceObject()`	Limited
(i)		`MultiSelectable`		Limited
(j)		`OwnerPIN`	`setValidatedFlag(boolean)`	Forbidden for basic applications
(k)			`reset()`	Limited
(l)			`update(byte[], short, byte)`	Limited
(m)			`resetAndUnblock()`	Limited
(n)		`Shareable`	ALL	Limited
(o)	`javacard.framework.service`			Forbidden

of whether or not the cardlet is a basic application.

Rule 19 *The cardlet must obey the restrictions in Table 4.4 on the use of the UICC API.*

AUTOMATIC This rule specifies *limited* use, which is a design decision that is not directly observable from the code (in the context of AFSCM, these limitations are documented in the Functional Specification and the Specific Declaration of Security). If the limitation specifications can be supplied to the analyzer, it is possible to identify the usage of certain methods of certain classes/interfaces in certain packages with simple syntactic analysis. Depending on the limitation, other types of analysis might be necessary. For example, to check 19d, we also need to use interprocedural control flow analysis to detect the use in related methods called during install and deletion. A semi-manual approach can also be taken by simply listing all the usage for further review.

Rule 20 *The cardlet must obey the restrictions in Table 4.5 on the use of the SIM (Application) Toolkit commands.*

Note: All other proactive commands are allowed.

AUTOMATIC Similarly to Rule 19, this rule is checkable using syntactic analysis but only after the *limited* usage clauses are specified. The semi-manual approach is also possible in this case.

Rule 21 *The cardlet must obey the restrictions in Table 4.6 on the use of the SIM (Application) Toolkit events.*

Note: All other proactive events[1] *are allowed.*

[1]The AFSCM guidelines erroneously refer to these events as proactive *commands*.

Table 4.3: AFSCM: Restrictions on the use of the GlobalPlatform API

	Packages	Class/Interfaces	Method	Rules
(a)	org.globalplatform.GPSystem	GPRegistryEntry	getRegistryEntry	Forbidden for basic applications
(b)			lockCard()	Forbidden for basic applications
(c)			setATRHistBytes	Forbidden for basic applications
(d)			setCardContentState	Forbidden for basic applications
(e)			terminateCard()	Forbidden for basic applications
(f)			deregisterService	Forbidden for basic applications
(g)		CVM	all except: – verify(), – getTriesRemaining(), – is*()	Forbidden for basic applications
(h)	org.globalplatform.contactless	CRELApplication	notifyCLEvent(...)	Forbidden for basic applications
(i)		CRSApplication	processCLRequest(...)	Forbidden for basic applications
(j)		GPCLSystem	getCardCLInfo(...)	Forbidden for basic applications
(k)			getGPCLRegistryEntry(...)	Forbidden for basic applications
(l)			getNextGPCLRegistryEntry(...)	Forbidden for basic applications
(m)			setCommunicationInterface(...)	Forbidden for basic applications
(n)			setVolatileProprietary(...)	Forbidden for basic applications

Table 4.4: AFSCM: Restrictions on the use of the UICC API

	Packages	Class/Interfaces	Method	Rules
(a)	uicc.access	FileView	ALL	Limited
(b)		UICCSystem	ALL	Limited
(c)	uicc.access.bertlvfile	ALL	ALL	Limited
(d)	uicc.access.fileadministration	ALL	ALL	Limited
(e)				It shall be used during installation and deletion stage in order to create/delete proprietary files used by the cardlet itself.
(f)	uicc.toolkit	ALL	ALL	See Table 4.5 and Table 4.6 for Commands and Events

Table 4.5: AFSCM: Restrictions on the use of the SIM Toolkit commands

	Proactive Commands	Usage
(a)	PRO_CMD_LAUNCH_BROWSER	Limited
(b)	PRO_CMD_PERFORM_CARD_APDU	Limited
(c)	PRO_CMD_POWER_OFF_CARD	Limited
(d)	PRO_CMD_PROVIDE_LOCAL_INFORMATION	Limited
(e)	PRO_CMD_RUN_AT_COMMAND	Limited
(f)	PRO_CMD_SEND_DATA	Limited
(g)	PRO_CMD_SEND_DTMF	Limited
(h)	PRO_CMD_SEND_SHORT_MESSAGE	Limited
(i)	PRO_CMD_SET_UP_CALL	Limited

Table 4.6: AFSCM: Restrictions on the use of the SIM Toolkit events

	Event	Usage
(a)	EVENT_MO_SHORT_MESSAGE_CONTROL_BY_NAA	Forbidden for Service Provider application
(b)	EVENT_CALL_CONTROL_BY_NAA	Forbidden for Service Provider application
(c)	EVENT_EVENT_DOWNLOAD_MT_CALL	Forbidden for Service Provider application
(d)	EVENT_EVENT_DOWNLOAD_CALL_DISCONNECTED	Forbidden for Service Provider application
(e)	EVENT_EVENT_DOWNLOAD_LOCATION_STATUS	Forbidden for Service Provider application
(f)	EVENT_EVENT_DOWNLOAD_USER_ACTIVITY	Forbidden for Service Provider application
(g)	EVENT_EVENT_DOWNLOAD_BROWSER_TERMINATION	Limited
(h)	EVENT_EVENT_DOWNLOAD_LOCAL_CONNECTION	Limited
(i)	EVENT_EVENT_DOWNLOAD_BROWSING_STATUS	Limited

AUTOMATIC Similar to previous rules (Rule 19 and 20). Here, additional information about whether or not the application is a Service Provider application is also needed.

Rule 22 *SIM API (TS 43.019) is obsolete and it is forbidden to use it.*

AUTOMATIC This rule can be checked by syntactically analyzing whether or not there is any reference to the SIM API in the code.

4.2.3 Memory Management

This section contains several rules that are related to preventing memory exhaustion by forbidding and limiting memory allocations.

Rule 23 *Applet object references must not be stored in static fields because when an application is deleted, memory can or cannot be properly released.*

AUTOMATIC This rule is checkable by inspecting if there is any assignment of applet object references to static fields using syntactic analysis. As the assigned reference might then be assigned to another field, we also need to keep track of this. This can be done using data flow analysis.

Rule 24 *Applications should not rely exclusively on the object deletion feature because:*

- *it may be time-consuming,*
- *there is no guarantee to free instantly the memory,*
- *and the result is not guaranteed on some cards.*

SEMI-MANUAL This rule includes the term *should not rely exclusively*, which indicates a design decision. It is, therefore, not checkable by only observing the code. However, we can assist reviewers by, for example, listing all object deletions. Furthermore, if Rule 30 is satisfied, this rule is rather irrelevant.

Rule 25 *Use of the scratch buffer is only authorized for local computations and is forbidden for exchanges between applets.*

SEMI-MANUAL Using data flow analysis, we can keep track the usage of the scratch buffer (accessible through `UICCPlatform.getTheVolatileByteArray`). If there are only write operations to the buffer and no read (or the other way around), it might indicate that the buffer is being used for exchanges between applets. This is, however, not always the case and thus will need to be checked by reviewer.

Rule 26 *Arrays must be allocated with a determined size.*

AUTOMATIC This rule can be checked using syntactic analysis by ensuring that the sizes supplied to all array allocations are constants.

Rule 27 *`FileView` objects must be allocated during application installation.*

AUTOMATIC `FileView` objects are retrieved by invoking the `getFileView()` method from the `UICCSystem` class. Using syntactic analysis along with interprocedural control flow analysis, it is possible to detect any calls to `getFileView()` that occur outside the applet's `install()` context.

Rule 28 *Files must be created only during installation phase.*

AUTOMATIC Files are created programmatically by invoking the `createFile()` method of `AdminFileView`. By checking the presence of `createFile()` invocations that are not in the applet's `install()` context, it is possible to check any violations to this rule. Syntactic and interprocedural control flow analysis are required to do so.

Rule 29 *Recursive code is forbidden.*

AUTOMATIC Recursion can be detected by inspecting the call graph of the program.

Rule 30 *All objects must be allocated in the installation phase or allocated with a singleton.*

AUTOMATIC This rule can be checked by ensuring that all ClassInstanceCreationExpression and ArrayCreationExpression as well as calls to `JCSystem.makeTransient...Array()` are in `install()` or any methods invoked by it using syntactic and interprocedural control flow analysis.

4.2.4 Handset Resources Management

The rule in this section concerns with the usage of handset resources.

Rule 31 *The cardlet must obey the restrictions in Table 4.7.*

SEMI-MANUAL This rule cannot be checked fully automatically because the restrictions are

Table 4.7: AFSCM: Cardlet's handset resources management restrictions

	Resources	Rules	Conditions
	Data		
(a)	Customer personal information access on the handset (PIM): Directory, agenda, notes, messages, sounds, pictures, ...	Limited access	Under user control
(b)	Capture interface access: sound record, pictures, localization, ...	Limited access	Under user control
	Communications		
(c)	Call interception (voice, visio, data, ..., etc.)	Limited access	Under user control
(d)	Set Up Call	Limited access	Under user control Fixed destination (Constant address not calculated – Tags information) Variable addresses are forbidden.
	Message Services		
(e)	Messages interception (SMS, MMS, e-mail, ..., etc.)	Limited access	Under user control
(f)	Messages sending (SMS, MMS, e-mail, ..., etc.)	Limited access	Under user control Fixed destination (Constant address not calculated – Tags information) Variable addresses are forbidden.
	Local Connectivity		
(g)	Local connection establishment (serial, IR, Bluetooth, WIFI, NFC, ...)	Limited access	Under user control
	Distant Connectivity		
(h)	Distant connection establishment (http, https, ...)	Limited access	Under user control Variable addresses are forbidden.
	Application Management		
(i)	Management or triggering of other applications	Limited access	
	Others		
(j)	Interrogation of capacities or configuration of the terminal	Limited access	Large Interrogation Forbidden: Interrogation clarifies resources used and strictly necessary for the application

not specified precisely. Identifying the use of mobile resources can be done using syntactic analysis. This rule overlaps with Rule 20.

4.2.5 System Management

This section contains rules related to system files access, file event and management.

Rule 32 *By default, access to files (reading, writing) and creation of files (except the application files) is forbidden.*

See Rule 33.

Rule 33 *Files authorized in restricted accesses (subjected to the MNO agreement) are specified in Table 4.8; access to all other files is forbidden!*

AUTOMATIC To check this rule, we need to specify the list of authorized files agreed by the MNO. As reading/writing file is a sequence of commands (getting the FileView object, select the EF and then read/write), we will need data flow analysis.

Rule 34 *The use of `ToolkitRegistry.registerFileEvent` method must be limited to specific application files.*

AUTOMATIC It is necessary to have the list of application files on which the method is allowed to be used in order to check this rule. Due to similar reason as rule 33, data flow analysis is necessary for this rule.

Rule 35 *Only a determined number of files must be created and only in the installation phase.*

AUTOMATIC This rule can be checked by ensuring that there is no call to `AdminFileView.createFile()` outside `install()`'s execution path and in the context of `install()`, the call to `createFile()`, if any, should not be in a loop or recursion construct. Syntactic analysis along with intra- and interprocedural control flow analysis are needed to do so.

Rule 36 *Files must only be created in ADF.*

AUTOMATIC There are 2 kinds of methods to get the `FileView` to be used in the `AdminFileView.createFile()` from `UICCSystem`: `getTheFileView()` and `getTheUICCView()`. The `getTheFileView()` retrieves a reference to a `FileView` object on an ADF file system and `getTheUICCView()` get a reference to a `FileView` object on the UICC file system. Using data flow analysis, we can track if the `createFile()` of the object obtained from `UICCSystem.getTheUICCView()` is used.

Rule 37 *ADFs and files must be accessed with determined identifiers.*

AUTOMATIC This rule essentially disallows scanning of files. We can check this by ensuring that the file identifier parameter of `select()` of the `FileView` is constant in the fashion of data flow analysis.

Rule 38 *Constants must be used for AID values (except application instance AID).*

AUTOMATIC With data flow analysis, it is possible to check if the byte array used for AID is constant. If the constants are declared as final fields, syntactic analysis is sufficient.

Rule 39 *Access to files controlled by the MNO is forbidden.*

This rule is subsumed by Rule 32 and 33.

Table 4.8: AFSCM: Files authorized in restricted accesses

	ID1	ID2	ID3	Nom	Limited access	Conditions
(a)	3F00			MF	–	
(b)	2FE2			ICCID	Read	
(c)	7F10			TELECOM	–	
(d)		6F3A		and	Read/Write	Under user control
(e)		6F3C		SMS	Read	Under user control
(f)		6F44		LND	Read	Under user control
(g)		5F3A		PUBLIC PHONE BOOK	–	Under user control
(h)			4F30	PBR	Read	Under user control
(i)			4F3A	and	Read/Write	Under user control
(j)			4F09	PBC	Read/Write	Under user control
(k)			4F40	ANR1	Read/Write	Under user control
(l)			4F60	SNE1	Read/Write	Under user control
(m)			4F61	SNE2	Read/Write	Under user control
(n)			4F50	EMAIL	Read/Write	Under user control
(o)			4F10	UID	Read/Write	Under user control
(p)			4F22	PSC	Read/Write	Under user control
(q)			4F23	CC	Read/Write	Under user control
(r)			4F24	PUID	Read/Write	Under user control
(s)	7F20			GSM	–	
(t)		6F07		IMSI	Read	
(u)		6F14		CPHS:ONS	Read	
(v)		6F46		SPN	Read	
(w)	7FFF			USIM		
(x)		6F07		IMSI	Read	
(y)		6F3C		SMS	Read	Under user control
(z)		6F46		SPN	Read	
(α)		6F80		ICI	Read	Under user control
(β)		6F81		OCI	Read	Under user control
(γ)		6F82		ICT	Read	Under user control
(δ)		6F83		OCT	Read	Under user control
(ϵ)		6FCE		MMSN	Read	Under user control
(ζ)		6FD0		MMSICP	Read	Under user control
(η)		6FD3		NIA	Read	Under user control

Rule 40 *Applications must only register for file update events on accessible files.*

This rule is an extension of Rule 32 and 33.

Rule 41 *The application should verify the content written in phone book files.*

SEMI-MANUAL The purpose of this rule is to prevent erroneous or malformed data to be written, which might result from, for example, data that contain control characters. There is no standard method to verify the content written in phone book files and thus

implementations may vary. We can assist the reviewer by detecting if there is any write operation to the phone book files.

Rule 42 *Resizing of files after their creation is forbidden.*

AUTOMATIC This rule can be checked by detecting the presence of calls to `FileAdminView.resize()` using syntactic analysis.

4.2.6 Exception and Error Management

The rules in this section deal with exception and error management.

Rule 43 *Exceptions should be used to handle errors and exceptional situations.*

SEMI-MANUAL Other ways to handle errors and exceptional situations are, for instance, using error statuses/flags and/or silently "correcting" the errors–any of which will not be reliably distinguishable from normal operations by examining the code. For this rule, we can check if exceptions are being used in the code using syntactic analysis. If there is no such use, it might indicate another way to handle errors is being used.

Rule 44 *All exceptions in library code must be caught.*

SEMI-MANUAL This rule can be enforced partially by checking whether or not all invocations of methods that might throw (checked) exceptions are in `try–catch` blocks using syntactic analysis. For unchecked exceptions, if all Blocks in the code are in `try–catch(RuntimeException)` or `try–catch(Exception)`, we can consider that the rule is satisfied.

Rule 45 *Explicit use of runtime exceptions is forbidden.*

AUTOMATIC This rule is checkable by syntactically analyzing the code, checking the presence of any reference in `ThrowStatement` to java.lang.RuntimeException and its subclasses.

Rule 46 *Only specific exception types must be thrown and caught.*

AUTOMATIC In other words, the base class `java.lang.Exception` should not be used. This can be checked using syntactic analysis.

Rule 47 *Application-specific exceptions must not be defined.*

AUTOMATIC Application-specific exceptions can be created by subclassing `java.lang.Throwable` or `java.lang.Exception`. By using syntactic analysis to detect any attempts to do so, this rule can be checked.

4.2.7 Robustness and Interoperability

Rule 48 *Development and use of platform specific APIs or libraries outside the scope of the platform certification TOE (Target Of Evaluation) is forbidden.*

AUTOMATIC The platform certification TOE information is not defined in the code. If this information is specified and supplied to the analyzer, it is possible to check this rule using syntactic analysis.

4.2.8 Interactions

The rules in this section are related to handset and external interactions.

Rule 49 *`ISOException (REPLY_BUSY)` shall not be used as a conclusion to event processing.*

AUTOMATIC This rule can be checked by observing the presence of `ISOException (REPLY_BUSY)` using syntactic analysis.

Rule 50 *Hidden channels are forbidden.*

SEMI-MANUAL Hidden channels are simply defined as "communication mechanisms that are non-standard" and thus the possibility is limitless. It is, therefore, impossible to check the presence of all such channels. Common hidden channels include usage of shared objects and public static fields. Both presences can be detected using syntactic analysis.

Rule 51 *Use of objects that implement system interfaces shared by another application is forbidden.*

SEMI-MANUAL It is generally hard to identify objects that implement system interfaces shared by another application. As this is also not very common, an easier approach will be to list all use of shareable interface and let the reviewer check if the shared objects implement system interfaces.

Rule 52 *Events registration and STK menu initialization must be done at the end of the installation phase.*

AUTOMATIC This rule can be checked by syntactic analysis complemented with interprocedural control flow analysis. Event registration is typically done using `ToolkitRegistry`'s `setEvent()` or `setEventList()` and menu initialization is done using `initMenuEntry()`.

Rule 53 *Use of Java Card RMI is forbidden.*

AUTOMATIC This rule can be checked using syntactic analysis. The usage of Java Card RMI is identifiable from the presence of the extensions from `java.rmi.Remote` in the case of the remote interfaces and `javacard.framework.service.CardRemoteObject` in the case of the remote objects.

4.2.9 Developments Rules

This section contains several rules related to applet developments.

Rule 54 *GP Privileges: a basic application must not have the following GlobalPlatform privileges:*

- *SD and associated privileges*
- *Card lock*
- *Card terminate*
- *Card reset*
- *CVM Management*
- *Global Delete*
- *Global lock*
- *Global registry*

– *Final application*

MANUAL GP privileges are defined outside the code.

Rule 55 *Telecom Applets: Either in 2G or 3G domains, a basic application shall never be assigned an access domain (for UICC file system or for any ADF, in normal access or administrative access) that gives more right than needed ("least privilege" principle).*

MANUAL Determining appropriate access domain is a design decision that is beyond the scope of code analysis.

Rule 56 *A `switch` statement must always include a `default` case.*

AUTOMATIC This rule can be easily checked using syntactic analysis. The rule is violated if there is no `default` SwitchLabel in any defined SwitchBlock in the code.

Rule 57 *Use of the `int` type is forbidden.*

AUTOMATIC This rule can be checked using syntactic analysis, ensuring that the PrimitiveType `int` tokens are not present in the code.

Rule 58 *A basic application must only accept commands and only return status words that are valid according to ISO7816-4 specification.*

AUTOMATIC This rule can be checked using syntactic analysis. In the analysis, we should look for conditional expressions which check that CLA is not `0xFF`, INS is even and its most significant nibble is not `0x6` or `0x9`, and the most significant byte of status word is in the range of `0x6X`–`0x9X`. In order to do so, first we need to identify the CLA and INS, both of which are obtained from the APDU buffer. These values might be stored in other variables and data flow analysis might be necessary to keep track of this.

Rule 59 *Low-order CLA bits should be masked.*

AUTOMATIC This can be checked by ensuring all expressions that use the CLA involve masking by `0xFC`. This can be done in every use or by rewriting the value once in the beginning. It is also possible that the CLA is stored in other variable after being masked and that that variable is used later on. Data flow analysis is required to track this.

Rule 60 *Registration to call and SMS control events is forbidden.*

AUTOMATIC There are two events related to this rule, *i.e.* `EVENT_CALL_CONTROL_BY_NAA` and `EVENT_MO_SHORT_MESSAGE_CONTROL_BY_NAA`. The usage of these events is already forbidden by Rule 21, which can be enforced by syntactic analysis.

Rule 61 *Registration to proprietary events is forbidden.*

AUTOMATIC This rule can be checked using syntactic analysis, examining the presence of non-standard events.

Rule 62 *In `ViewHandler` and `EditHandler`, only methods with separate tags, with the tags being determined, must be used.*

AUTOMATIC This rule practically means that there should be no calls to the methods `ViewHandler.compareValue()`, `ViewHandler.copy()`, `ViewHandler.copyValue()` and `EditHandler.appendArray()`, which is observable using syntactic analysis.

Rule 63 *The application should protect all accesses to handlers by an exception handler.*

AUTOMATIC This rule is checkable using syntactic analysis. By examining the syntax tree, it should be possible to detect any access to handlers that is not protected by a `try–catch` block.

Rule 64 *The application should properly declare the attributes of classes, fields, and methods.*

AUTOMATIC The rule requires all classes, fields and methods to be declared as privately as possible and final whenever possible. By globally analyze the use of all classes, fields and methods, we can determine the proper visibility (for instance, if a method is never used in any other classes, it should be private). Similarly, if there is no change made during the course of the program, we can set them to final. There are special-purpose tools that do exactly this, such as the UCDetector.

Rule 65 *All constants should be named and declared as static final fields.*

AUTOMATIC The first part of the rule can be checked using syntactic analysis. In particular, there should not be any Literals in the right hand side of assignments nor in any method calls' parameters. In order to ensure that all constants are declared as `static final` fields, a list of all constants in the code is needed. One approach to do this is by doing a data flow analysis, finding variables whose values are never changed throughout the program.

Rule 66 *An identifier must be used in parameter of the function calls.*

AUTOMATIC We can use syntactic analysis to check this rule. In particular, the Expression in the ArgumentList of MethodInvocation should be FieldAccess.

4.3 Rules–Tools Capabilities Mapping

In general, when we observe Table 4.1 presented earlier in Section 4.1, we can directly relate with the capabilities of the tools discussed in Chapter 3. Any rules with level 0, 1 and 2 should be able to be implemented with little to medium effort in any tools as all the necessary capabilities are available. The rules with level 3 mostly require custom data flow analysis and thus might be hard to implement in the commercial tools. In any cases, the effort needed to extend Jlint is higher as it does not provide any customization features.

CHAPTER 5

Analysis of Riscure Secure Application Programming Patterns

In this chapter, an analysis of the programming patterns for preventing Fault Injection described in the Riscure guidelines [Witteman, 2012] is presented. In the analysis, references to the elements of the Java grammar are typeset as `terminal` and NonTerminal as described in Appendix B.

5.1 Overview

Compared to the AFSCM guidelines, the Riscure guidelines is more abstract in nature, conforming to the definition of guidelines by McGraw [McGraw, 2006] as discussed in Chapter 2. Consequently, in order to adopt the knowledge in the guidelines, appropriate rules have to be derived. While this is quite straight-forward in some cases (*e.g.* FAULT.DEFAULTFAIL), the others are not (*e.g.* FAULT.DOUBLECHECK). More thorough discussions of each guideline are presented in subsequent sections. In general, it should be possible to incorporate the guidelines once they are derived into rules. However, further research is necessary in the area of deriving rules from the guidelines.

5.2 Basic Components

There are several classes of information that are being used across the guidelines. In order not to discuss the same things repeatedly, this information is described in this section to be referred later in the analysis.

Central to the discussion of Riscure guidelines are the notions of *sensitive data*, which are represented in *sensitive fields*, and *sensitive operations*, which are related to *sensitive methods* and *sensitive conditions*. Considering the CIA triad, data can be sensitive with respect to confidentiality, integrity and/or availability. For simplicity, these various kinds of sensitive data will be referred as *x*-sensitive data, with *x* being the properties of the data that should be preserved. For example, data whose confidentiality and integrity must be protected will be referred as confidentiality-and-integrity-sensitive data.

There are basically two ways of recognizing sensitive data and operations:

1. *Supplied information*: An easy but precise way to do this is by supplying extra information about sensitive code and operations. This can be done by, for example, adding annotations to the code. This approach is particularly good if the amount of sensitive data and/or operations is limited.
2. *Heuristics*: It is often possible to infer the sensitivity of data from their usage and the sensitivity of operations from the data involved. This approach is approximate and how good the result is will depend on the heuristic algorithms applied. Using this approach, less manual labor will be required, with the expense of accuracy and completeness.

Both approaches can be combined to create a good balance between the quality of result and the amount of work required.

5.2.1 Detecting Sensitive Data

As discussed earlier, in order to accurately differentiate sensitive data from the non-sensitive ones, we can annotate fields containing sensitive data or use any other ways of supplying extra information.

In general, it is difficult to use heuristics to identify sensitive data. However, there are some (classes of) data that are typically sensitive, for instance:

- Cryptographic keys are integrity-sensitive. Furthermore, secret keys are also confidentiality-sensitive.
- PIN objects are confidentiality-and-integrity-sensitive.
- State variables are in general integrity-sensitive.
- Data whose checksum are calculated are typically integrity-sensitive.

5.2.2 Detecting Sensitive Operations

Sensitive operations are operations which may result in the properties of the data that should be preserved against being violated. As confidentiality can only be violated by *read* operations and integrity by *update* operations, it follows directly that among the sensitive operations are reading confidentiality-sensitive data and updating integrity-sensitive data. The violation to the *availability* property can be seen as an exhaustion of resources that prevents access (read and/or write) to the availability-sensitive data. Recalling that the means of attack is fault injection, it is simply impossible to safeguard the availability of data programmatically and thus we can safely ignore the availability property in our discussion.

Therefore, for the heuristics of detecting sensitive operations, we have:

- For the fields containing integrity-sensitive data, any updating operations (*e.g.* assignments) to such fields are sensitive operations.
- For the fields containing confidentiality-sensitive data, operations that use the value of the fields are generally sensitive operations. In the case that the data being read (or the result of the operations involving the sensitive data) are then stored in another fields, these fields might become sensitive as well and proper care must be taken to track the information flow. This is possible, for example, in the fashion of taint analysis. While there are cases where operations can change the confidentiality property of the data (*e.g.* anonymization

functions), these cases are not easily distinguishable and thus it is better to err on the safe side and consider all operations using confidentiality-sensitive data as sensitive.

It is worth noting that although the above heuristics imply the requirement of knowing what kind of sensitive data we are dealing with, we can still have a safe approximation of sensitive operations even if the only information we know is whether or not some particular field contains sensitive data. This can be done by assuming that both confidentiality and integrity should be preserved. As with sensitive data, the supplied information approach can also be used to detect sensitive operations.

5.2.2.1 Detecting Sensitive Methods

Sensitive methods are methods that handle sensitive operations. In other words, if we have a sensitive operation in a method, this information should be propagated into all the callers of such method.

Theoretically, it might also be possible to have a method that does not have any sensitive operations but returns a value that is sensitive by specification. In this case, we will need to supply the information to the analyzer (*i.e.* the static code analysis tool) as the analyzer will not be able to derive this information by inspecting the code.

Note that we will also need to supply the sensitivity information of methods from external libraries.

5.2.2.2 Detecting Sensitive Conditions

There are several basic variants in which the sensitive conditions can manifest themselves in the code, as shown in listings 5.1 to 5.4. Here `guard` represents a logical expression that does the necessary authorization in order to run the sensitive operation (in `block_with_sensitive_op()`) and `fail()` represents a piece of code that will stop the current process. Note that it is possible to create more complex variants by combining these basic variants.

Listing 5.1: Sensitive Condition Variant 1

```
if (guard) {
  block_with_sensitive_op();
} else {
  fail();
}
```

Listing 5.2: Sensitive Condition Variant 2

```
if (!guard) {
  fail();
} else {
  block_with_sensitive_op();
}
```

Listing 5.3: Sensitive Condition Variant 3

```
if (guard) {
  block_with_sensitive_op();
}
```

Listing 5.4: Sensitive Condition Variant 4

```
if (!guard) {
  fail();
}
block_with_sensitive_op();
```

In order to match the code with the possible variants previously mentioned, there are several other things that we first need to detect, namely the `guard`, `block_with_sensitive_op()` and `fail()`.

Detecting the Guard

As it is used for sensitive conditions, a proper `guard` is typically a logical expression that involves integrity-sensitive data or sensitive method.

Detecting the Fail

In the context of smartcard, the common way to stop a process is by throwing an `ISOException` (except the "success" status codes, *e.g.* 9000 in ISO7816-compliant applets), which can be easily detected by syntactic analysis. Alternatively, in some worse cases (*e.g.* abnormal behavior is detected), the card can turn into mute state (*i.e.* running an infinite loop), which is detectable by examining the control flow graph.

Detecting Blocks with Sensitive Operations

Basically, `block_with_sensitive_op()` is a Block which contains one or more Statements that deals with sensitive data. In particular, as discussed previously, ones that read confidentiality-sensitive data and write to integrity-sensitive data. Once the sensitive operations are identified, it is trivial to find the respective Block that contains them by examining the (A)ST. In order to detect the sensitive operations, it is necessary to identify the sensitive data.

5.2.3 Putting Everything Together: An Example

Consider the following snippet of a sample Wallet applet (the full listing is available in Appendix A):

Listing 5.5: Wallet Applet Snippet

```
1  package example.wallet;
2  import javacard.framework.*;
3
4  public class Wallet extends Applet {
5    //** constants declaration omitted
6
7    OwnerPIN pin;                                              I = {pin}, C = {pin}
8
9    //@Sensitive(SensitiveData.INTEGRITY)
10   short balance;                                             I = {pin, balance}, C = {pin}
11
12   private Wallet (byte[] bArray, short bOffset, byte bLength){
13     pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);
14     pin.update(bArray, (short)(bOffset), MAX_PIN_SIZE);      sensitive operation
15     register();
16   }
17
18   public static void install(byte[] bArray, short bOffset, byte bLength){
19     new Wallet(bArray, bOffset, bLength);
20   }
21
22   //** select() and deselect() omitted
23
24   public void process(APDU apdu) {
25     byte[] buffer = apdu.getBuffer();
26
27     //** checks omitted
28     if (buffer[ISO7816.OFFSET_CLA] != Wallet_CLA)            not sensitive
29       ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);    fail()
30
```

```
31      switch (buffer[ISO7816.OFFSET_INS]) {
32        case GET_BALANCE: getBalance(apdu); return;
33        case DEBIT: debit(apdu); return;                                  sensitive operation
34        case CREDIT: credit(apdu); return;
35        case VERIFY: verify(apdu); return;
36        default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);      fail()
37      }
38    }
39
40    //** credit() omitted
41
42    private void debit(APDU apdu) {                                       sensitive method
43      if (!pin.isValidated())                                             sensitive condition
44          ISOException.throwIt(SW_WARNING_STATE_CHANGED);                 fail()
45
46      byte[] buffer = apdu.getBuffer();
47      byte numBytes = (byte)(buffer[ISO7816.OFFSET_LC]);
48      byte byteRead = (byte)(apdu.setIncomingAndReceive());
49
50      if ((numBytes != 1) || (byteRead != 1))                             not sensitive
51          ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);                  fail()
52
53      byte debitAmount = buffer[ISO7816.OFFSET_CDATA];
54
55      if ((debitAmount > MAX_TRANSACTION_AMOUNT)
56              || (debitAmount < 0))                                       not sensitive?
57          ISOException.throwIt(SW_WRONG_P1P2);                            fail()
58
59      if ((short)(balance - debitAmount) < (short)0)                      sensitive condition
60          ISOException.throwIt(SW_WRONG_P1P2);                            fail()
61
62      balance = (short) (balance - debitAmount);                          sensitive operation
63    }
64
65    private void getBalance(APDU apdu) {                                  not sensitive
66      byte[] buffer = apdu.getBuffer();
67      short le = apdu.setOutgoing();
68
69      if (le < 2)                                                         not sensitive
70          ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);                  fail()
71
72      apdu.setOutgoingLength((byte)2);
73
74      buffer[0] = (byte)(balance >> 8);                                   not sensitive
75      buffer[1] = (byte)(balance & 0xFF);                                 not sensitive
76
77      apdu.sendBytes((short)0, (short)2);
78    }
79
80    //** verify() omitted
81  }
82  }
```

First, `pin` and `balance` are identified as sensitive data. The `pin` is of type `OwnerPIN`, which is confidentiality-and-integrity-sensitive. As normally we don't want the balance in a wallet to be

modified by an unauthorized user, `balance` is defined as integrity-sensitive. This information cannot be derived automatically from the code. In the example, this information is provided in a comment with a specific format. A typical way to insert this kind of meta-information is by using annotations. However, annotations are not generally supported by Java Card. The new Java Card 3 supports annotations, including ones for defining sensitivity.

After the sensitive data have been identified, it is possible to heuristically find sensitive operations. Line 13 of the code is sensitive because it updates the `pin`, which is integrity-sensitive. Here, the analyzer needs to know that the method `update()` updates the data. This information can either be supplied or heuristically determined (*e.g.* by matching the method name to a set of predefined name patterns). Similarly, Line 62 is also sensitive because `balance` is assigned with a new value. Line 74 and 75, however, are not sensitive even though they use `balance` as they are reading the value and `balance` is just integrity-sensitive.

As calls to sensitive methods are also sensitive operations, we need to identify sensitive methods as well to properly identify all sensitive operations. This can be done in two fashions. First, we can follow the control flow and for each method call we recursively examine whether or not there is any sensitive operation in the method (top-down approach). The other way is to start from the sensitive operations and if they are inside a method, the method is sensitive and all calls to the method should then be flagged as sensitive operations (bottom-up approach). By doing this, we can find that `debit()` is a sensitive method and Line 33, which calls it, contains a sensitive operation.

After the sensitive data and operations are identified, we can proceed to identify the sensitive conditions. As discussed previously, we first need to identify the `fail()`. In our example, this includes Line 29, 36, 44, 51, 57, 60 and 70. All of them throw `ISOException` with status codes indicating failure.

There are 6 `if` constructs in our code, all of which are possible candidates of sensitive conditions, at Line 28, 43, 50, 55, 59 and 69. By comparing to the patterns described in Section 5.2.2.2, we can see that the conditionals at Line 28, 50, 55 and 69 are not sensitive conditions as their logical expressions do not include integrity-sensitive data. Conditionals at Line 43 and 59 are sensitive conditions, containing the `pin` and `balance`, respectively (both are of the variant 4).

From this simple example, we can see that the notion of sensitive condition in our previous discussion (which is derived from the definition in the FAULT.DOUBLECHECK pattern of Riscure guidelines) failed to consider Line 55 as sensitive. This condition checks the boundary to prevent malicious input to the sensitive operation that will subsequently take place. An attacker who manages to bypass this check can, for example, enter a negative `debitAmount`, causing the `debit` operation to increase the `balance` instead. A simple remedy of this is to include boundary check conditions as sensitive conditions. There are a limited number of variations in which the check logical expressions are normally written and thus automatical detection should be possible.

5.2.4 Sensitivity Propagation

As discussed in Section 5.2.2, confidentiality-sensitive data may ‘taint’ a field during assignments. The snippet in listing 5.6 shows an example of this.

Listing 5.6: Sensitivity Propagation Snippet 1

```
1   //@Sensitive(SensitiveData.CONFIDENTIALITY)
2   short secret;                                   C = {secret}
3   short temp1;
4   short temp2
5
6   secret = 99;
7   temp1 = 80;
8   temp2 = secret;                                 C = {secret, temp2}
9   temp1 = temp2;                                  C = {secret, temp2, temp1}
10  temp2 = 100;                                    C = {secret, temp1}
11  secret = 80;                                    C = {secret, temp1}
12  temp2 = temp1;                                  C = {secret, temp1, temp2}
```

Initially, the set of confidentiality-sensitive data (represented as the set C in the listing comments) contains the variable `secret`, which is defined as confidentiality-sensitive. At Line 8, `temp2` is assigned the value of `secret`, which makes it contains confidentiality-sensitive data and thus is included in the set. Similarly, at Line 9, `temp1` becomes tainted from the assignment of `temp2`, which has been tainted before. The `temp2` is then reassigned with a non-sensitive value at Line 10, making it no longer sensitive. Reassigning `secret` does not change the set because `secret` is specified to always contain confidentiality-sensitive data. At Line 12, `temp2` is tainted again from the value held by `temp1`, which is an old value of `secret` transferred via `temp2` previously.

Obviously, in this case we assume that an old copy of confidentiality-sensitive data does *not* lose its sensitivity when the field in which it was originally stored is reassigned with new sensitive data. In this case, we can use a forward data flow analysis with the following gen/kill algorithm:

- Maintain a set of all defined sensitive fields (call this the fixed set).
- Gen set: If the statement is a declaration of a field in the fixed set, add the field to the gen set; If the statement is an assignment with a field currently in the entry set, add the field that is being assigned (the left hand side) into the gen set. Otherwise, the set is empty.
- Kill set: If the statement is an assignment with a value or a field not in the entry set, add the field that is being assigned into the kill set, unless that field is in the fixed set. Otherwise, the set is empty.

More concretely, we have: $fixed = \{\text{secret}\}$

ℓ	$entry_\ell$	$kill_\ell$	gen_ℓ	$exit_\ell$
2	$\emptyset$	$\emptyset$	{secret}	{secret}
8	{secret}	$\emptyset$	{temp2}	{secret, temp2}
9	{secret, temp2}	$\emptyset$	{temp1}	{secret, temp2, temp1}
10	{secret, temp2, temp1}	{temp2}	$\emptyset$	{secret, temp1}
11	{secret, temp1}	$\emptyset$	$\emptyset$	{secret, temp1}
12	{secret, temp1}	$\emptyset$	{temp2}	{secret, temp1, temp2}

Things become more complicated if we assume that an old copy of confidentiality-sensitive data *does* lose its sensitivity when the field in which it was originally stored is reassigned with new sensitive data, as demonstrated in listing 5.7.

Listing 5.7: Sensitivity Propagation Snippet 2

```
1   //@Sensitive(SensitiveData.CONFIDENTIALITY)
2   short secret1;          fixed = {secret1}; C = C[secret1] = {secret1}
3   short temp1;
4   short temp2
5
6   secret1 = 99;
7   temp1 = 80;
8   temp2 = secret1;        C = C[secret1] = {secret1, temp2}
9   temp1 = temp2;          C = C[secret1] = {secret1, temp2, temp1}
10
11  //@Sensitive(SensitiveData.CONFIDENTIALITY)
12  short secret2 = 0;      fixed = {secret1, secret2}; C[secret2] = {secret2}
13                          C = C[secret1] ∪ C[secret2] = {secret1, temp2, temp1, secret2}
14  temp1 = secret2;        C[secret2] = {secret2, temp1}; C[secret1] = {secret1, temp2}
15  temp2 = 100;            C[secret1] = {secret1}; C = {secret1, secret2, temp1}
16  temp2 = temp1;          C[secret2] = {secret2, temp1, temp2}; C = {secret1, secret2, temp1, temp2}
17  secret1 = 80;           C[secret1] = {secret1}; C = {secret1, secret2, temp1, temp2}
18  secret1 = secret2;      C[secret2] = {secret2, temp1, temp2, secret1}; C = {secret1, secret2, temp1, temp2}
19  secret2 = 90;           C[secret2] = {secret2}; C = {secret1, secret2}
```

In this case, we will have to maintain separate sets for each originating confidentiality-sensitive data so that when the data is overwritten, all the tainted fields can be removed from the set. In the listing comments, C[x] denotes the set of fields that are tainted by the sensitive data from the sensitive field x. The resulting set of confidentiality-sensitive data C is the union of C[x] for all x.

The gen/kill algorithm has to be adapted accordingly, with separate gen set and kill set for each sensitive field:

- Gen set: If the statement is a declaration of a field x in the fixed set, add x to the gen set of x; If the statement is an assignment with a field currently in the entry set of y, add the field x that is being assigned (the left hand side) into the gen set of y. Otherwise, the set is empty.
- Kill set: If the statement is an assignment to a field in the entry set of y with a value or a field not in the entry set of y, add the field x that is being assigned into the kill set of y, unless x is in the fixed set. If the field x is in the fixed set, add all fields in the entry set of x except x into the kill set of x. Otherwise, the set is empty.

Applying the algorithm to the example code, we get: (s denotes secret and t denotes temp)

ℓ	$fixed_\ell$	$entry_\ell$	$kill_\ell$	gen_ℓ	$exit_\ell$
2	{s1}	∅	∅	s1 = {s1}	s1 = {s1}
8	{s1}	s1 = {s1}	∅	s1 = {t2}	s1 = {s1, t2}
9	{s1}	s1 = {s1, t2}	∅	s1 = {t1}	s1 = {s1, t2, t1}
12	{s1, s2}	s1 = {s1, t2, t1}	∅	s2 = {s2}	s1 = {s1, t2, t1}, s2 = {s2}
14	{s1, s2}	s1 = {s1, t2, t1}, s2 = {s2}	s1 = {t1}	s2 = {t1}	s1 = {s1, t2}, s2 = {s2, t1}
15	{s1, s2}	s1 = {s1, t2}, s2 = {s2, t1}	s1 = {t2}	∅	s1 = {s1}, s2 = {s2, t1}
16	{s1, s2}	s1 = {s1}, s2 = {s2, t1}	∅	s2 = {t2}	s1 = {s1}, s2 = {s2, t1, t2}
17	{s1, s2}	s1 = {s1}, s2 = {s2, t1, t2}	s1 = {s1}	∅	s1 = {s1}, s2 = {s2, t1, t2}
18	{s1, s2}	s1 = {s1}, s2 = {s2, t1, t2}	∅	s2 = {s1}	s1 = {s1}, s2 = {s2, t1, t2, s1}
19	{s1, s2}	s1 = {s1}, s2 = {s2, t1, t2, s1}	s2 = {t1, t2, s1}	∅	s1 = {s1}, s2 = {s2}

5.3 Analysis of the Patterns

In this section we will look more thoroughly into each of the patterns for preventing fault injection attacks.

FAULT.CRYPTO *Check for fault injection during or after crypto. Verify any ciphered data before transmission by deciphering or repeated enciphering. If the deciphered data matches the original data to be ciphered, or the repeated enciphering matches the original result, it is most likely that the encryption was not corrupted.*

This pattern can be checked quite straightforwardly by keeping track of encrypted data using data flow analysis. Afterwards, we should ensure that there is a check of whether *plaintext* = `decrypt`(*data*) or *data* = `encrypt`(*plaintext*) before any transmission operation of such encrypted data takes place. In the case where the standard cryptographic algorithm is not used, it will be necessary to indicate the encryption/decryption functions.

We can easily adapt this into a tool that has a taint propagation capability such as Fortify (*see* Section 3.1.1). The dataflow source will be the encryption function, the sink is the transmission operation and the cleansing function is the (deciphering/repeated enciphering) check.

FAULT.CONSTANT.CODING *Do not use trivial constants for sensitive data. These constants should use non-trivial values that are unlikely to be set through fault injection.*

The usage of trivial constants is easy to check by using the structural analysis. The pattern also recommends that a set of values should preferably have the maximum hamming distance. The analyzer can easily calculate the hamming distances of a set of values and issue a warning if they are under certain threshold. As we cannot easily see the grouping of such values from the code, we need to have this information either supplied or inferred using some heuristics (*e.g.* state variables in Java Card applet code are often named in the form of `STATE_*`).

FAULT.DETECT *Verify sensitive data. Sensitive data can for instance be protected by checksum. Data protected in this way should be verified at regular intervals. Ideally the integrity of sensitive data should be verified each time when used. For convenience, data can be encapsulated in so-called security controlled objects that have their own methods to preserve integrity.*

In order to precisely check this pattern, we need to know what kind of checksum is being used. This information can either be supplied or determined from a set of possible approaches. Once this information is obtained along with the information of sensitive data, it is possible to check this pattern by ensuring that every usage of sensitive data (*i.e.* sensitive operations) takes place in one of the variety of sensitive conditions as described in Section 5.2.2.2, with the `guard` being the checksum test.

FAULT.DEFAULTFAIL *When checking conditions (switch or if) check all possible cases and fail by default. Also a final else statement in an if-else construct should lead to a fail.*

To check this pattern, we simply need to inspect the BlockStatements in the switch with `default` SwitchLabel and Statement in the `else` part of IfThenElseStatement for a `fail`.

FAULT.FLOW *Use counters that keep track of the correctness of the execution path. Check counters to verify completion of execution path.*

This pattern is difficult to check unless specific forms of check counters are assumed. In the case that a single counter approach as described in the guidelines is used, we can check whether or not there is a variable that is incremented multiple times (proportional to the number of statements) throughout the code.

FAULT.DOUBLECHECK *Double check sensitive conditions. A conditional process based on sensitive data should double check the data. Preferably these checks should not be identical, but complementary as the attacker will have to perform two different types of attack.*

Similarly to FAULT.DETECT, we can check this pattern by doing structural analysis. In this case, however, we need to check that every sensitive condition is either nested in another (higher level) sensitive condition's `block_with_sensitive_op()` or contains at least one other (lower level) sensitive condition in its `block_with_sensitive_op()`. For more accurate analysis, the `guard`s of such sensitive conditions should be analyzed to ensure they refer to the same condition. Considering that the same condition can be represented in various ways, this more accurate analysis is hard. Approximations can be made, for example, by only checking if the conditions use the same set of variables (with integrity-sensitive data).

FAULT.LOOPCHECK *Verify loop completion.*

It is possible to check whether or not this pattern is being applied by utilizing the AST. In particular, an IfThenStatement or IfThenElseStatement should follow with the Expression being either the same or the complement of the terminating condition of the loop.

This check is redundant and should not be necessary if FAULT.FLOW is applied.

FAULT.BRANCH *Do not use booleans for sensitive decisions. Fault injection typically changes actual values to simple values, like 0 or 1. Non-trivial numerical values are more difficult to set by fault injection. Sensitive choices should therefore not be coded as a boolean value, but rather as a non-trivial numerical value.*

It is fairly easy to check this pattern once the sensitive conditions are identified by checking the type of variables in the `guard`.

FAULT.RESPOND *Monitor and respond to fault injection attacks. Monitoring can be done by repeatedly checking known data for changes (see FAULT.DETECT). The defence could consist of incident logging and temporarily or permanently disabling functionality.*

This pattern is related to FAULT.DETECT. The defence need to be specified more clearly in order to check whether or not this pattern is applied.

FAULT.DELAY *Use random length delays around the use of sensitive code and data to reduce the risk of success. Time based fault injection can become impractical.*

To check this pattern, we will need to identify the random length delays. Loops with one of the parameter containing a random value might be a good indicator of such delays. Once these delays can be identified, we will just need to see whether or not sensitive operations are preceded by delays. Note that the delays may be encapsulated in methods, in which case the information should be propagated as necessary.

FAULT.BYPASS *Make sure that faults are detected in the same function that executes, or invokes the protected functionality.*

This pattern is an extension to FAULT.DETECT. It is easy to check if the detection is in the same function that contains sensitive operations by observing the AST.

5.4 Pattern–Tools Capabilities Mapping

The major difficulty of checking the presence of the patterns in Riscure guidelines is that we need to identify the sensitive data and operations. Using the heuristic approaches presented earlier in this chapter will require a custom data flow analysis, which may require a considerable effort to implement. Similarly, if we decide to supply the information about sensitive data and operations manually, we need to extend the tools to accept this information. Nevertheless, the basic building blocks of these approaches are available already in most tools. Therefore, in the mapping presented in Table 5.1, most pattern-tool combinations are placed in level 2, which means that some of the necessary capabilities are already available in the tool but it might require considerable effort to implement such check. A major exception to this is Jlint, whose capabilities are fixed to its built-in knowledge. This means that extending the tool will require more effort, placing it to level 3 for all patterns (considerable effort is necessary).

Another exception is the pattern FAULT.LOOPCHECK. This pattern does not deal exclusively with sensitive data or operations and, therefore, the detection is not necessary. Consequently, this pattern is easier to check than the others.

Table 5.1: Riscure's Guidelines Pattern–Tools Capabilities Mapping

Pattern	Description	Fortify	Jtest	CxSuite	PMD	FindBugs	Jlint
CRYPTO	Verify any ciphered data before transmission by deciphering or repeated enciphering	1	2	2	2	2	3
CONSTANT_CODING	Do not use trivial constants for sensitive data	2	2	2	2	2	3
DETECT	Verify sensitive data, for instance by checksum	2	2	2	2	2	3
DEFAULTFAIL	When checking conditions check all possible cases and fail by default	2	2	2	2	2	3
FLOW	Use counters that keep track of the correctness of the execution path	2	2	2	2	2	3
DOUBLECHECK	Double check sensitive conditions	2	2	2	2	2	3
LOOPCHECK	Verify loop completion	1	1	1	1	1	2
BRANCH	Do not use booleans for sensitive decisions	2	2	2	2	2	3
RESPOND	Monitor and respond to fault injection attacks	2	2	2	2	2	3
DELAY	Use random length delays around the use of sensitive code and data to reduce the risk of success	2	2	2	2	2	3

Continued on next page . . .

Table 5.1 – *Continued from previous page*

Pattern	Description	Fortify	Jtest	CxSuite	PMD	FindBugs	Jlint
BYPASS	Make sure that faults are detected in the same function that executes, or invokes the protected functionality	2	2	2	2	2	3

Note: The numbers stated for each Pattern/Tool combination indicate the estimated difficulty of implementing the rule in the particular tool

0: already available in the tool (the soundness and completeness are tool-dependant)

1: necessary capabilities are already available in the tool, can be implemented with minimal effort

2: some of the necessary capabilities are already available in the tool, it might require considerable effort to implement

3: very little or no necessary capabilities are available in the tool, it will require considerable effort to implement

As with any other applications of static analysis, it is possible to do easier analyses which are less accurate by taking certain assumptions. For example, by assuming that all decisions are sensitive, the FAULT.BRANCH pattern can be detected easily by checking whether or not booleans are used in conditionals. While in general this assumption is not likely to be true and will result in false positives, this kind of sloppier analyses might still be beneficial for assisting human reviewers.

CHAPTER 6

Conclusions

6.1 Summary

Code review is one of the most effective best practices of software security. However, it is generally a tedious and error-prone process. In order to reduce errors and the workload of human reviewer, various assisting tools have been developed. Most of the tools belong to the class of bug finders that employ static code analysis technique.

In the recent years, Java cards, smartcards that use Java technology, have been used extensively in many different applications. These applications include SIM and banking cards, which require a high degree of security. In the process to fulfill this requirement, one of the most common approaches taken is code review.

While there are various static code analysis tools available to assist code review, none of them are designed to scan Java Card applets. Although generic Java static code analysis tools can be used, this will result in less accurate results due to some differences in the language specifications. Furthermore, as smartcards, Java cards are also prone to different kinds of attacks such as the side channel analysis and fault injection attacks, which require different preventive code protections. On the other hand, through the experience of developing Java Card applets, various knowledge catalogs have been produced. The basic idea of this project is to utilize these knowledge catalogs to bring the software security knowledge of Java cards, or smartcards in general, into the static code analysis tools.

In particular, the goal of this project was to investigate to what extent static code analysis can be used to verify compliance to existing development guidelines and programming patterns in order to have automated security reviews of Java Card applets. In order to do so, a study has been conducted to examine the capabilities of several prominent static analysis tools. Additionally, two different knowledge catalogs have been analyzed: the AFSCM Cardlet Development Guidelines v2.2 [AFSCM, 2012] and the Riscure Secure Application Programming Patterns [Witteman, 2012].

6.2 Result

From the study, it was found that most modern static code analysis tools have sufficient capabilities to implement the knowledge derived from the analyzed knowledge catalogs. However, the

knowledge needs to be specified in a concrete manner, in the form of rules. Knowledge that is described in a more abstract manner such as guideline or principle needs to be specified into rules first before it can be incorporated into the static code analysis knowledge base.

It was observed from the survey that commercial tools tend to be more intensively developed and contain a larger set of built-in knowledge. However, due to its open nature, open source tools in general offer more flexibility for customization. Nevertheless, all of the analyzed tools can be extended to implement at least some of the rules from the knowledge catalogs, with varying degree of difficulties.

In the commercial tools, the extensions are limited into the knowledge base (*i.e.* we can write new rules, but not adding new analysis functionalities). Due to the imposed licensing agreement, it was not possible to analyze and compare the analysis capabilities of the tools in detail. However, the basic analysis capabilities of all the analyzed commercial tools do not appear to significantly differ. All the tools use structural as well as interprocedural control and data flow analysis. The differentiation of the tools seems to be placed in the area of the built-in knowledge and the easy-to-use factor, which ranges from the customization of rules to the integration capabilities to various other tools in the software development process.

The open source tools vary a bit more significantly in their analysis capabilities and approach. As it was developed for bug finding purpose since its inception, FindBugs provides the most comprehensive analysis techniques. PMD was first developed for style checking purpose. Consequently, it has a solid structural analysis capabilities but the control and dataflow analysis, which were contributed later, are still quite basic. Both FindBugs and PMD allow easy extensions to the knowledge base as well as to the analysis modules. Jlint is less developed and more focused to a specific class of usage. It does not provide an easy way to add new knowledge into its knowledge base. However, as modifications to the code are allowed, it is still possible to extend both the knowledge base and the analysis techniques.

By observing the differences discussed previously, in general open source tools are more suitable if we want to incorporate rules that require custom analysis techniques, such as the Riscure guidelines. For simpler rules such as ones of the AFSCM guidelines, using commercial tools can be a good alternative as they are generally better maintained. Unless it is not desired to work on bytecode for any reason, FindBugs seems to be the best option due to its flexibility. Otherwise, PMD is a good alternative despite its limited data flow analysis capabilities.

The difficulty level of applying knowledge that has been derived into rules will vary depending on the desired level of accuracy and performance as well as the assumptions made of the code (for example, state variables can be easily identified if developers are assumed to follow a common practice of using `STATE_*` as the names).

The major contributions of this work are the comparison of several static analysis tools and the analyses of the AFSCM guidelines and the Riscure fault injection secure programming patterns.

6.3 Future Work

This study provides an analysis of two knowledge catalogs. The logical next step is to use the analysis result to implement the rules in chosen static analysis tools. An early effort has been done to do so using the PMD static code analysis tools. This project is available online at `http://www.roberto.li/security/javacard`. Clearly, it will also be interesting to analyze more related knowledge catalogs.

As mentioned in Chapter 5, the Riscure guidelines need to be derived into rules. Several approaches have been suggested in the chapter, but more research in this area is necessary.

The comparison of the tools described in this work is limited in several ways. First of all, the comparison only covers a number of static analysis tools. This is mainly due to the difficulties of obtaining licenses from the commercial tools vendors for the evaluation purpose. A further work can be done in this department, analyzing and comparing more available tools. Furthermore, this study took for granted the capabilities reported by the documentation of the tools and did not investigate further on the performance (*e.g.* false positive/false negative, speed) of such capabilities. A brief look at the available source code of the tools and the provided rules suggests that some tools opted to sacrifice accuracy for speed, or vice versa. While this trade-off is perfectly reasonable and generally resulted from experience along with various assumptions, it will be interesting to benchmark the performance of the tools, especially to see how they perform on "real-world" applications.

Several benchmark suites for various programming languages and specific application categories (*e.g.* web applications) are available to check the performance of analysis tools. However, there is currently no such suite for Java Card applets nor smartcard applications in general. A development of Java Card and smartcard applications-focused benchmark suites can be beneficial especially to assess the implementation of related rules as previously suggested.

Finally, the knowledge catalogs and their analyses presented in this work consider only the classic Java Card (2.x) technology. A newer Java Card 3 is coming and it provides various additional features that might impact security. Once this newer technology has been adopted in the market, an update to the knowledge catalogs and the analyses might be necessary.

Appendix A

Code Example

This chapter contains the full listing of the example referred in Chapter 5.

Listing A.1: Example Java Card Wallet applet

```
package example.wallet;
import javacard.framework.*;

public class Wallet extends Applet {
  final static byte Wallet_CLA = (byte) 0x80;

  // instructions
  final static byte SELECT = (byte) 0xA4;
  final static byte VERIFY = (byte) 0x20;
  final static byte CREDIT = (byte) 0x30;
  final static byte DEBIT = (byte) 0x40;
  final static byte GET_BALANCE = (byte) 0x50;

  // limits
  final static short MAX_BALANCE = 0x7FFF;
  final static byte MAX_TRANSACTION_AMOUNT = 127;
  final static byte PIN_TRY_LIMIT = (byte) 0x03;
  final static byte MAX_PIN_SIZE = (byte) 0x04;

  // status code
  final static short SW_WARNING_STATE_CHANGED = 0x6300;
  final static short SW_WRONG_P1P2 = 0x6B00;

  OwnerPIN pin;

  //@Sensitive(SensitiveData.INTEGRITY)
  short balance;

  private Wallet (byte[] bArray, short bOffset, byte bLength){
    pin = new OwnerPIN(PIN_TRY_LIMIT, MAX_PIN_SIZE);
    pin.update(bArray, (short)(bOffset), MAX_PIN_SIZE);
    register();
  }

  public static void install(byte[] bArray, short bOffset, byte bLength){
    new Wallet(bArray, bOffset, bLength);
  }
```

```
  public boolean select() {
    if (pin.getTriesRemaining() == 0)
        return false;
    return true;
  }

  public void deselect() {
    pin.reset();
  }

  public void process(APDU apdu) {
    byte[] buffer = apdu.getBuffer();

    // ignore channel bits
    buffer[ISO7816.OFFSET_CLA] = (byte)(buffer[ISO7816.OFFSET_CLA] & (byte)0xFC);

    if ((buffer[ISO7816.OFFSET_CLA] == 0) && (buffer[ISO7816.OFFSET_INS] == SELECT))
        return;
    if (buffer[ISO7816.OFFSET_CLA] != Wallet_CLA)
        ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);

    switch (buffer[ISO7816.OFFSET_INS]) {
      case GET_BALANCE: getBalance(apdu); return;
      case DEBIT: debit(apdu); return;
      case CREDIT: credit(apdu); return;
      case VERIFY: verify(apdu); return;
      default: ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
  }

  private void credit(APDU apdu) {
    if (! pin.isValidated())
        ISOException.throwIt(SW_WARNING_STATE_CHANGED);

    byte[] buffer = apdu.getBuffer();
    byte numBytes = buffer[ISO7816.OFFSET_LC];
    byte byteRead = (byte)(apdu.setIncomingAndReceive());

    if ((numBytes != 1) || (byteRead != 1))
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    byte creditAmount = buffer[ISO7816.OFFSET_CDATA];

    if ((creditAmount > MAX_TRANSACTION_AMOUNT) || (creditAmount < 0))
         ISOException.throwIt(SW_WRONG_P1P2);

    if ((short)(balance + creditAmount) > MAX_BALANCE)
        ISOException.throwIt(SW_WRONG_P1P2);

    balance = (short)(balance + creditAmount);
  }

  private void debit(APDU apdu) {
    if (!pin.isValidated())
        ISOException.throwIt(SW_WARNING_STATE_CHANGED);
```

```
    byte[] buffer = apdu.getBuffer();
    byte numBytes = (byte)(buffer[ISO7816.OFFSET_LC]);
    byte byteRead = (byte)(apdu.setIncomingAndReceive());

    if ((numBytes != 1) || (byteRead != 1))
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    byte debitAmount = buffer[ISO7816.OFFSET_CDATA];

    if ((debitAmount > MAX_TRANSACTION_AMOUNT) || (debitAmount < 0))
        ISOException.throwIt(SW_WRONG_P1P2);

    if ((short)(balance - debitAmount) < (short)0)
        ISOException.throwIt(SW_WRONG_P1P2);

    balance = (short) (balance - debitAmount);

  }

  private void getBalance(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    short le = apdu.setOutgoing();

    if (le < 2)
        ISOException.throwIt(ISO7816.SW_WRONG_LENGTH);

    apdu.setOutgoingLength((byte)2);

    buffer[0] = (byte)(balance >> 8);
    buffer[1] = (byte)(balance & 0xFF);

    apdu.sendBytes((short)0, (short)2);
  }

  private void verify(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    byte byteRead = (byte)(apdu.setIncomingAndReceive());

  if (pin.check(buffer, ISO7816.OFFSET_CDATA, byteRead) == false)
     ISOException.throwIt(SW_WARNING_STATE_CHANGED);
}
}
```

APPENDIX **B**

Java Grammar

This chapter describes a complete grammar of Java 1.6 as defined in [Bosworth, 2011]. The grammar is extracted from the Java Language Specification, Third Edition [Gosling et al., 2005] and optimized for LL(1) parser.

B.1 Grammar Notation

The format of the rules is adapted from the one described in Chapter 2 of [Gosling et al., 2005], with several minor modifications in typesetting.

Terminal symbols are shown in `fixed width` font. These are to appear in a program exactly as written.

Nonterminal symbols are shown in sans serif typeface. The definition of a nonterminal is introduced by the name of the nonterminal being defined followed by a colon. One or more alternative right-hand sides for the nonterminal then follow on succeeding lines. For example, the syntactic definition:

IfThenStatement:
    `if` `(` Expression `)` Statement

states that the nonterminal IfThenStatement represents the token `if`, followed by a left parenthesis token, followed by an Expression, followed by a right parenthesis token, followed by a Statement.

As another example, the syntactic definition:

ArgumentList:
    Argument
    ArgumentList `,` Argument

states that an ArgumentList may represent either a single Argument or an ArgumentList, followed by a comma, followed by an Argument. This definition of ArgumentList is *recursive*, that is to say, it is defined in terms of itself. The result is that an ArgumentList may contain any positive number of arguments. Such recursive definitions of nonterminals are common.

The subscripted suffix "opt", which may appear after a terminal or nonterminal, indicates an optional symbol. The alternative containing the optional symbol actually specifies two right-hand sides, one that omits the optional element and one that includes it.

This means that:

BreakStatement:
`break` Identifier$_{opt}$ `;`

is a convenient abbreviation for:

BreakStatement:
`break ;`
`break` Identifier `;`

and that:

BasicForStatement:
`for (` ForInit$_{opt}$ `;` Expression$_{opt}$ `;` ForUpdate$_{opt}$ `)` Statement

is a convenient abbreviation for:

BasicForStatement:
`for ( ;` Expression$_{opt}$ `;` ForUpdate$_{opt}$ `)` Statement
`for (` ForInit `;` Expression$_{opt}$ `;` ForUpdate$_{opt}$ `)` Statement

which in turn is an abbreviation for:

BasicForStatement:
`for ( ; ;` ForUpdate$_{opt}$ `)` Statement
`for ( ;` Expression `;` ForUpdate$_{opt}$ `)` Statement
`for (` ForInit `; ;` ForUpdate$_{opt}$ `)` Statement
`for (` ForInit `;` Expression `;` ForUpdate$_{opt}$ `)` Statement

which in turn is an abbreviation for:

BasicForStatement:
`for ( ; ; )` Statement
`for ( ; ;` ForUpdate `)` Statement
`for ( ;` Expression `; )` Statement
`for ( ;` Expression `;` ForUpdate `)` Statement
`for (` ForInit `; ; )` Statement
`for (` ForInit `; ;` ForUpdate `)` Statement
`for (` ForInit `;` Expression `; )` Statement
`for (` ForInit `;` Expression `;` ForUpdate `)` Statement

so the nonterminal BasicForStatement actually has eight alternative right-hand sides.

A very long right-hand side may be continued on a second line by substantially indenting this second line, as in:

ConstructorDeclaration:
    ConstructorModifiers$_{opt}$ ConstructorDeclarator
        Throws$_{opt}$ ConstructorBody

which defines one right-hand side for the nonterminal ConstructorDeclaration.

When the words "*one of*" follow the colon in a grammar definition, they signify that each of the terminal symbols on the following line or lines is an alternative definition. For example, the lexical grammar contains the production:

ZeroToThree: *one of*
    `0 1 2 3`

which is merely a convenient abbreviation for:

ZeroToThree:
    `0`
    `1`
    `2`
    `3`

When an alternative in a lexical production appears to be a token, it represents the sequence of characters that would make up such a token. Thus, the definition:

BooleanLiteral: *one of*
    `true false`

in a lexical grammar production is shorthand for:

BooleanLiteral:
    `t r u e`
    `f a l s e`

The right-hand side of a lexical production may specify that certain expansions are not permitted by using the phrase "*but not*" and then indicating the expansions to be excluded, as in the productions for InputCharacter and Identifier:

InputCharacter:
    UnicodeInputCharacter *but not* CR *or* LF
Identifier:
    IdentifierName *but not a* Keyword *or* BooleanLiteral *or* NullLiteral

Finally, a few nonterminal symbols are described by a descriptive phrase in *italic* roman type in cases where it would be impractical to list all the alternatives:

RawInputCharacter:
    any Unicode character

B.2 Lexical Grammar

UnicodeInputCharacter:
    UnicodeEscape
    RawInputCharacter

UnicodeEscape:
    \ UnicodeMarker HexDigit HexDigit HexDigit HexDigit

UnicodeMarker:
    u UnicodeMarker$_{opt}$

RawInputCharacter:
    any Unicode character

HexDigit: *one of*
    `0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F`

LineTerminator:
    the ASCII LF *character, also known as "newline"*
    the ASCII CR *character, also known as "return"*
    the ASCII CR *character followed by the ASCII* LF *character*

InputCharacter:
    UnicodeInputCharacter *but not* CR *or* LF

Input:
    InputElements$_{opt}$ Sub$_{opt}$

InputElements:
    InputElement InputElements$_{opt}$

InputElement:
    WhiteSpace
    Comment
    Token

Token:
    Identifier
    Keyword
    Literal
    Separator
    Operator

Sub:
    the ASCII SUB *character, also known as "control-Z"*

WhiteSpace:
    the ASCII SP *character, also known as "space"*

the ASCII HT *character, also known as "horizontal tab"*
the ASCII FF *character, also known as "form feed"*
LineTerminator

Comment:
TraditionalComment
EndOfLineComment

TraditionalComment:
/ * CommentTail

EndOfLineComment:
/ / CharactersInLine$_{opt}$

CommentTail:
* CommentTailStar
NotStar CommentTail

CommentTailStar:
/
* CommentTailStar
NotStarNotSlash CommentTail

NotStar:
InputCharacter *but not* *
LineTerminator

NotStarNotSlash:
InputCharacter *but not* * *or* /
LineTerminator

CharactersInLine:
InputCharacter CharactersInLine$_{opt}$

Identifier:
IdentifierChars *but not a* Keyword *or* BooleanLiteral *or* NullLiteral

IdentifierChars:
JavaLetter IdentifierRest$_{opt}$

IdentifierRest:
JavaLetterOrDigit IdentifierRest$_{opt}$

JavaLetter:
any Unicode character that is a Java letter

JavaLetterOrDigit:
any Unicode character that is a Java letter-or-digit

Keyword: *one of*

```
abstract  continue  for         new        switch
assert    default   if          package    synchronized
boolean   do        goto        private    this
break     double    implements  protected  throw
byte      else      import      public     throws
case      enum      instanceof  return     transient
catch     extends   int         short      try
char      final     interface   static     void
class     finally   long        strictfp   volatile
const     float     native      super      while
```

Literal:
- IntegerLiteral
- FloatingPointLiteral
- BooleanLiteral
- CharacterLiteral
- StringLiteral
- NullLiteral

IntegerLiteral:
- DecimalIntegerLiteral
- HexIntegerLiteral
- OctalIntegerLiteral

DecimalIntegerLiteral:
- DecimalNumeral $\text{IntegerTypeSuffix}_{\text{opt}}$

HexIntegerLiteral:
- HexNumeral $\text{IntegerTypeSuffix}_{\text{opt}}$

OctalIntegerLiteral:
- OctalNumeral $\text{IntegerTypeSuffix}_{\text{opt}}$

IntegerTypeSuffix: *one of*
- `l` `L`

DecimalNumeral:
- `0`
- NonZeroDigit $\text{Digits}_{\text{opt}}$

Digits:
- Digit $\text{Digits}_{\text{opt}}$

Digit:
- `0`
- NonZeroDigit

NonZeroDigit: *one of*
`1 2 3 4 5 6 7 8 9`

HexNumeral:
`0` x HexDigits
`0` X HexDigits

HexDigits:
HexDigit $\text{HexDigits}_{\text{opt}}$

OctalNumeral:
`0` OctalDigits

OctalDigits:
OctalDigit $\text{OctalDigits}_{\text{opt}}$

OctalDigit: *one of*
`0 1 2 3 4 5 6 7`

FloatingPointLiteral:
DecimalFloatingPointLiteral
HexadecimalFloatingPointLiteral

DecimalFloatingPointLiteral:
Digits `.` $\text{Digits}_{\text{opt}}$ $\text{ExponentPart}_{\text{opt}}$ $\text{FloatTypeSuffix}_{\text{opt}}$
`.` Digits $\text{ExponentPart}_{\text{opt}}$ $\text{FloatTypeSuffix}_{\text{opt}}$
Digits ExponentPart $\text{FloatTypeSuffix}_{\text{opt}}$
Digits $\text{ExponentPart}_{\text{opt}}$ FloatTypeSuffix

ExponentPart:
ExponentIndicator SignedInteger

ExponentIndicator: *one of*
`e E`

SignedInteger:
Sign_{opt} Digits

Sign: *one of*
`+ -`

FloatTypeSuffix: *one of*
`f F d D`

HexadecimalFloatingPointLiteral:
HexSignificand BinaryExponent $\text{FloatTypeSuffix}_{\text{opt}}$

HexSignificand:
HexNumeral

HexNumeral `.`
`0x` HexDigits$_{opt}$ `.` HexDigits
`0X` HexDigits$_{opt}$ `.` HexDigits

BinaryExponent:
BinaryExponentIndicator SignedInteger

BinaryExponentIndicator: *one of*
`p` `P`

BooleanLiteral: *one of*
`true` `false`

CharacterLiteral:
`'` SingleCharacter `'`
`'` EscapeSequence `'`

SingleCharacter:
InputCharacter *but not* `'` *or* `\`

StringLiteral:
`"` StringCharacters$_{opt}$ `"`

StringCharacters:
StringCharacter StringCharacters$_{opt}$

StringCharacter:
InputCharacter *but not* `"` *or* `\`
EscapeSequence

EscapeSequence:

`\ b`	`\u0008`: *backspace* BS
`\ t`	`\u0009`: *horizontal tab* HT
`\ n`	`\u000a`: *linefeed* LF
`\ f`	`\u000c`: *form feed* FF
`\ r`	`\u000d`: *carriage return* CR
`\ "`	`\u0022`: *double quote* "
`\ '`	`\u0027`: *single quote* '
`\ \`	`\u005c`: *backslash* \
OctalEscape	`\u0000` *to* `\u00ff`: *from octal value*

OctalEscape:
`\` OctalDigit
`\` OctalDigit OctalDigit
`\` ZeroToThree OctalDigit OctalDigit

ZeroToThree: *one of*
`0` `1` `2` `3`

NullLiteral:
`null`

Separator: *one of*
`( ) { } [ ] ; , .`

Operator: *one of*
```
=   >   <   !   ~   ?   :
==  <=  >=  !=  &&  ||  ++  -
+   -   *   /   &   |   ^   %   <<   >>   >>>
+=  -=  *=  /=  &=  |=  ^=  %=  <<=  >>=  >>>=
```

B.3 Syntactic Grammar

B.3.1 Types and Values

Type:
PrimitiveType
ReferenceType

PrimitiveType: *one of*
`byte short int long char float double boolean`

ReferenceType:
ClassOrInterfaceType
ArrayType

ClassOrInterfaceType:
Identifier TypeArguments$_{opt}$ ClassOrInterfaceTypeRest$_{opt}$

ClassOrInterfaceTypeRest:
`.` Identifier TypeArguments$_{opt}$ ClassOrInterfaceTypeRest$_{opt}$

InterfaceType:
ClassOrInterfaceType

ClassType:
ClassOrInterfaceType

ArrayType:
Type `[ ]`

TypeParameter:
TypeVariable TypeBound$_{opt}$

TypeVariable:
Identifier

TypeBound:
    `extends` ClassOrInterfaceType AdditionalBoundList$_{opt}$

AdditionalBoundList:
    `&` InterfaceType AdditionalBoundList$_{opt}$

TypeArguments:
    `<` ActualTypeArgumentList `>`

ActualTypeArgumentList:
    ActualTypeArgument
    ActualTypeArgument `,` ActualTypeArgumentList

ActualTypeArgument:
    ReferenceType
    Wildcard

Wildcard:
    `?` WildcardBounds$_{opt}$

WildcardBounds:
    `extends` ReferenceType
    `super` ReferenceType

B.3.2 Names

PackageName:
    Identifier
    Identifier `.` PackageName

TypeName:
    Identifier
    Identifier `.` PackageOrTypeName

ExpressionName:
    Identifier
    Identifier `.` AmbiguousName

MethodName:
    Identifier
    Identifier `.` AmbiguousName

PackageOrTypeName:
    Identifier
    Identifier `.` PackageOrTypeName

AmbiguousName:
    Identifier
    Identifier `.` AmbiguousName

B.3.3 Packages

CompilationUnit:
PackageDeclaration$_{opt}$ ImportDeclarations$_{opt}$ TypeDeclarations$_{opt}$

ImportDeclarations:
ImportDeclaration
ImportDeclaration ImportDeclarations

TypeDeclarations:
TypeDeclaration
TypeDeclaration TypeDeclarations

PackageDeclaration:
Annotations$_{opt}$ `package` PackageName `;`

ImportDeclaration:
`import` ImportDeclarationRest

ImportDeclarationRest:
PackageOrTypeName TypeImportDeclarationRest
`static` TypeName `.` StaticImportDeclarationRest

TypeImportDeclarationRest:
`;`
`.` `*` `;`

StaticImportDeclarationRest:
Identifier `;`
`*` `;`

TypeDeclaration:
ClassDeclaration
InterfaceDeclaration
`;`

B.3.4 Classes

ClassDeclaration:
NormalClassDeclaration
EnumDeclaration

NormalClassDeclaration:
ClassModifiers$_{opt}$ `class` Identifier TypeParameters$_{opt}$ Super$_{opt}$ Interfaces$_{opt}$ ClassBody

ClassModifiers:
ClassModifier ClassModifiers$_{opt}$

ClassModifier: *one of*
Annotation `public` `protected` `private` `abstract` `static` `final` `strictfp`

TypeParameters:
    < TypeParameterList >

TypeParameterList:
    TypeParameter
    TypeParameter , TypeParameterList

Super:
    `extends` ClassType

Interfaces:
    `implements` InterfaceTypeList

InterfaceTypeList:
    InterfaceType
    InterfaceType , InterfaceTypeList

ClassBody:
    { $\text{ClassBodyDeclarations}_{opt}$ }

ClassBodyDeclarations:
    ClassBodyDeclaration
    ClassBodyDeclaration ClassBodyDeclarations

ClassBodyDeclaration:
    FieldDeclaration
    MethodDeclaration
    ClassDeclaration
    InterfaceDeclaration
    ;
    InstanceInitializer
    StaticInitializer
    ConstructorDeclaration

FieldDeclaration:
    $\text{FieldModifiers}_{opt}$ Type VariableDeclarators ;

VariableDeclarators:
    VariableDeclarator
    VariableDeclarator , VariableDeclarators

VariableDeclarator:
    VariableDeclaratorId
    VariableDeclaratorId = VariableInitializer

VariableDeclaratorId:
    Identifier Dims_{opt}

Dims:
    `[ ]` Dims$_{opt}$

VariableInitializer:
    Expression
    ArrayInitializer

FieldModifiers:
    FieldModifier
    FieldModifier FieldModifiers

FieldModifier: *one of*
    Annotation `public protected private static final transient volatile`

MethodDeclaration:
    MethodHeader MethodBody

MethodHeader:
    MethodModifiers$_{opt}$ TypeParameters$_{opt}$ TypedMethodDeclarator Throws$_{opt}$

TypedMethodDeclarator:
    Type MethodDeclarator Dims$_{opt}$
    `void` MethodDeclarator

MethodDeclarator:
    Identifier `(` FormalParameterList$_{opt}$ `)`

FormalParameterList:
    LastFormalParameter
    FormalParameters `,` LastFormalParameter

FormalParameters:
    FormalParameter
    FormalParameter `,` FormalParameters

FormalParameter:
    VariableModifiers$_{opt}$ Type VariableDeclaratorId

VariableModifiers:
    VariableModifier VariableModifiers$_{opt}$

VariableModifier:
    `final`
    Annotation

LastFormalParameter:
    VariableModifiers$_{opt}$ Type `...`$_{opt}$ VariableDeclaratorId

MethodModifiers:
    MethodModifier MethodModifiers$_{opt}$

MethodModifier: *one of*
    Annotation `public protected private abstract static final synchronized native strictfp`

Throws:
    `throws` ExceptionTypeList

ExceptionTypeList:
    ExceptionType
    ExceptionType `,` ExceptionTypeList

ExceptionType:
    ClassType
    TypeVariable

MethodBody:
    Block
    `;`

InstanceInitializer:
    Block

StaticInitializer:
    `static` Block

ConstructorDeclaration:
    ConstructorModifiers$_{opt}$ ConstructorDeclarator Throws$_{opt}$ ConstructorBody

ConstructorDeclarator:
    TypeParameters$_{opt}$ SimpleTypeName `(` FormalParameterList$_{opt}$ `)`

SimpleTypeName:
    Identifier

ConstructorModifiers:
    ConstructorModifier ConstructorModifiers$_{opt}$

ConstructorModifier: *one of*
    Annotation `public protected private`

ConstructorBody:
    `{` ExplicitConstructorInvocation$_{opt}$ BlockStatements$_{opt}$ `}`

ExplicitConstructorInvocation:
    NonWildTypeArguments$_{opt}$ `this` `(` ArgumentList$_{opt}$ `)` `;`

NonWildTypeArguments$_{opt}$ `super` `(` ArgumentList$_{opt}$ `)` `;`
Primary `.` NonWildTypeArguments$_{opt}$ `super` `(` ArgumentList$_{opt}$ `)` `;`

NonWildTypeArguments:
`<` ReferenceTypeList `>`

ReferenceTypeList:
ReferenceType
ReferenceType `,` ReferenceTypeList

EnumDeclaration:
ClassModifiers$_{opt}$ `enum` Identifier Interfaces$_{opt}$ EnumBody

EnumBody:
`{` EnumConstantsopt `,`$_{opt}$ EnumBodyDeclarations$_{opt}$ `}`

EnumConstants:
EnumConstant
EnumConstant `,` EnumConstants

EnumConstant:
Annotations$_{opt}$ Identifier Arguments$_{opt}$ ClassBody$_{opt}$

Arguments:
`(` ArgumentList$_{opt}$ `)`

EnumBodyDeclarations:
`;` ClassBodyDeclarations$_{opt}$

B.3.5 Interfaces

InterfaceDeclaration:
NormalInterfaceDeclaration
AnnotationTypeDeclaration

NormalInterfaceDeclaration:
InterfaceModifiers$_{opt}$ `interface` Identifier TypeParameters$_{opt}$ ExtendsInterfaces$_{opt}$ InterfaceBody

InterfaceModifiers:
InterfaceModifier InterfaceModifiers$_{opt}$

InterfaceModifier: *one of*
Annotation `public protected private abstract static strictfp`

ExtendsInterfaces:
`extends` InterfaceTypeList

InterfaceBody:
{ InterfaceMemberDeclarations$_{opt}$ }

InterfaceMemberDeclarations:
InterfaceMemberDeclaration InterfaceMemberDeclarations$_{opt}$

InterfaceMemberDeclaration:
ConstantDeclaration
AbstractMethodDeclaration
ClassDeclaration
InterfaceDeclaration
;

ConstantDeclaration:
ConstantModifiers$_{opt}$ Type VariableDeclarators ;

ConstantModifiers:
ConstantModifier ConstantModifiers$_{opt}$

ConstantModifier: *one of*
Annotation `public static final`

AbstractMethodDeclaration:
AbstractMethodModifiers$_{opt}$ TypeParameters$_{opt}$ TypedMethodDeclarator Throws$_{opt}$;

AbstractMethodModifiers:
AbstractMethodModifier AbstractMethodModifiers$_{opt}$

AbstractMethodModifier: *one of*
Annotation `public abstract`

AnnotationTypeDeclaration:
InterfaceModifiers$_{opt}$ `@ interface` Identifier AnnotationTypeBody

AnnotationTypeBody:
{ AnnotationTypeElementDeclarations$_{opt}$ }

AnnotationTypeElementDeclarations:
AnnotationTypeElementDeclaration AnnotationTypeElementDeclarations$_{opt}$

AnnotationTypeElementDeclaration:
AbstractMethodModifiers$_{opt}$ Type Identifier `( )` DefaultValue$_{opt}$;
ConstantDeclaration
ClassDeclaration
InterfaceDeclaration
EnumDeclaration
AnnotationTypeDeclaration
;

DefaultValue:
    `default` ElementValue

Annotations:
    Annotation Annotations_{opt}

Annotation:
    @ TypeName $\text{AnnotationRest}_{opt}$

AnnotationRest:
    NormalAnnotationRest
    SingleElementAnnotationRest

NormalAnnotationRest:
    ($\text{ElementValuePairs}_{opt}$)

ElementValuePairs:
    ElementValuePair
    ElementValuePair , ElementValuePairs

ElementValuePair:
    Identifier = ElementValue

ElementValue:
    ConditionalExpression
    Annotation
    ElementValueArrayInitializer

ElementValueArrayInitializer:
    { $\text{ElementValues}_{opt}$ $,_{opt}$ }

ElementValues:
    ElementValue
    ElementValue , ElementValues

SingleElementAnnotationRest:
    (ElementValue)

B.3.6 Arrays

ArrayInitializer:
    { $\text{VariableInitializers}_{opt}$ $,_{opt}$ }

VariableInitializers:
    VariableInitializer
    VariableInitializer , VariableInitializers

B.3.7 Blocks and statements

Block:
    { BlockStatements$_{opt}$ }

BlockStatements:
    BlockStatement BlockStatements$_{opt}$

BlockStatement:
    LocalVariableDeclarationStatement
    ClassDeclaration
    Statement

LocalVariableDeclarationStatement:
    LocalVariableDeclaration ;

LocalVariableDeclaration:
    VariableModifiers Type VariableDeclarators

Statement:
    StatementWithoutTrailingSubstatement
    LabeledStatement
    IfThenStatement
    IfThenElseStatement
    WhileStatement
    ForStatement

StatementWithoutTrailingSubstatement:
    Block
    EmptyStatement
    ExpressionStatement
    AssertStatement
    SwitchStatement
    DoStatement
    BreakStatement
    ContinueStatement
    ReturnStatement
    SynchronizedStatement
    ThrowStatement
    TryStatement

StatementNoShortIf:
    StatementWithoutTrailingSubstatement
    LabeledStatementNoShortIf
    IfThenElseStatementNoShortIf
    WhileStatementNoShortIf
    ForStatementNoShortIf

IfThenStatement:
`if` (Expression) Statement

IfThenElseStatement:
`if` (Expression) StatementNoShortIf `else` Statement

IfThenElseStatementNoShortIf:
`if` (Expression) StatementNoShortIf `else` StatementNoShortIf

EmptyStatement:
;

LabeledStatement:
Identifier : Statement

LabeledStatementNoShortIf:
Identifier : StatementNoShortIf

ExpressionStatement:
StatementExpression ;

StatementExpression:
Assignment
PreIncrementExpression
PreDecrementExpression
PostIncrementExpression
PostDecrementExpression
MethodInvocation
ClassInstanceCreationExpression

AssertStatement:
`assert` Expression_1 ;
`assert` Expression_1 : Expression_2 ;

SwitchStatement:
`switch` (Expression) SwitchBlock

SwitchBlock:
{ $\text{SwitchBlockStatementGroups}_{\text{opt}}$ $\text{SwitchLabels}_{\text{opt}}$ }

SwitchBlockStatementGroups:
SwitchBlockStatementGroup $\text{SwitchBlockStatementGroups}_{\text{opt}}$

SwitchBlockStatementGroup:
SwitchLabels BlockStatements

SwitchLabels:
SwitchLabel $\text{SwitchLabels}_{\text{opt}}$

SwitchLabel:
`case` ConstantExpression `:`
`case` EnumConstantName `:`
`default :`

EnumConstantName:
Identifier

WhileStatement:
`while (` Expression `)` Statement

WhileStatementNoShortIf:
`while (` Expression `)` StatementNoShortIf

DoStatement:
`do` Statement `while (` Expression `) ;`

ForStatement:
BasicForStatement
EnhancedForStatement

BasicForStatement:
`for (` $\text{ForInit}_{\text{opt}}$ `;` $\text{Expression}_{\text{opt}}$ `;` $\text{ForUpdate}_{\text{opt}}$ `)` Statement

ForStatementNoShortIf:
`for (` $\text{ForInit}_{\text{opt}}$ `;` $\text{Expression}_{\text{opt}}$ `;` $\text{ForUpdate}_{\text{opt}}$ `)` StatementNoShortIf

ForInit:
StatementExpressionList
LocalVariableDeclaration

ForUpdate:
StatementExpressionList

StatementExpressionList:
StatementExpression
StatementExpression `,` StatementExpressionList

EnhancedForStatement:
`for (` $\text{VariableModifiers}_{\text{opt}}$ Type Identifier `:` Expression `)` Statement

BreakStatement:
`break` $\text{Identifier}_{\text{opt}}$ `;`

ContinueStatement:
`continue` $\text{Identifier}_{\text{opt}}$ `;`

ReturnStatement:
`return` $\text{Expression}_{\text{opt}}$ `;`

ThrowStatement:
`throw` Expression `;`

SynchronizedStatement:
`synchronized` `(` Expression `)` Block

TryStatement:
`try` Block Catches
`try` Block $\text{Catches}_{\text{opt}}$ `finally` Block

Catches:
CatchClause $\text{Catches}_{\text{opt}}$

CatchClause:
`catch` `(` FormalParameter `)` Block

B.3.8 Expressions

Primary:
PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray:
Literal
Type `.` `class`
`void` `.` `class`
`this`
TypeName `.` `this`
`(` Expression `)`
ClassInstanceCreationExpression
FieldAccess
MethodInvocation
ArrayAccess

ClassInstanceCreationExpression:
`new` $\text{TypeArguments}_{\text{opt}}$ ClassOrInterfaceType `(` $\text{ArgumentList}_{\text{opt}}$ `)` $\text{ClassBody}_{\text{opt}}$
Primary `.` `new` $\text{TypeArguments}_{\text{opt}}$ Identifier $\text{TypeArguments}_{\text{opt}}$ `(` $\text{ArgumentList}_{\text{opt}}$ `)` $\text{ClassBody}_{\text{opt}}$

ArgumentList:
Expression
Expression `,` ArgumentList

ArrayCreationExpression:
`new` ArrayCreationExpressionRest

ArrayCreationExpressionRest:
PrimitiveType DimSpecifier
ClassOrInterfaceType DimSpecifier

DimSpecifier:
    DimExprs Dims_{opt}
    Dims ArrayInitializer

DimExprs:
    DimExpr $\text{DimExprs}_{\text{opt}}$

DimExpr:
    [Expression]

FieldAccess:
    Identifier
    Primary . Identifier
    `super` . Identifier
    TypeName . `super` . Identifier

MethodInvocation:
    MethodName BracedArgumentList
    Primary . MethodInvocationRest
    `super` . MethodInvocationRest
    TypeName . `super` . MethodInvocationRest
    TypeName . NonWildTypeArguments Identifier BracedArgumentList

BracedArgumentList:
    ($\text{ArgumentList}_{\text{opt}}$)

MethodInvocationRest:
    $\text{NonWildTypeArguments}_{\text{opt}}$ Identifier BracedArgumentList

ArrayAccess:
    ExpressionName [Expression]
    PrimaryNoNewArray [Expression]

PostfixExpression:
    Primary
    ExpressionName
    PostIncrementExpression
    PostDecrementExpression

PostIncrementExpression:
    PostfixExpression ++

PostDecrementExpression:
    PostfixExpression --

UnaryExpression:
    PreIncrementExpression
    PreDecrementExpression

+ UnaryExpression
- UnaryExpression
UnaryExpressionNotPlusMinus

PreIncrementExpression:
++ UnaryExpression

PreDecrementExpression:
-- UnaryExpression

UnaryExpressionNotPlusMinus:
PostfixExpression
~ UnaryExpression
! UnaryExpression
CastExpression

CastExpression:
(PrimitiveType Dims_{opt}) UnaryExpression
(ReferenceType) UnaryExpressionNotPlusMinus

MultiplicativeExpression:
UnaryExpression $\text{MultiplicativeExpressionRest}_{\text{opt}}$

MultiplicativeExpressionRest:
* MultiplicativeExpression
/ MultiplicativeExpression
% MultiplicativeExpression

AdditiveExpression:
MultiplicativeExpression $\text{AdditiveExpressionRest}_{\text{opt}}$

AdditiveExpressionRest:
+ AdditiveExpression
- AdditiveExpression

ShiftExpression:
AdditiveExpression $\text{ShiftExpressionRest}_{\text{opt}}$

ShiftExpressionRest:
<< ShiftExpression
>> ShiftExpression
>>> ShiftExpression

RelationalExpression:
ShiftExpression $\text{RelationalExpressionRest}_{\text{opt}}$

RelationalExpressionRest:
< RelationalExpression

> RelationalExpression
<= RelationalExpression
>= RelationalExpression
`instanceof` ReferenceType

EqualityExpression:
RelationalExpression EqualityExpressionRest$_{opt}$

EqualityExpressionRest:
== EqualityExpression
`!=` EqualityExpression

AndExpression:
EqualityExpression
EqualityExpression & AndExpression

ExclusiveOrExpression:
AndExpression
AndExpression ^ ExclusiveOrExpression

InclusiveOrExpression:
ExclusiveOrExpression
ExclusiveOrExpression | InclusiveOrExpression

ConditionalAndExpression:
InclusiveOrExpression
InclusiveOrExpression && ConditionalAndExpression

ConditionalOrExpression:
ConditionalAndExpression
ConditionalAndExpression || ConditionalOrExpression

ConditionalExpression:
ConditionalOrExpression
ConditionalOrExpression ? Expression : ConditionalExpression

AssignmentExpression:
ConditionalExpression
Assignment

Assignment:
LeftHandSide AssignmentOperator AssignmentExpression

LeftHandSide:
ExpressionName
FieldAccess
ArrayAccess

AssignmentOperator: *one of*

```
=     *=    /=    %=   +=   -=
<<=   >>=   >>>=  &=   ^=   |=
```

Expression:

AssignmentExpression

ConstantExpression:

Expression

Glossary

2G refers to the second generation of mobile telecommunication networks, which have a cellular architecture and use digital technology, instead of analog as in the first generation [Rankl & Effing, 2003]

3G refers to the third generation of mobile telecommunication networks, which improves various aspects (*e.g.* data transmission rate) of the second generation network [Rankl & Effing, 2003]

AFSCM is a non-profit association established by three French MNOs to facilitate the technical development and to promote contactless mobile services

APDU is a software data container used to package data for an application for exchange between a smartcard and a terminal; APDUs can be classified into C-APDUs and R-APDUs [Rankl & Effing, 2003]

API is a software interface, specified in detail, that provides access to specific functions of a program [Rankl & Effing, 2003]

applet is a program written in the Java programming language and executed by the virtual machine of a computer [Rankl & Effing, 2003]

asset is anything within an environment that should be protected [Tittel et al., 2004]

asynchronous data transmission is data transmission in which the data are transmitted independent of any prescribed timing reference [Rankl & Effing, 2003]

attack is the exploitation of a vulnerability by a target threat agent [Tittel et al., 2004]

attack pattern is a generalization derived from large sets of software exploits, useful for identifying and qualifying the risk that a particular exploit will occur in a system [McGraw, 2006]

attacker is any person who attempts to perform a malicious action against a system [Tittel et al., 2004]

authentication is the process of verifying or testing that the identity claimed by a subject is valid [Tittel et al., 2004]

authorization is a process that ensures that the requested activity or object access is possible given the rights and privileges assigned to the authenticated identity (*i.e.* subject) [Tittel et al., 2004]

availability is the ability to ensure users have timely and reliable access to their information assets [Wylder, 2003]

buffer overflow is a vulnerability that can cause a system to crash or allow the user to execute commands and gain access to the system outside what are normally allowed [Tittel et al., 2004]

bug is a mismatch between implementation and specification [Møller, 2003]

bytecode in the context of Java systems, is the intermediate code produced (compiled) from the source code by a Java compiler [Rankl & Effing, 2003]

C-APDU is a command sent from a terminal to a smart card, consisting of a command header and an optional command body [Rankl & Effing, 2003]

cardlet is another name for an applet in the realm of smartcards [Rankl & Effing, 2003]

CIA triad is a well-known model of essential security principles consisting of confidentiality, integrity and availability [Tittel et al., 2004]

cipher is a system that hides the true meaning of a message using a variety of techniques to alter and/or rearrange the characters or words of a message to achieve confidentiality [Tittel et al., 2004]

ciphertext is a message that has been encrypted for transmission [Tittel et al., 2004]

command in the realm of smartcard operating systems, is an instruction to the smartcard, in the form of C-APDUs to perform a specific action [Rankl & Effing, 2003]

confidentiality is the ability to prevent of unauthorized use or disclosure of information [Wylder, 2003]

contactless card is a type of smartcard for which energy and data are transferred using electromagnetic fields without any contact with the card [Rankl & Effing, 2003]

countermeasures are actions taken to patch a vulnerability or secure a system against an attack [Tittel et al., 2004]

covert channel is the means by which data can be communicated outside of normal, expected, or detectable methods [Tittel et al., 2004]

cryptographic key is the parameter that individualizes the encryption or decryption process [Rankl & Effing, 2003]

dedicated file is a directory in a smart card file system [Rankl & Effing, 2003]

denial of service is a type of attack that prevents a system from processing or responding to legitimate traffic or requests for resources and objects [Tittel et al., 2004]

digital signature is a method for ensuring a recipient that a message truly came from the claimed sender and that the message was not altered while in transit between the sender and recipient [Tittel et al., 2004]

domain is a realm of trust or a collection of subjects and objects that share a common security policy [Tittel et al., 2004]

EEPROM is a type of non-volatile memory, which is used in smartcards [Rankl & Effing, 2003]

EF is the actual data storage element in a smart card file tree [Rankl & Effing, 2003]

encrypt is the process used to convert a message into ciphertext [Tittel et al., 2004]

encryption is the art and science of hiding the meaning or intent of a communication from recipients not meant to receive it [Tittel et al., 2004]

exploit is an instance of attack on a computer system as a result of one or more vulnerabilities [McGraw, 2006]

flaw is a problem in the design of the software [McGraw, 2006]

full duplex is a data transmission method in which each of the communicating parties can transmit and receive concurrently [Rankl & Effing, 2003]

glitch is a very short voltage dropout or voltage spike [Rankl & Effing, 2003]

GlobalPlatform is an internationally active association founded in 1999 by various smartcard companies to standardize technologies for multiapplication smartcards [Rankl & Effing, 2003]

guideline is a recommendation for things to do or to avoid during software development, described at the semantic level; guidelines are more concrete than principles, but still work at a higher level than rules [McGraw, 2006]

half duplex is a data transmission method in which each of the communicating parties cannot concurrently send and receive data [Rankl & Effing, 2003]

historical risk is a risk that is identified in the course of an actual software development effort; historical risks are good sources for early identification of potential issues, clues for effective mitigation and improvements to the consistency and quality of risk management in the software development process [McGraw, 2006]

integrity is the ability to ensure information is accurate, complete and has not been modified by unauthorized users or processes [Wylder, 2003]

Java is a hardware-independent, object-oriented programming language developed by the Sun Corporation, whose source code is translated by a compiler into standardized bytecode to be interpreted by a virtual machine based on the target hardware and operating system platforms [Rankl & Effing, 2003]

Java Card is a multiapplication smartcard incorporating the Java Card operating system, which can manage and run programs written in Java [Rankl & Effing, 2003]

MF is the root directory of the file tree (a special type of DF) and is automatically selected after the smart card has been reset [Rankl & Effing, 2003]

microcontroller is a self-contained and fully functional computer on a single chip, consisting of a CPU, volatile memory, non-volatile memory and suitable interfaces for off-chip communications [Rankl & Effing, 2003]

non-volatile memory is a type of memory that retains its content even in the absence of power [Rankl & Effing, 2003]

plaintext is a message that has not been encrypted [Tittel et al., 2004]

polling refers to periodic program-driven querying of an input channel in order to detect an incoming message [Rankl & Effing, 2003]

principle is a statement of general security wisdom, stemming from real-world experience of building secure systems; principles exist at a philosophical level and, therefore, are abstract and tend to be generic in nature [McGraw, 2006]

proactivity is a smartcard transaction mechanism that allows a smartcard to independently initiate actions in the terminal (circumventing the normal master–slave relationship between the terminal and the smartcard), realized by requiring the terminal to periodically poll the smartcard [Rankl & Effing, 2003]

R-APDU is a reply sent by a smartcard in response to a C-APDU received from a terminal command [Rankl & Effing, 2003]

RAM is a type of volatile memory, which is used in smart cards as working memory [Rankl & Effing, 2003]

reset refers to restoring a computer (in this case, a smartcard) to a clearly defined initial state [Rankl & Effing, 2003]

Riscure is a company specializing in the security analysis of smart cards and embedded devices

risk is the likelihood that any specific threat will exploit a specific vulnerability to cause harm to an asset [Tittel et al., 2004]

risk analysis is an element of risk management that includes analyzing an environment for risks, evaluating each risk as to its likelihood of occurring and cost of damage, assessing the cost of various countermeasures for each risk, and creating a cost/benefit report for safeguards to present to upper management [Tittel et al., 2004]

risk management is a detailed process of identifying factors that could damage or disclose data, evaluating those factors in light of data value and countermeasure cost, and implementing cost-effective solutions for mitigating or reducing risk [Tittel et al., 2004]

ROM is a type of non-volatile memory, which is used in smart cards to store programs and static data as its content cannot be altered [Rankl & Effing, 2003]

rule is a more concrete version of a guideline, described at the syntax level and, thus, language-specific [McGraw, 2006]

safeguard is anything that removes a vulnerability or protects against one or more specific threats [Tittel et al., 2004]

serial data transmission is a type of data transmission in which individual data bits are sent sequentially along a data line [Rankl & Effing, 2003]

SIM is a GSM-specific smartcard which bears the identity of the subscriber, and its primarily function is to secure the authenticity of the mobile station with respect to the network [Rankl & Effing, 2003]

SIM (Application) Toolkit is a specification that allows the SIM to assume an active role in controlling the mobile telephone, forming the basis for most supplementary applications in mobile telephones [Rankl & Effing, 2003]

smartcard is a card containing a microcontroller with a CPU, volatile and non-volatile memory [Rankl & Effing, 2003]

software defects are problems in software which include both implementation bugs and design flaws McGraw [2006]

software security is the engineering of software so that it continues to function correctly under malicious attack McGraw [2008]

synchronous data transmission is a form of data transmission in which data transmission depends on a predefined timing reference (*e.g.* clock signal applied to the chip) Rankl & Effing [2003]

terminal is a device, possibly having a keypad and display, that provides electrical power to the smartcard and enables it to exchange data [Rankl & Effing, 2003]

threat is a potential occurrence that may cause an undesirable or unwanted outcome on an organization or to a specific asset [Tittel et al., 2004]

TOE is the IT system to be evaluated [Rankl & Effing, 2003]

transmission protocol in the smartcard world, is the mechanisms used for transmitting and receiving data between a terminal and a smartcard [Rankl & Effing, 2003]

UICC is a smartcard having a smartcard operating system in accordance with ISO/IEC 7816 that is optimized for telecommunications applications [Rankl & Effing, 2003]

vulnerability is the result of a software defect that can be used by an attacker to do malicious actions that affect the security of a computer system [McGraw, 2006]

Abbreviations

2G	Second Generation
3G	Third Generation
ADF	Application Data File
AFSCM	Association Française du Sans Contact Mobile
AID	Application Identifier
APDU	Application Protocol Data Unit
API	Application Programming Interface
AST	Abstract Syntax Tree
ATR	Answer to Reset
C-APDU	Command APDU
CAD	Card Acceptance Device
CIA	Confidentiality, Integrity and Availability (*see glossary*: CIA triad)
CLA	Class
CPU	Central Processing Unit
CVM	Cardholder Verification Method
DF	Dedicated File, *also* Directory File (*see glossary*)
EEPROM	Electrically Erasable Programmable Read Only Memory
EF	Elementary File
ETSI	European Telecommunications Standards Institute
GP	GlobalPlatform
GSM	Global System for Mobile Communications
HP	Hewlett-Packard
IBM	International Business Machines Corporation
INS	Instruction
ISO	International Organization for Standardization
IT	Information Technology
ITS4	It's the Software Stupid Security Scanner
JCDK	Java Card Development Kit
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JVM	Java Virtual Machine
MF	Master File

MNO Mobile Network Operator

OTA Over the Air

PIN Personal Identification Number
PPS Protocol Parameter Selection

R-APDU Response APDU
RAM Random Access Memory
RMI Remote Method Invocation
ROM Read Only Memory

SAT SIM Application Toolkit, *also* STK
SCA Source Code Analyzer
SCA Source Code Analysis
SD Security Domain
SIM Subscriber Identity Module
SMS Short Message Service
STK SIM Application Toolkit, *also* SAT

TLV Tag-Length-Value, *also* Type-Length-Value
TOE Target of Evaluation
TPDU Transmission Protocol Data Unit
TS Technical Specification

UICC Universal Integrated Circuit Card

VM Virtual Machine

Bibliography

AFSCM (2010). NFC cardlet development guidelines, release 1.1. Association Française du Sans Contact Mobile Specifications.

AFSCM (2012). NFC cardlet development guidelines, release 2.2. Association Française du Sans Contact Mobile Specifications.

Almaliotis, V., Loizidis, A., Katsaros, P., Louridas, P., & Spinellis, D. (2008). Static program analysis for Java Card applets. In G. Grimaud, & F.-X. Standaert (Eds.) *Smart Card Research and Advanced Applications*, vol. 5189 of *Lecture Notes in Computer Science*, (pp. 17–31). Springer Berlin / Heidelberg.

Alpern, B., & Schneider, F. B. (1985). Defining liveness. *Information Processing Letters*, *21*(4), 181 – 185.
URL http://www.sciencedirect.com/science/article/pii/0020019085900560

Armorize (2012). Codesecure v4: Leader in static code analysis. Online. Accessed June 24, 2012.
URL http://www.armorize.com/codesecure/

Artho, C., & Biere, A. (2001). Applying static analysis to large-scale, multi-threaded java programs. In *Proceedings of the 13th Australian Conference on Software Engineering*, ASWEC '01, (pp. 68–). Washington, DC, USA: IEEE Computer Society.
URL http://dl.acm.org/citation.cfm?id=872024.872575

Ayewah, N., & Pugh, W. (2008). A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*, DEFECTS '08, (pp. 1–5). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/1390817.1390819

Ayewah, N., & Pugh, W. (2010). The Google FindBugs fixit. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, (pp. 241–252). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/1831708.1831738

Baca, D., Carlsson, B., & Lundberg, L. (2008). Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, (pp. 79–88). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/1375696.1375707

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A., & Nutt, R. (1957). The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, IRE-AIEE-ACM '57 (Western), (pp. 188–198). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/1455567.1455599

Balasubramanian, S. (2008). *Strides Towards Better Application Security*. Master's thesis, Utah State University.

Beckert, B., & Mostowski, W. (2003). A program logic for handling javacard's transaction mechanism. In *Proceedings of the 6th international conference on Fundamental approaches to software engineering*, FASE'03, (pp. 246–260). Berlin, Heidelberg: Springer-Verlag.
URL http://dl.acm.org/citation.cfm?id=1762980.1763005

Benenson, Z., Freiling, F. C., Holz, T., Kesdogan, D., & Penso, L. D. (2006). Safety, liveness, and information flow: Dependability revisited. ARCS 2006 Workshops, Frankfurt/Main, Germany.

Bezakova, I., Pashko, O., & Surendran, D. (2000). Smartcards survey. Online. Accessed July 23, 2012.
URL http://people.cs.uchicago.edu/~dinoj/smartcard/

BITS (2012). Software assurance framework. Whitepaper BITS/The Financial Services Roundtable.
URL http://www.bits.org/publications/security/BITSSoftwareAssurance0112.pdf

Black, P. E., Kass, M., Koo, M., & Fong, E. (2011). Source code security analysis tool functional specification version 1.1. NIST Special Publication 500-268 v1.1.

Bosworth, R. (2011). Java 1.6 grammar, version: RNB 28th March 2011 20:03.
URL http://www.cmis.brighton.ac.uk/staff/rnb/bosware/javaSyntax/rules.html

Breunesse, C. B., no, N. C., Huisman, M., & Jacobs, B. (2004). Formal methods for smart cards: An experience report.

Burdy, L., Requet, A., Lanet, J. L., & Vigie, L. (2003). Java applet correctness: a developer-oriented approach. In *In Proc. Formal Methods Europe*, (pp. 422–439). Springer.

Cadonau, J. (2008). OTA and secure SIM lifecycle management. In K. E. Mayes, & K. Markantonakis (Eds.) *Smart Cards, Tokens, Security and Applications*, chap. 11, (pp. 257–275). Springer US.

Cataño, N., & Huisman, M. (2002). Formal specification and static checking of gemplus' electronic purse using esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, FME '02, (pp. 272–289). London, UK, UK: Springer-Verlag.
URL http://dl.acm.org/citation.cfm?id=647541.730158

Cesare, S., & Xiang, Y. (2012). *Software Similarity and Classification*, chap. 4: Formal Methods of Program Analysis, (pp. 35–48). Springer.

Checkmarx (2012a). Cxsuite: Comprehensive application security. Online. Accessed June 22, 2012.
URL http://www.checkmarx.com/Products.aspx?id=3

Checkmarx (2012b). *CxSuite User Manual*.

Chess, B. (2009). RATS: a rough auditing tool for security. Online. Accessed August 19, 2012.
URL http://code.google.com/p/rough-auditing-tool-for-security/

Chess, B., & West, J. (2007). *Secure Programming with Static Analysis*. Addison-Wesley Professional, first ed.

Christopher, C. N. (2006). Evaluating static analysis frameworks. Analysis of Software Artifacts, Carnegie Mellon University.

Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1983). Automatic verification of finite state concurrent system using temporal logic specifications: a practical approach. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, (pp. 117–126). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/567067.567080

Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, *8*(2), 244–263.
URL http://doi.acm.org/10.1145/5397.5399

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2001). Progress on the state explosion problem in model checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, (pp. 176–194). London, UK, UK: Springer-Verlag.
URL http://dl.acm.org/citation.cfm?id=647348.724453

Cousot, P. (1978). *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. Ph.D. thesis, Grenoble University.

Cousot, P. (2005). An informal overview of abstract interpretation. Lecture notes on MIT's Course 16.399: "Abstract interpretation", 2005.

Cousot, P. (2008). Abstract interpretation in a nutshell.
URL http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Cousot, P., & Cousot, R. (1992). Abstract interpretation frameworks. *Journal of Logic and Computation*, *2*, 511–547.

Coverity (2012). Coverity static analysis. Online. Accessed June 24, 2012.
URL http://www.coverity.com/products/static-analysis.html

de Leeuw, K., & Bergstra, J. (Eds.) (2007). *The History of Information Security: A Comprehensive Handbook*. Elsevier B.V.

Deshmukh, A. (2011). Smart card communication using pci mcus. Application note Microchip Technology Inc.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, *18*, 453–457.

DIN (2010). A survey of card standards. Tech. rep., DIN German Institute of Standardization.

D'Silva, V. (2009). Tales from verification history.

Evans, D., Guttag, J., Horning, J., & Tan, Y. M. (1994). LCLint: a tool for using specifications to check code. *SIGSOFT Softw. Eng. Notes*, *19*(5), 87–96.
URL http://doi.acm.org/10.1145/195274.195297

Evans, D., & Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Softw.*, *19*(1), 42–51.
URL http://dx.doi.org/10.1109/52.976940

Feiman, J., & MacDonald, N. (2010). Magic quadrant for static application security testing. Gartner RAS Core Research Note G00208743.

Fetzer, J. H. (1988). Program verification: the very idea. *Communications of the ACM*, *31*(9), 1048–1063.
URL http://doi.acm.org.janus.libr.tue.nl/10.1145/48529.48530

FindBugs (2012). Findbugs fact sheet. Online. Accessed June 24, 2012.
URL http://findbugs.sourceforge.net/factSheet.html

Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., & Stata, R. (2002). Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, (pp. 234–245). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/512529.512558

Floyd, R. W. (1967). Assigning meanings to programs. *Mathematical aspects of computer science*, *19*(19-32), 1.

Fortify, H. (2012a). *HP Fortify Software Security Center v3.50 System Requirements*.

Fortify, H. (2012b). HP Fortify static code analyzer (SCA). Online. Accessed June 20, 2012.
URL http://www.hpenterprisesecurity.com/products/hp-fortify-software-security-center/hp-fortify-static-code-analyzer/

Fortify, H. (2012c). *HP Fortify Static Code Analyzer v3.50 Custom Rules Guide*.

Fortify, H. (2012d). *HP Fortify Static Code Analyzer v3.50 User Guide*.

Gadyatskaya, O., Lostal, E., & Massacci, F. (2011). Load time security verification. In *Proceedings of the 7th international conference on Information Systems Security*, ICISS'11, (pp. 250–264). Berlin, Heidelberg: Springer-Verlag.
URL http://dx.doi.org/10.1007/978-3-642-25560-1_17

Gärtner, F. C. (2002). Revisiting liveness properties in the context of secure systems. Tech. rep., École Polytechnique Fédérale de Lausanne (EPFL).

Gemalto (2008). The review.

Gemalto (2009). Java Card™ & STK applet development guidelines, version 2.0. Gemalto WG.GGS.4.0042.

Giesen, D. (1998). Philoshophy and practical implementation of static analyzer tools. QA Systems Technologies B.V. Whitepaper.

Girard, P., & Lanet, J. (1999). Java Card or how to cope with the new security issues raised by Open Cards. In *In Gemplus Developer Conference '99*, (pp. 21–22).

GlobalPlatform (2006). Card specification version 2.2.

Goldstine, H. H., & von Neumann, J. (1947). Planning and coding of problems for an electronic computing instrument. Tech. rep., Institute for Advanced Study.

Gosling, J., Joy, B., Steele, G., & Bracha, G. (2005). *The Java™ Language Specification, Third Edition*. Addison-Wesley.

GrammaTech (2012). Codesonar static analysis tool overview. Online. Accessed June 24, 2012.
URL http://www.grammatech.com/products/codesonar/overview.html

Hambartsumyan, H. (2011). *Precise XSS Detection with Static Analysis using String Analysis*. Master's thesis, Technische Universiteit Eindhoven.

Harris, L. C., & Miller, B. P. (2005). Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, *33*(5), 63–68.
URL http://doi.acm.org/10.1145/1127577.1127590

Harrold, M. J., Rothermel, G., & Orso, A. (2005). Representation and analysis of software. Lecture Notes.

Harrold, M. J., Rothermel, G., & Sinha, S. (1998). Computation of interprocedural control dependence. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '98, (pp. 11–20). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/271771.271780

Henzinger, T. (2006). Model checking, theorem proving, and abstract interpretation: The convergence of formal verification technologies. John von Neumann Computer Society International Symposium.
URL http://www.jaist.ac.jp/~bjorner/ae-is-budapest/talks/Sept19pm2_Henzinger.pdf

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*, 576–585.

Hovemeyer, D. (2004). Using findbugs. Lecture Notes University of Maryland.
URL http://www.cs.umd.edu/class/fall2004/cmsc433/lectures/find-bugs-2up.pdf

Hovemeyer, D., & Pugh, W. (2004). Finding bugs is easy. *SIGPLAN Not.*, *39*(12), 92–106.
URL http://doi.acm.org/10.1145/1052883.1052895

Hovemeyer, D. H., & Pugh, W. W. (2012). *FindBugs Manual*. University of Maryland.

Howard, M. (2000). Inside the secure windows initiative. Online. Accessed July 21, 2012.
URL http://technet.microsoft.com/en-us/library/cc723542.aspx

Howard, M., & LeBlanc, D. (2002). *Writing Secure Code*. Microsoft, second ed.

Howard, M., & Lipner, S. (2006). *The Security Development Lifecycle*. Microsoft.

IBM (2012). Rational AppScan Source Edition. Online. Accessed May 20, 2012.
URL http://www-01.ibm.com/software/rational/products/appscan/source/

ISO-7816 (1987–2007). Identification cards - integrated circuit cards with contacts.

Jacobs, B., Marché, C., & Rauch, N. (2004). Formal verification of a commercial smart card applet with multiple tools. In *AMAST'04*, (pp. 241–257).

Jaspan, C., Chen, I.-C., & Sharma, A. (2007). Understanding the value of program analysis tools. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, (pp. 963–970). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/1297846.1297964

Johnson, S. C. (1978). Lint, a c program checker. In *Comp. Sci. Tech. Report*, (pp. 78–1273).

Jurgensen, T. M., & Guthery, S. B. (2002). *Smart Cards: the Developer's Toolkit*. Macmillan Technical Publishing.

Klocwork (2012). Klocwork insight: On-the-fly source code analysis. Online. Accessed June 24, 2012.
URL http://www.klocwork.com/products/insight/index.php

Knizhnik, K., & Artho, C. (2011). *Jlint 3.1.2 manual: Java program checker*.

Ko, H., & Caytiles, R. D. (2011). A review of smartcard security issues. *Journal of Security Engineering*, (pp. 359–370).

Kratkiewicz, K. J. (2005). *Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code*. Master's thesis, Harvard University.

Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, *SE-3*, 125–143.

LaToza, T. D., & Myers, B. A. (2011). Designing useful tools for developers. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, PLATEAU '11, (pp. 45–50). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/2089155.2089166

Livshits, V. B., & Lam, M. S. (2005). Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, SSYM'05, (pp. 18–18). Berkeley, CA, USA: USENIX Association.
URL http://dl.acm.org/citation.cfm?id=1251398.1251416

Logozzo, F., & Fähndrich, M. (2008). On the relative completeness of bytecode analysis versus source code analysis. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, CC'08/ETAPS'08, (pp. 197–212). Berlin, Heidelberg: Springer-Verlag.
URL http://dl.acm.org/citation.cfm?id=1788374.1788393

Loizidis, A., Vasilios, A., & Panagiotis, K. (2011). Static program analysis of multi-applet JavaCard applications. In *Software Engineering for Secure Systems: Industrial & Research Perspectives*, (pp. 286–304). IGI Global.
URL http://delab.csd.auth.gr/papers/SESS2011lak.pdf

Malloy, B., & Voas, J. (2004). Programming with assertions: a prospectus [software development]. *IT Professional*, *6*(5), 53 – 59.

Marche, C., Mohring, P. C., & Urbain, X. (2004). The Krakatoa Tool for Certification of Java/JavaCard Programs Annotated in JML. *Journal of Logic and Algebraic Programming*, *58*(1-2), 89–106.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.7.4458

Markantonakis, K. (2008). Multi application smart card platforms and operating systems. In K. E. Mayes, & K. Markantonakis (Eds.) *Smart Cards, Tokens, Security and Applications*, chap. 3, (pp. 51–83). Springer US.

Markantonakis, K., & Mayes, K. (2008). Smart cards for banking and finance. In K. E. Mayes, & K. Markantonakis (Eds.) *Smart Cards, Tokens, Security and Applications*, chap. 5, (pp. 115–138). Springer US.

Mayes, K. (2008). An introduction to smart cards. In K. E. Mayes, & K. Markantonakis (Eds.) *Smart Cards, Tokens, Security and Applications*, chap. 1, (pp. 1–25). Springer US.

Mayes, K., & Evans, T. (2008). Smart cards for mobile communications. In K. E. Mayes, & K. Markantonakis (Eds.) *Smart Cards, Tokens, Security and Applications*, chap. 4, (pp. 85–113). Springer US.

McGraw, G. (2004). Software security. In *Building Security In*. IEEE Security and Privacy.

McGraw, G. (2006). *Software Security: Building Security In*. Addison Wesley Professional.

McGraw, G. (2008). Automated code review tools for security. *Computer*, *41*, 92–95.

Mead, N. R., & McGraw, G. (2005). A portal for software security. In *Building Security In*. IEEE Security and Privacy.

Møller, A. (2003). An introduction to analysis and verification of software. University of Aarhus. URL http://cs.au.dk/~amoeller/talks/verification.pdf

Monniaux, D. (2011). Abstract interpretation. First Summer School on Formal Techniques, Menlo College.

Nanda, M. G., Gupta, M., Sinha, S., Chandra, S., Schmidt, D., & Balachandran, P. (2010). Making defect-finding tools work for you. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, (pp. 99–108). New York, NY, USA: ACM. URL http://doi.acm.org/10.1145/1810295.1810310

Nielson, F., Nielson, H. R., & Hankin, C. (1999). *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Nielson, H. R., & Nielson, F. (1992). *Semantics with applications: a formal introduction*. New York, NY, USA: John Wiley & Sons, Inc.

Okun, V., Guthrie, W. F., Gaucher, R., & Black, P. E. (2007). Effect of static analysis tools on software security: preliminary investigation. In *Proceedings of the 2007 ACM workshop on Quality of protection*, QoP '07, (pp. 1–5). New York, NY, USA: ACM. URL http://doi.acm.org/10.1145/1314257.1314260

Ouimet, M. (2008). Formal software verification: Model checking and theorem proving. Tech. rep., Massachusetts Institute of Technology.

Padua, D. (2000). The fortran i compiler. *Computing in Science and Engineering*, *2*, 70–75.

Parasoft (2011). *Parasoft Jtest 9.2 User's Guide*.

Parasoft (2012). Parasoft Jtest: Java testing, static analysis, code review. Online. Accessed May 20, 2012. URL http://www.parasoft.com/jsp/products/jtest.jsp?itemId=14

Pistoia, M., Chandra, S., Fink, S. J., & Yahav, E. (2007). A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, *46*, 265–288.

PMD (2012a). Pmd: How it works. Online. Accessed June 24, 2012. URL http://pmd.sourceforge.net/pmd-5.0.0/howitworks.html

PMD (2012b). Welcome to PMD. Online. Accessed June 24, 2012. URL http://pmd.sourceforge.net/pmd-5.0.0/

Potter, B., & McGraw, G. (2004). Software security testing. In *Building Security In*. IEEE Security and Privacy.

Queille, J.-P., & Sifakis, J. (1982). Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, (pp. 337–351). London, UK, UK: Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=647325.721668

Rankl, W., & Effing, W. (2003). *Smart Card Handbook*. Chichester, West Sussex, England: John Wiley & Sons, Ltd, 3rd ed.

Rice, H. G. (1953). Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, *74*(2), 358–366.
URL http://www.jstor.org/stable/1990888

Ruiu, D. (2006). Learning from information security history. *Security Privacy, IEEE*, *4*(1), 77 – 79.

Rushby, J. (1993). Critical system properties: Survey and taxonomy. Tech. rep., SRI International.

Rutar, N., Almazan, C. B., & Foster, J. S. (2004). A comparison of bug finding tools for Java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, (pp. 245–256). Washington, DC, USA: IEEE Computer Society.
URL http://dx.doi.org/10.1109/ISSRE.2004.1

Salus, P. H. (1998). Net insecurity: Then and now (1969-1998). Delivered at SANE '98. Note available online. Accessed June 24, 2012.
URL http://www.sane.nl/events/sane98/aftermath/salus.html

Schneider, F. B. (1987). Decomposing properties into safety and liveness. Tech. rep., Ithaca, NY, USA.

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., & Sommerlad, P. (2005). *Security Patterns: Integrating Security and Systems Engineering*. Wiley.

Schwartzbach, M. I. (2008). Lecture notes on static analysis. BRICS, Department of Computer Science, University of Aarhus, Denmark.

Sifakis, J. (1982). A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, *18*(3), 227 – 258.
URL http://www.sciencedirect.com/science/article/pii/0304397582900676

Smelt, G. (2011). *Programming Web Applications Securely*. Master's thesis, Radboud University of Nijmegen.

Sotirov, A. I. (2005). *Automatic Vulnerability Detection using Static Source Code Analysis*. Master's thesis, The University of Alabama.

SSD Team (2011). Smart analyzer. Online. Accessed July 8, 2012.
URL http://secinfo.msi.unilim.fr/smart-analyzer/

Sun Microsystems (1998). *Java Card Applet Developer's Guide*, 1.10 ed.

Tittel, E., Stewart, J. M., & Chapple, M. (2004). *CISSP: Certified Information Systems Security Professional*. SYBEX Inc., 2nd ed.

Townley, A. S. (2005). Pragmatic security: Making the most of what you have. *Information Security Bulletin*, *10*, 331–338.

Trevisan, L. (2009). Notes on Rice's theorem. Lecture Notes U.C. Berkeley.

Tulloch, M. (2003). *Microsoft Encyclopedia of Security*. Microsoft Press.

Tunstall, M. (2008). Smart card security. In K. E. Mayes, & K. Markantonakis (Eds.) *Smart Cards, Tokens, Security and Applications*, chap. 9, (pp. 195–228). Springer US.

Turing, A. (1949). Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, (pp. 67–69). MIT Press.
URL http://dl.acm.org/citation.cfm?id=94938.94952

van den Berg, J., & Jacobs, B. (2001). The loop compiler for java and jml. (pp. 299–312). Springer.

Veracode (2012). Source code security analyzer. Online. Accessed June 24, 2012.
URL http://www.veracode.com/security/source-code-security-analyzer

Vétillard, E., & Marlet, R. (2003). Enforcing portability and security policies on Java Card applications. In *e-Smart '03*.

Viega, J., Bloch, J., Kohno, Y., & McGraw, G. (2000). ITS4: a static vulnerability scanner for c and c++ code. In *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*, (pp. 257 –267).

Viega, J., & McGraw, G. (2001). *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley.

Wagner, D. A. (2000). *Static analysis and computer security: New techniques for software assurance*. Ph.D. thesis, University of California at Berkeley.

Wagner, S., Jürjens, J., Koller, C., & Trischberger, P. (2005). Comparing bug finding tools with reviews and tests. In *Proceedings of the 17th IFIP TC6/WG 6.1 international conference on Testing of Communicating Systems*, TestCom'05, (pp. 40–55). Berlin, Heidelberg: Springer-Verlag.
URL http://dx.doi.org/10.1007/11430230_4

Waite, G., & Mayes, K. (2008). Application development environments for Java and SIM toolkit. In K. E. Mayes, & K. Markantonakis (Eds.) *Smart Cards, Tokens, Security and Applications*, chap. 10, (pp. 229–255). Springer US.

Ware, M. S. (2008). *Writing Secure Java Code: A Taxonomy of Heuristics and an Evaluation of Static Analysis Tools*. Master's thesis, James Madison University.

Ware, M. S., & Fox, C. J. (2008). Securing Java code: heuristics and an evaluation of static analysis tools. In *Proceedings of the 2008 workshop on Static analysis*, SAW '08, (pp. 12–21). New York, NY, USA: ACM.
URL http://doi.acm.org/10.1145/1394504.1394506

Wheeler, D. A. (2007). Flawfinder. Online. Accessed August 19, 2012.
URL http://sourceforge.net/projects/flawfinder/

Whittaker, J., & Thompson, H. (2003). *How to Break Software Security*. Addison-Wesley.

Witteman, M. (2002). Advances in smartcard security. *Information Security Bulletin*, (pp. 11–22).

Witteman, M. (2012). Secure application programming in the presence of side channel attacks. Riscure B.V. Whitepaper.

Wögerer, W. (2005). A survey of static program analysis techniques. Technische Universität Wien.

Wylder, J. (2003). *Strategic information security*. Auerbach Publications.

Young, M., & Pezze, M. (2008). *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons.

Zakinthinos, A. (1996). *On The Composition Of Security Properties*. Ph.D. thesis, University of Toronto.

Zanero, S. (2002). Smart card content security: Defining "tamperproof" for portable smart media. Version 1.0.