ORB ACADEMIC PUBLISHER

Int. J. Comput. Sci. Theor. App., 2014, vol. 2, no. 2., p. 23-34
Available online at www.orb-academic.org

International Journal of
**Computer Science:
Theory and Application**
ISSN: 2336-0984

Towards the Verification and Validation of Software Security Properties Using Static Code Analysis

Zeineb Zhioua[1,⋆], Stuart Short[1], Yves Roudier[2]

[1] *SAP Labs France.*
[2] *EURECOM, Sophia Antipolis, France.*
Email: zeineb.zhioua@sap.com, stuart.short@sap.com, yves.roudier@eurecom.fr
⋆ *Corresponding author.*

Abstract
Developing and delivering secure software is a challenging task, that gets even harder when the developer tries to adhere to both application and organization-specific security requirements. Different approaches have been proposed to facilitate this task, such as code analysis that aims at detecting flaws in the developed software before it is released and deployed to customer. This paper discusses a number of static code analysis approaches and presents different code analysis tools adopting each a specific analysis technique. These tools are evaluated against a sample code illustrating different security challenges that can be addressed using an approach that helps detecting security properties. The latter can be transformed into abstract security policies that can be validated against explicit security requirements. This would help the developer throughout the software development lifecycle and to ensure the compliance with security specifications.

Keywords
Static Analysis — Code Analysis Tools — Security Properties — Program Modeling.
© *2014 by Orb Academic Publisher. All rights reserved.*

1. Introduction

[1] In the software development lifecyle, flaws and errors can be introduced during the different phases, from design to development. These undetected errors can turn into security vulnerabilities at run-time, and can be exploited by intruders who introduce serious damages to the software critical resources. Certain flaws can be detected easily, while others can remain undetected for different reasons, namely they emerge rarely or never. Undetected flaws ultimately entail maintenance costs, in addition to losses incurred by potential attacks. The use of code analysis and security testing techniques are now well established to find out such flaws.

In the development lifecycle, developers resort to the usage of static code analysis tools to assist them in avoiding these flaws, and help them develop and produce secure and safe software. However, these tools fall short in verifying the adherence of the developed software to application and organization security requirements. Retrieving and validating security properties in software present a challenging problem that different researchers have tried to tackle based on a variety of security analysis tools. Each of the tools adopts a specific analysis technique such as lexical analysis, syntactic and semantic analysis, is focused on specific security properties and requires human intervention to different degrees.

[1]This work is an extended version of [54] published in *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*

Developers may also rely on the dynamic analysis that offers the possibility to take action in the presence of dynamic input, which constitutes one of the strengths of this approach. It often considers the program as a black-box, and analyzes its run-time behavior usually without any access to the source code. Dynamic analysis reports failures at the instant they occur, and provides details allowing the developer/tester to make the required corrections. One of the shortcomings of this approach is that it requires a sufficient number of test cases and is quite time-consuming, yet it cannot ensure an automatic verification or a complete coverage of the test cases space of the analyzed program.

In comparison, static analysis in all its forms ensures a complete coverage of the program branches [4], used APIs, program dependencies, or configuration files explored. Static analysis refers to different methodologies, including model checking and model provers, to verify the execution paths of a program without actually executing it [2]. Unlike manual review, which relies on the tedious examination of sequences of the concrete or symbolic execution of a program, static code analyzers can capture comprehensive and accurate models of the software, like for instance an abstract representation of all the execution paths, which test-case execution falls short to cover.

We present the research methodology we have conducted with the objective of covering the main topics of our interest. The starting point was the modeling and derivation of security properties that different researches have dealt with, and proposed different approaches to fill the gap in the representation and validation of security properties. To this aim, they make use of formal methods, such as static analysis techniques. This led us to a second topic, that is the static code analysis approaches and techniques used to securing software. We discuss in this paper different static analysis approaches and what are the challenges ahead in order to assess different types of security properties, notably complex ones defined at the application level in close relationship with the design.

In order to deepen our analysis, we collected researches and publications that have dealt with security properties modeling and validation, and the static analysis approaches used to this aim. For more concrete results, we selected and tested a number of static code analysis tools, covering those used to find security vulnerabilities, as well as those used detect security properties in the source code. This led us to the comparison between the selected tools ‘used approaches, with the objective of determining where major innovations can be achieved; this is depicted in the proposed approach section.

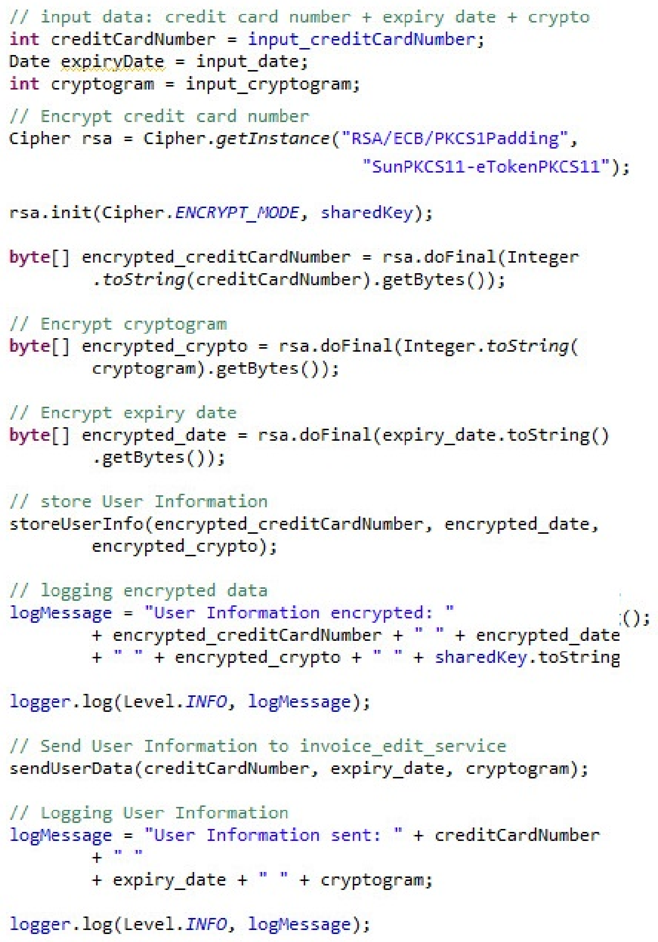
This paper is organized as follows: Section 2 highlights common security issues based on a motivating example. Section 3 discusses different definitions of software security properties that can be found in the literature. Section 4 then briefly surveys different static code analysis techniques and program representation models that are used today, while Section 5 provides further details on a selection of five static code analysis tools representative of current uses. Section 6 discusses the applicability of such tools, as well as experimental results with respect to the motivating example. Finally, section 7 concludes the paper and discusses future work.

2. Motivating Example

With the objective of illustrating further the concepts discussed in the previous section, we present a sample code (Figure 1 and Figure 2) in which we have injected a number of security flaws and introduced the notion of security properties.

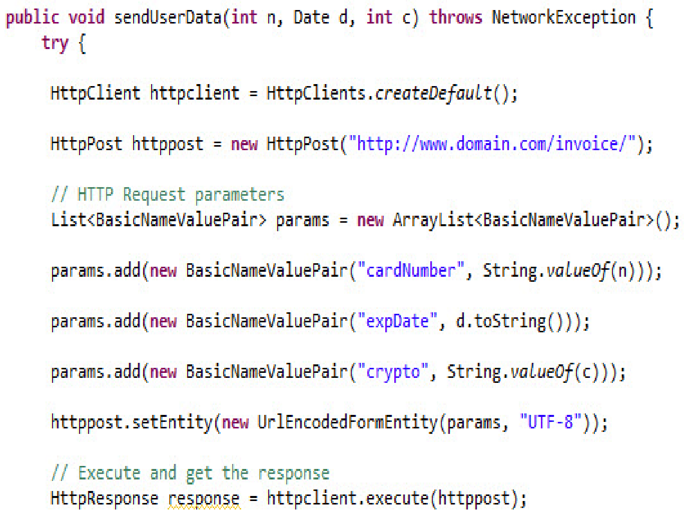
In the first three lines of the sample code (Figure 1), are declared three variables that are assigned the user’s payment information provided as input. The credit card number, the 3-digit cryptogram and the expiry date are then encrypted using a third-party encryption library. The encrypted data is afterwords stored in the database (sendUserData method) for different reasons, such as the payment made.

This may provide assurance about the confidentiality of these critical data in storage; and here we mean that these sensitive data have to be kept secret when stored and persisted in



```
// input data: credit card number + expiry date + crypto
int creditCardNumber = input_creditCardNumber;
Date expiryDate = input_date;
int cryptogram = input_cryptogram;

// Encrypt credit card number
Cipher rsa = Cipher.getInstance("RSA/ECB/PKCS1Padding",
                                         "SunPKCS11-eTokenPKCS11");

rsa.init(Cipher.ENCRYPT_MODE, sharedKey);

byte[] encrypted_creditCardNumber = rsa.doFinal(Integer
        .toString(creditCardNumber).getBytes());

// Encrypt cryptogram
byte[] encrypted_crypto = rsa.doFinal(Integer.toString(
        cryptogram).getBytes());

// Encrypt expiry date
byte[] encrypted_date = rsa.doFinal(expiry_date.toString()
        .getBytes());

// store User Information
storeUserInfo(encrypted_creditCardNumber, encrypted_date,
        encrypted_crypto);

// logging encrypted data
logMessage = "User Information encrypted: "                        ();
        + encrypted_creditCardNumber + " " + encrypted_date
        + " " + encrypted_crypto + " " + sharedKey.toString

logger.log(Level.INFO, logMessage);

// Send User Information to invoice_edit_service
sendUserData(creditCardNumber, expiry_date, cryptogram);

// Logging User Information
logMessage = "User Information sent: " + creditCardNumber
        + " "
        + expiry_date + " " + cryptogram;

logger.log(Level.INFO, logMessage);
```

Figure 1. Sample code.



```
public void sendUserData(int n, Date d, int c) throws NetworkException {
   try {

     HttpClient httpclient = HttpClients.createDefault();

     HttpPost httppost = new HttpPost("http://www.domain.com/invoice/");

     // HTTP Request parameters
     List<BasicNameValuePair> params = new ArrayList<BasicNameValuePair>();

     params.add(new BasicNameValuePair("cardNumber", String.valueOf(n)));

     params.add(new BasicNameValuePair("expDate", d.toString()));

     params.add(new BasicNameValuePair("crypto", String.valueOf(c)));

     httppost.setEntity(new UrlEncodedFormEntity(params, "UTF-8"));

     // Execute and get the response
     HttpResponse response = httpclient.execute(httppost);
```

Figure 2. SendUserData method.

the database. However, the invocation of the method sendUserData (Figure 2) sends the sensitive information in plain text to an external source (network call: HTTP quey). This is a security breach that automated source code vulnerability detection tools cannot recognize automatically using string-matching, and independently from the application expected security objectives.

Even though the critical data (the assets) are encrypted, the program cannot be deemed as secure or, in this case, ensuring the confidentiality of the payment information on one hand. On the other hand, the action of sending these sensitive information in plain text is a severe vulnerability, that violates the security property "confidentiality in transit", and hence, violates the overall security property "confidentiality".

Integrity and confidentiality of data are classically guaranteed by the implementation of access control mechanisms, assuring that only authorized principals are allowed to access to the data. If an unauthorized party gets access to the sensitive payment information, the property "confidentiality" is then breached.

A common use case consists in using logging functionality for analysis and auditing purposes. As we can see in the sample code, the encrypted data are logged with the encryption key (publicKey), and the payment information are then logged in plain text; this represents a severe security threat, that will only be detected after the software is released to the customer, or even worse, after the flaw is exploited by intruders.

The encryption mechanism may guarantee that the data is kept secret, but can't provide assurance about where and how the data will propagate, where it will be stored, or where it will be sent or processed. This entails the need for controlling information flow using static code analysis. This same idea is emphasized by Andrei Sabelfeld, and Andrew C. Myers [35], who deem necessary to analyze how the information flows through the program. According to the authors, a system is deemed to be secure regarding the property confidentiality, if the system as a whole ensures this property. If we had a security policy that expresses the "confidentiality of user's payment information" requirement, current static code analysis tools will not be able to concretize it or to relate it to the concrete user information variables in the source code. Hence, the code analysis tools will not afford to verify the compliance of the developed program with this security policy. If a known vulnerability is identified, static code analysis tools will only detect the exact location where this flaw occurs, that is, in which line of the source code, but they don't provide the means to back-track the vulnerability and identify the source that led to it.

The main issues we raised in this sample code are related to capabilities of static code analysis tools to:

- define the assets to be protected: we mean by assets the critical resources/variables in the source code
- represent/concretize abstract security properties with respect to the code: in other words, how to map between the abstract security policy and the assets to be protected
- detect the presence of the "confidentiality in storage" of the assets
- detect the bad programming practice "send critical data in plain text"
- back-track the source of the vulnerability: we need to backtrack the security vulnerability and identify its sources
- establish the mapping between the vulnerability and the violated security properties: security vulnerability can be perceived as violation of security property
- detect the bad programming practice "storage/logging of the encryption key with the encrypted data in the same table"
- translate the detected properties into a security policy

3. Security Properties

A number of studies have proposed the use of static code analysis with the objective of establishing the satisfaction of security properties in software implementations. The objectives of such analysis vary widely as security properties definitions are not universal or common, and are understood very diversely as well.

Many authors take it for granted that security properties can be defined universally. In contrast, some authors contend that security properties are highly dependent on the level of abstraction considered and on the application developed. For instance, [20], Antonio Maña and Gimena Pujol [1] classify security properties along several dimensions in emerging open and distributed environments:

Abstract Security Properties (ASP) represent security properties considered over the initial draft of the software architecture during application requirements engineering, and can be formalized. Concrete Security Properties (CSP) map ASPs to security mechanisms or algorithms implemented into software. The same ASP can be proven by different CSPs. ASPs and CSPs can be connected by logical relationships, for instance through the logical implication, and which they term Semantic Security Properties (SSP).

In contrast, Domain Security Properties (DSP) are specifications of security properties generic to a given domain.

ASPs are for instance often considered in relationship with complex access control or usage control model policies introduced into software, especially through language based security approaches. For instance, the JIF framework [50] relies on the static analysis of the information flows of a security policy and the verification of the conformance of the flows implemented with those specified in that policy. However, application level security concerns have not really made their way into mainstream static code analysis techniques, which focus mostly on concrete security properties, and which is sometimes even restricted to code safety analysis.

Code safety can be considered as the most concrete level of abstraction in software for what regards security properties. Practitioners generally look for the absence of exploitable safety vulnerabilities in software. Though this is most generally addressed through security testing, static validation may help. The dual modeling problem introduced by John Wilander et al. [9, 8] for instance outlined the correlation between security properties and bad programming practices that may result in safety issues. Wilander et al. introduce a static analysis methodology based on the detection of security and insecurity patterns. The analysis starts by ruling out the presence of an insecurity pattern in the code. If one such pattern is detected, the analysis then proceeds to check whether faulty behaviors are prevented by a security pattern encompassing the insecurity pattern scope. This means that potential vulnerabilities are foiled by the presence of appropriate security mechanisms. The analysis relies on the mapping of functionalities over code snippets. This mapping may somewhat restrict the expressivity of the analysis in that only simple functions are likely to be automatically recognized in a complex source code. The insecurity and security patterns, together with the mapping between both types of patterns can be regarded as a security policy.

A number of researchers similarly regards security properties as classes of good vs. bad programming practices with respect to a given security policy. For instance, Aris Zakinthinos and E.S. Lee [19] define security properties as the instantiation of a security policy that can be satisfied by more than one single property.

The authors distinguish between security properties regarding high-level (trusted) and low-level (untrusted) users, in such a way that low-level users are not able to make deductions about events generated by the high-level users.

In the same definition provided by John Wilander, security properties can be seen as *trace-based* properties, that is, the property is defined as a set of statements or instructions in a program. The trace-based property holds for a whole system if it is valid for the set of all individual traces in the program, and not for one single trace [46]. This argument works only for trace-based properties.

Another definition [18] considers security properties following a classical IT security policy: such a policy ensures that software "assets" or resources have CIA (confidentiality, integrity, availability) properties, and therefore that their confidentiality cannot be violated, and that they cannot be corrupted, or made unavailable, if so specified. The network security properties of authorization, message confidentiality and integrity, non-repudiation of sending or receipt are also used in distributed software, or in software involving several principals. In the following section, we will explore more the static code analysis and the capabilities it offers in the aim of identifying and validating security properties in source code.

4. Static Code Analysis

In the previous section, we presented and discussed different definitions attributed to security properties in the literature, and most of them were considered in the modeling and representation of security properties that source code analysis tools consider in the analysis approach they adopt.

4.1 Static Code Analysis in the Practice of Software Development

The delivery and deployment of secure software has always been a challenging issue that software vendors try to achieve. Creating more secure software can help reduce security related maintenance and update costs, given the fact that it helps reducing, or even eliminating the sources requiring security corrections.

According to Fabian van den Broek [14], static code analysis can be used in two situations; the first is during testing, and the second during development and before testing.

Many developers rely on the latter, that is testing, to ensure the safety of their programs; but this doesn't guarantee the security of the developed software, given the fact that it aims at verifying the compliance with functional requirements [33]. Hence, functional testing falls short in covering the security aspect of the software under inspection. In addition, it is applied only to executable and not to source code or byte-code, and takes place rather late in the software development process. Unlike testing, static code analysis can be applied to single files or to entire program code, and doesn't require the development to be complete. In the earlier stages of the development process, developers may make mistakes and programming errors that can be detected by compilers, that can, in many cases, provide corrections to the detected flaws, and the development process is still ongoing. However, this principle is not applicable to most security vulnerabilities that can remain undetected. The more a flaw is undiscovered, the greater it costs to fix [5].

Some organizations try to overcome this lack of focus on security by performing Penetration tests [2]. Another approach consists in detecting the security flaws in the source code of the developed software before it is released or even tested. This can be manual, meaning that tests are carried out by human analyst/developer, or automatic, using a code analysis tool. Note that manual code review can be much more time-consuming than automatic code review, specially when the entire source code is to be analyzed, or the code to inspect is large. [37]

From this perspective, automatic static code analysis can be integrated and applied regularly to the Software Development Life cycle. These tools are to be used to complement the manual code review, and not to totally replace it as precised by Jernej et al. [38]. Having all this is mind, we can emphasize on the importance and necessity of the static code analysis in reducing the sources of security issues in the early stages of the software development life cycle, and before it is released to customers.

4.2 Techniques for Static Code Analysis

Static code analyzers are used for different purposes, mainly for bugs and security vulnerabilities detection. They are also used to verify the preserved security properties and for program understanding as well [6]. Control-flow and Data-flow are two of the commonly adopted formal methods for program representations and static analysis without executing it.

4.2.1 Model Checking

Model checking is one of the formal approaches that was first introduced by Steffen and Schmidt [53], and is applicable to programs having finite states, or that can be reduced to finite state. This technique allows the automatic verification of properties on finite-state systems, and operates in two steps: it requires first the model construction, and as a second step the properties specification and modeling.

The model construction consists in transforming the system into a formalism (such as the Kripke structure [11]) accepted by a model checking tool. The modeling may also require a certain level of abstraction in order to eliminate irrelevant details. Model checking requires as a second step the definition and the specification of the properties to be met by the software model subject of the analysis, and is usually given using logical formalism, like for instance the temporal logic. However, once the specification of the requirements is achieved, no human intervention can be performed on the input specification. Most of the Model Checking methods are focused on the Temporal Logic, and were introduced by a number of researches, among them [22], who proposed a Model Checker allowing to verify the compliance of finite state systems to a Temporal Logic specification. The verification is performed by exploring the state space in order to determine whether the specified properties are satisfied or not.

4.2.2 Control-flow analysis

Control-flow analysis is one of the common used techniques for static code analysis. The program control-flow is modeled as a directed Control Flow Graph (CFG), and was first introduced by Frances E. Allen [34]. CFG is a directed graph that is used to represent blocs of code in the the form of nodes, the control dependencies in the form of directed edges, starting with an entry node and concluding with the end point of the program. The CFG construction can be carried out based on an abstract syntax graph representation such as AST (Abstract Syntax Tree) to which control flow information are introduced [12] [13]. The main focus of this technique is to determine how the procedures in a program call each other, as well as to determine which functions are effectively called.

4.2.3 Data-flow analysis

Data-flow analysis, on the other hand, is based on the abstract representation of the analyzed program semantics, and is focused on the extraction of the possible values of data. It aims at representing data dependencies in the source code, and allows to track the effect of input data. It aims also at mapping program's statements with the data-flow. The latter gathers information about the possible set of values [29], and is often performed on the Control Flow Graph. Data-flow analysis has for objective to statically predict the the dynamic behavior of the analyzed program.

4.2.4 Symbolic analysis

Symbolic analysis consists in considering the program variables. According to Wolfgang Wogerer [30], this approach can be seen as a compiler that translates the program being analyzed to an intermediate language, consisting of symbolic expressions and recurrences. This technique is supported by computer algebra systems, that adopt simplification methods to ensure the quality of the output results. The analyzed program consists of three parts: the state, the state condition and the path condition. As for the state, it is composed of a (variable, value) pair. State condition is a logic formula that describes assumptions about the variables values. The path condition, on the other hand, is a logic formula that defines the condition for which the program point is reached. Symbolic analysis is deemed to be useful in transforming unpredictable loops to predictable sequences, and is mainly used for code optimization, performed by compilers.

4.2.5 Information-flow analysis

Controlling how information flows through out a program is of paramount importance when dealing with information security. Information flow is mainly analyzed using dynamic analysis approach, however, static code analysis can be used to approximate the information flow propagation and ensure their security, according to Pistoia et al. [4] The main objective of this approach is to analyze how the information flows through a program [35], and verify that no sensitive information is leaked. The starting point in the information flow analysis is to perform a classification of the variables and method calls in a program with *security levels*: H for high level security (secret, private) and L for low security (public) [45]. This classification makes possible the verification of *non-interference* property stating that high level interactions do not interfere with low level interactions and observations, meaning that it is impossible for an observer to draw conclusions about high (secure) information from public output.

In the information-flow terminology, we distinguish between direct (explicit) and indirect (implicit) information flow, legal and illegal information flow. The direct flow arises from direct data flows, that is explicit data assignments, and the indirect is induced indirectly by branching control flow [48].

4.3 Program Representation Models

Static code analysis operates on an representation of the program to analyze, which is basically an abstraction that exploits and represents the properties of this program. Different representation models have been proposed, and they do depend on the context and the usage needs behind it.

4.3.1 Abstract Syntax Tree

Abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of the program source code. Each node of the tree denotes a construct occurring in the source code. The syntax is ”abstract”, that is, a number of real syntax details has been removed. Abstract syntax trees are data structures widely used in compilers, due to their property of representing the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. Abstract syntax trees are also used in program analysis and program transformation systems. Expressions in AST may be nested or complex, which complicates the analysis that may require simpler representation.

4.3.2 Control Flow Graph

Control flow graphs have been the usual program transformation representing the control flow and easing the determination of the control constraints on which the operations depend. CFG is a directed graph that was first introduced by Frances E. Allen [34], and is used to represent blocs of code in the form of nodes, the control dependencies in the form of directed edges, starting with an entry node and augmented with the end point of the program. In addition to the entry and exit vertices, CFG has two types of vertices; statement, predicate vertices; the former has one successor, and the latter has one true-successor and one false-successor.

The CFG construction can be carried out based on an abstract syntax graph representation such as AST (Abstract Syntax Tree) to which control flow information are introduced [12] [13]. The main focus of this technique is to determine how the procedures in a program call each other, as well as to determine which functions are effectively called.

4.3.3 Data Flow Graph

Data Flow Graph (DFG) is a directed graph representing the data dependencies in a program. The nodes have input and output data ports. The edges of a DFG model the connections between output ports and input ports; in other words, it point out the consumption of the input data and produce the output data. This graph allows the modeling of operations in a functional model, and doesn't take into consideration the conditional controls [57].

4.3.4 Program Dependence Graph

PDG is a directed graph, that was first proposed by Ferrante et al.[58] as a program representation taking into consideration both control and data relationships in a program. The nodes are predicates (variable declarations, assignments, control predicates) and edges are data and control dependence representation; both types are computed using respectively control-flow and dataflow analysis. PDG is the intra-procedural representation of a program, and considers the control and data flow dependencies within each procedure. On the other hand, SDG modeling considers the inter-procedural calls, that is, the control and data dependencies between procedures in a program.

SDG [21] is an inter-procedural dependence graph representation, and is an extension to the Program Dependence Graph (PDG). SDGs were first proposed by the authors of "Interprocedural Slicing Using Dependence Graph" [36], and have proved to be useful in performing deep analysis of programs [59]. They have been developed over the last two decades, and consist now a basis to perform the code analysis and proved to be useful in program optimization, based on different approaches, such as slicing[36] [44] [49] or Model Checking [39].

PDGs have the ability to represent the information flow in a program, and have different properties, such as being flow-sensitive, context-sensitive and object-sensitive [56].

PDGs are flow-sensitive, meaning that the order of statements is taken into account, and only feasible paths (possible executions) are indicated on the graph. The context-sensitivity is perceived from the fact that the method calling context is taken into consideration; different calls to the same method are considered separately and represented each in a separate node in the graph. If the PDG was context-insensitive, the different invocations for the same method would be merged in one single node on the graph. PDGs are also object-sensitive, that is, the object which is an instance of a class, is not considered as an atomic entity, and its attributes are taken into account.

All the sensitivities mentioned above improve the precision of the analysis that can be performed on the PDG/SDG [56].

In the figure below, is represented the Program Dependence Graph of the sample code in Figure 1. The construction of this graph is not trivial, and requires additional details that are not covered in this paper, such as the exceptions analysis [56], the multi-threading, the points-to analysis [52] that aims at determining the set of objects pointed by a reference object or a reference variable.

5. Static Code Analysis Tools

In order to have more concrete information, we conducted an analysis on source code analysis tools. The performed research doesn't aim at classifying or ranking the tools, but instead, to allow a deeper understanding of the analysis approach adopted by these tools, to underline the program modeling scheme they make use of, as well as the security properties they are able to detect. Static analysis tools are used mainly to find security vulnerabilities in the code, so that the developer makes the needed corrections on the detected security flaws before the software is released to customer.

We investigated a number of static analysis tools with the aim of identifying their analysis methodologies, and comparing their accuracy and performance. The precision of the analysis performed by the studied tools is estimated regarding the amount of false positives they report. The main goal behind this analysis is to underline the shortcomings of the approaches, and to shed the light on areas where enhancements can be proposed. Some of the security analysis tools generate an important number of false positives (false alarm), which reduces the efficiency of the considered tool [7]. On the other hand, a tool that reduces both false negatives (flaws that the tool doesn't report) and false positives (reported flaws that the program doesn't contain) is deemed to be more accurate. Automated tools usually use string regular expressions they match against source code statements in order to identify security vulnerabilities [51]. Other elements are to be taken into consideration when making an investigation about static analysis tools, namely the considered security properties, the required human intervention amount, as well as the output visual aspect and interpretation complexity.

This section presents a research on a number of static analysis tools. We would consider ones that allow the detection and evaluation of security properties, as well as tools that are

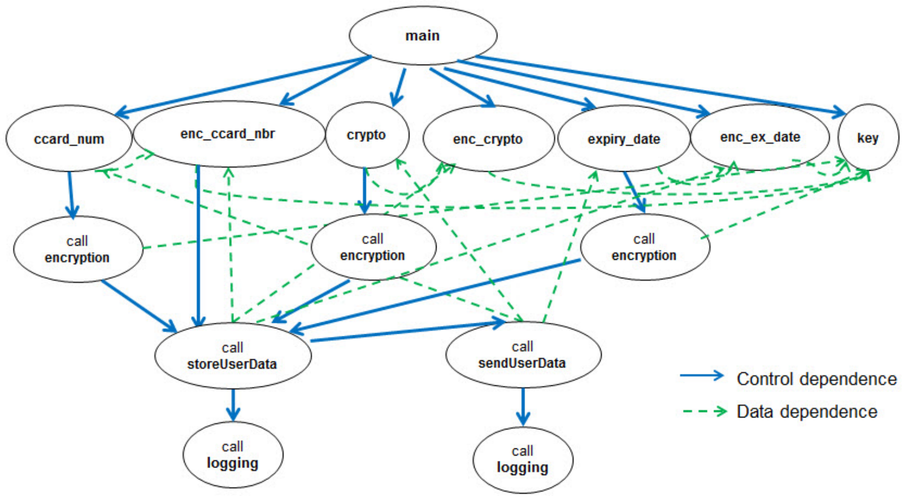


Figure 3. Program Dependence Graph

used to detect security vulnerabilities in the source code. We will justify the second alternative based on the fact that security vulnerabilities are violations of security properties. We carried out experiments on the sample code (Figure 1, Figure 2) using the presented tools. For some of the considered tools, we had to translate the sample code to the C programming language, as these tools only support C.

5.1 MOPS

MOPS (MOdel checking Program for Security Properties) [3] makes use of the model checking technique to check for violation of security rules, that are defined as "temporal safety properties". It is based on a formal modeling approach for both the program and the security properties, and proceeds by the analysis of the implemented models.

As for program representation, MOPS models the program in the form of Push Down Automaton (PDA), that contains all the feasible execution paths. Push Down Automata are used as tools to analyze procedural sequential programs, and more specifically those having recursive procedures [17]. As for automata, they are according to Schneider [16] used in the objective of specifying security policies that can be enforced by mechanisms. MOPS makes use of this approach to model security properties in the form of Finite State Automata (FSA), that dictate the order of security-relevant operations sequence. The modularity of security properties was also proposed by MOPS; this approach allows the decomposition of complex security properties into simpler and reusable basic security properties, that are easy to model and to extend (such as role based access). MOPS verifies that the security properties are properly respected in all the execution paths of the analyzed program, making use of the model checking on the PDA, and checks if risky states are reachable within the PDA.

5.2 SPlint

"Secure Programming LINT" (SPlint) [15] is an annotation-based data-flow static code analyzer for C for security vulnerabilities and programming flaw detection. It makes use of "annotations" (semantic comments) entered by the developer. The annotations serve as specification of the constraints (properties) about a library, a variable, a function or a type. In other words, annotations serve as properties specification. SPlint execution is an iterative process, that helps the developer/analyst to detect vulnerabilities locations, and to eliminate the warnings by adjusting annotations or modifying the code. SPlint parses the source code of the program subject to the analysis, and generates the Abstract Syntax Tree (AST) based on the formal semantics of the program's programming language. The annotations entered by the developer serve as specification of the high-level security properties about an asset. SPlint generates constraints from the annotations entered by the developer and adds them to the Abstract Syntax Tree (AST) of the program to analyze. As for annotations format, they are similar to comments in C: /* @notnull@ */ and they are syntactically associated to functions parameters, return values, variables, etc. Annotations allow expressing intra-procedural pre-conditions and post-conditions on assets. The developer/analyst can customize the security properties, and add security patterns to be detected in the AST according to his needs, which makes SPlint an extensible tool. If the property expressed by the annotation is violated, SPlint reports a warning for any return path that fails to satisfy the property.

5.3 GraphMatch

GraphMatch is a code analysis tool/prototype for security policy violation detection [9]. For the program modeling, GraphMatch makes use of a widely used tool called CodeSurfer[2] [31] that

[2]`http://hiper.cis.udel.edu/lp/lib/exe/fetch.php/courses/cisc879/codesurfer-demo.pdf`

generates the System Dependence Graph (SDG) from the source code provided as input. SDG [21] is an inter-procedural dependence graph representation, and is an extension to the Program Dependence Graph (PDG). SDGs were first proposed by the authors of *"Interprocedural Slicing Using Dependence Graph"* [36], and have proved to be useful in performing deep analysis of programs [32]. They have been developed over the last two decades, and consist now a basis to perform the code analysis, based on different approaches, such as slicing [36] [44] [49] or Model Checking [39]
As for the PDG, it is a directed graph whose nodes are predicates (variable declarations, assignments, control predicates) and edges are data and control dependence representation; both types are computed using respectively control-flow and data-flow analysis. PDG is the intra-procedural representation of a program, and considers the control and data flow dependencies within a procedure. On the other hand, SDG modeling considers the inter-procedural calls, that is, the control and data dependencies between procedures in a program. Given the fact that the generated SDG is in a proprietary file format, the authors have deemed necessary to transform the generated SDG into the GraphMatch's file format [10]. GraphMatch allows the users to customize the positive and negative security patterns that the program will be analyzed against, as well as to define the relationships between positive and negative security properties. John Wilander [8] has considered examples of security properties covering both positive and negative ones, that according to the author meet good and bad programming practices.[8]. GraphMatch traverses the generated SDG with the objective of finding security pattern matching. For a security property violation to be raised, GraphMatch proceeds in two steps: verification of the negative pattern matching first, followed by a verification of the embedding positive security pattern. If negative security patterns are found, the tool doesn't report a violation immediately, but proceeds to the verification of the embedding security property. If it corresponds to a positive security pattern, then GraphMatch doesn't raise a warning. However, if the embedding security pattern match is not found, GraphMatch raises a warning and reports a security property violation.

5.4 Fortify

Fortify [28] is a static analysis tool that processes the source code in a way similar to a code compiler. It has the ability to detect and fix vulnerabilities in the source code and to be run on multiple environments (Windows, Linux, Mac). Fortify takes as input the source code of a single file or an entire application composed of many files, and conducts a semantic analysis approach; it represents semantically the control flow and the data flow of the code. The tool is able to map the execution and the data flow and can for example recognize that input data are left untested or invalidated before being passed to a function or component. Fortify performs an inter-procedural analysis in the objective of making the analysis as accurate as possible. [3] The tool detects four types of issues: Semantic, Data Flow, Control Flow, Configuration and Structural. The Semantic Analyzer detects potentially dangerous uses of functions and APIs at the intra-procedural level. Basically a smart *GREP* [4]. The Data Flow analyzer detects potential vulnerabilities that involve tainted data (user-controlled input) put to potentially dangerous use. The data flow analyzer uses global, inter-procedural taint propagation analysis to detect the flow of data between a source (site of user input) and a sink (tainted data, or dangerous function call or operation) The Structural Analyzer detects flaws in the structure or the definition of the program. As for the Configuration Analyzer, it looks for dangerous flaws in the application deployment configuration files. The Control Flow Analyzer detects potentially dangerous sequences of operations. By analyzing control flow paths in a program, the control flow analyzer determines whether a set of operations are executed in a certain order.

The generated file is then processed by the Audit Workbench Tool that presents the results in a user-friendly format [27]. The Audit Workbench tool is customizable and enables the user to configure the custom rules (from the rules packs) for audit; the user selects the types of issues he wants to be warned about. The Workbench tool flags the detected vulnerabilities, provides the problem description and how it might be fixed.

5.5 Joana

Joana [47] is a framework that statically analyzes Java programs for the security properties confidentiality and integrity. The tool first generates from the program source a SDG (System Dependence Graph), which constitutes an over-approximation of the information flow in the program subject to the analysis. The SDG contains apart from the nodes representing the statements and the variable declarations in the program, contains also edges referring to control and data dependencies between nodes. The dependencies represent direct (explicit) dependencies as well as indirect and transitive dependencies (implicit), which allows to represent on the SDG only feasible paths.

The user annotates the generated SDG with security levels: high for secret variables, and low for public variables or method invocations. Joana then performs the analysis taking into account the annotations provided by the user and aims at detecting illegal information flows, that is, flows that violate the non-interference property between two security levels *a* and *b* stating that no information with security level *a* could influence information with security level *b* [55].

6. Evaluation and Discussion

In this section, we will reflect more upon the outcome of the static code analysis tools investigation, and will illustrate the results in a summary table, containing the main points of our

[3]`http://stackoverflow.com/questions/13051974/how-does-fortify-software-work`

[4]`http://blog.linuxacademy.com/linux/grep-tutorial-searching-file-contents/`

work. We focus on the program modeling approach that exploits the source code properties, and represents the program in a faithful model.

Different criteria are to be taken into account for evaluating a static code analysis tool, such as the security properties representation model, and which analysis approach is used to validate these security properties. The ability of the tool in modeling and detecting specified security properties is highly dependent on the abstraction level of the program modeling.
Another criterion is the soundness of the analysis tool. It can be evaluated taking into account the type of security vulnerabilities the tool is able to detect, as well as the amount of false positives and false negatives it produces. Reporting an important amount of false alarms can be misleading to the users, who will get discouraged of using that tool [40]. The soundness and accuracy of the tool's performed analysis is highly dependent on the precision of the code representation [41]; false alarms can emanate from an incorrect modeling of the system.

We are also interested in the mapping between detected vulnerabilities and the violated security properties, which is not covered by most of the studied static analysis tools.

The tools usability is one of our points of interest. A tool is deemed to be usable if its generated results are understandable to an average developer, meaning not having advanced knowledge in security. The usability can also emanate from the quality of its output and how it is presented to the user.

We focus also on the ability of the tool to allow users define their own rules against which the program will be evaluated. We consider the latter of paramount importance when dealing with static code analysis; most of the analyzed tools have their set of predefined rules. From this perspective, the tool will not detect a flaw if the corresponding pattern is not predefined.

We consider also the extensibility of the tool, that is, the integration possibilities it offers. A good tool has to be efficient and scalable enough to perform the analysis on complex or large programs, taking into account the dependencies between components without impacting the speed of its execution.

Table 1. Evaluation of Static Code Analysis Tools.

	MOPS	SPlint	GraphMatch	Fortify	JOANA
Program model	PDA	CFG	SDG	semantic DFG and CFG	SDG
Security properties model	FSA	Constraint-based	positive security pattern (PDG)	NA	Non-interference properties
Security vulner-abilities model	NA	NA	PDG	Signature-based patterns	NA
Static analy-sis method	Intra-procedural Control Flow, Inter-procedural Control Flow and Model-checking	Intra-procedural Control Flow, Intra-procedural Data Flow	Security pattern matching	Inter-procedural semantic, Data Flow, Control Flow configura-tion and structural analysis	Information flow anal-ysis
Customizable security properties	yes	yes	yes	yes [5]	yes
Extensibility	NA	NA	No public API	Commercial tool	Open source
Output	Text	Text	Text	HTML or XML file [60]	Text
Supported program-ming languages	C	C	C	C#, ABAP, C, C++, COBOL, Java, PHP, Python, Visual Basic, JavaScript, VB Script, etc. [28]	Java (source and bytecode)
Analysis re-sult	No violation reported	No violation reported	NA	Privacy vio-lation	Illegal informa-tion flows detected

SPlint is able to detect not only security-related vulnerabilities, but also coding errors that may affect the quality of the code [6]. However, SPlint doesn't handle multiple programming languages, and is only limited to C programs. Regarding its accuracy, SPlint produces an important number of false positives that lead to confusion when interpreting the results. In addition, it performs only intra-procedural data flow analysis; the control-flow and the inter-procedural data-flow analysis it performs are very limited. SPlint relies on annotations added by developers in their source code, in other words, a number of vulnerabilities will remain undetected if specific annotations are not added.

MOPS is control-flow sensitive, and doesn't consider data-flow dependence, that according to the others limits its scalability. Other shortcomings of MOPS are its incapability to analyze multi-threaded programs or dynamic methods invocation. The experiment on our sample code translated in C programming language, detected no violation for both SPlint and MOPS.

As for GraphMatch tool, it proceeds by model checking using the dependence graph, which combines both the program data and control dependencies, and proceeds by positive and negative security patterns. The tool doesn't scale to the analysis of distributed systems, but only considers a single source code file. Another issue with this tool is its non-scalability [9]. In addition, the graph matching performed by this tool has high complexity, which can impact the performance of the analysis. The tool is more focused on the liveness and safety properties, such as integer input validation and the double *free()* flaw where the *free()* method is called twice attempting to free the same memory allocated using the *malloc()* method. GraphMatch is mainly focused on the order and sequence of instructions, but doesn't cover high level security properties such as confidentiality. We couldn't carry out the experiment on our sample code, as GraphMatch was not available.

As for Fortify, this tool is scalable, and doesn't have restrictions on the size of the program to analyze. The performed analysis on our sample code produced as output a critical issue ”Privacy violation”, accompanied with explanation about the vulnerability (private user information enters the program, the data is written to an external location), its location, and recommendations on how to avoid it. However, it produces false positives. [6]

To have a basis for the information flow analysis it performs, the Joana tool requires annotations on the SDG (System Dependence Graph) it first generates. The user/developer needs as a first step to specify sources and sinks of the flows to analyze. In our sample code, we annotated the sensitive information (variables creditCardNumber, expiryDate and cryptogram) as *High* sources, the logging statement and the sendUserData method invocations as *Low* sinks.

As shown in the motivating example presented in section 2, static analysis tools allow only the detection of security vulnerabilities, but are unable to identify the security properties that might be affected by this vulnerability. Such identification can be realized by exploiting the vulnerabilities knowledge base (such as NVD). On the other hand, where no security vulnerability is detected, can one deem the analyzed program as secure? Going beyond the detection of security vulnerabilities, and tackling the problem of retrieving security properties using static code analysis is not trivial. Besides, the used encryption mechanism doesn’t provide assurance about the security property ”confidentiality”, that is highly dependent on the flow of the critical data through the program model, which has to be accurate, and has to exploit the source code properties, meaning the control and data dependencies. Besides, the model needs to have a certain level of abstraction and exploit as much as possible the program properties.

PDGs are considered a standard tool that allows the modeling of information flow through a program [42], and their strength consists in considering the order of sequences in the program. Hence, they provide an over-approximation of the analyzed program possible behaviors at run-time [43]. SDG modeling, which is an extension to the PDG, considers the inter-procedural calls, that is, the control and data dependencies between procedures in a program. From this perspective, we foresee the use of SDG as a program modeling approach, for its accuracy and its capability in modeling information-flow through a program.

7. Conclusion and Future Work

In this paper, we conducted an assessment on the selected static code analysis approaches and tools, with the objective of identifying the main shortcomings where major innovations can be performed. We outlined the main issues that consist in retrieving preserved security properties in the source code using code analysis, mapping the security vulnerabilities with the violated security properties, as well as the verification of the security properties when no vulnerability is detected. Static code analysis here consists in mapping security vulnerabilities to a program abstraction model. We presented how verification of simple security properties in the source code took place with different such representation models.

One of the main issues we are trying to tackle is how to translate the detected security properties enforced by mechanisms, into a concrete security policy, that can be afterwords validated against the security requirements expressed in the specification of the program. At this level, we have to be specific when discussing the security requirements that can be classified in levels, such as general security requirements, meaning the security requirements of the organization, and application-specific security requirements, meaning those that are defined in the context of that specific application.
Running a code analysis tool allows to detect the vulnerability in the exact location, but sometimes, programmers and code reviewers need to identify the source of that vulnerability [43], and hence to back-track the statements that led to that incorrect state[44].

Back-tracking vulnerabilities can be performed through the traversal of the program [44] regarding a variable of interest and the statement of the incorrect state. Weiser proposed the ”Program Slicing” technique [49] that was afterwords adopted in a number of researches [36] [32].

In relation to the paper’s motivating example we have also showed that the static code analysis tools discussed are not able to cover all the outlined issues. We plan to extend one of these tools in order to accommodate more complex security properties that may be derived from the code, even when no security vulnerability or threat is detected (e.g., confidentiality in storage). In order to overcome these challenges, we plan to address a few fundamental problems in coming work, namely:

- One such key question will be how to translate detected security properties enforced by mechanisms into a security policy that can be afterwards validated against the security requirements expressed in the specification of the program? Making the result of such an analysis easy to understand and to interpret for a developer will also be critical.

- A second critical question related to the determination of the scope of a vulnerability. How to detect the source of a vulnerability must be addressed in relationship with the type of static analysis selected? Running a code analysis tool allows the detection of a vulnerability in the exact code location, but sometimes, programmers and code reviewers need to identify the source of that vulnerability [43], and hence to back-track the analysis that led to such an incorrect state[44]. A related concern is how to determine which security property is under threat when a vulnerability is detected.

Acknowledgment

This work was partly supported by the EU-funded project OPTET [grant no. 317631].

References

[1] MAÑA, Antonio et PUJOL, Gimena. Towards formal specification of abstract security properties. In : *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*. IEEE, 2008. p. 80-87.

[2] CHESS, Brian et MCGRAW, Gary. Static analysis for security. *IEEE Security & Privacy*, 2004, vol. 2, no 6, p. 76-79.

[3] CHEN, Hao et WAGNER, David. MOPS: an infrastructure for examining security properties of software. In : *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002. p. 235-244.

[4] PISTOIA, Marco, CHANDRA, Satish, FINK, Stephen J., et al. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 2007, vol. 46, no 2, p. 265-288.

[5] CHESS, Brian et WEST, Jacob. *Secure Programming with Static Analysis, Software Security Series edition.*

[6] HELLSTRÖM, Patrik. *Tools for static code analysis: A survey*. 2009.

[7] MICHAEL, C.C., et LAVENHAR, Steven. *Source Code Analysis Tools - Overview.*, Cigital, Inc. 2005-2007 (n.d.).

[8] WILANDER, John. Modeling and visualizing security properties of code using dependence graphs. In : *Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden*. 2005. p. 65-74.

[9] WILANDER, John, et al. Pattern matching security properties of code using dependence graphs. In : IN Proceeding of the First International Workshop on Code Based Software Security Assessments. 2005.

[10] WILANDER, John. *Contributions to Specification, Implementation, and Execution of Secure Software*. 2013.

[11] HUTH, Michael. Model checking modal transition systems using Kripke structures. In : *Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2002. p. 302-316.

[12] SÖDERBERG, Emma, EKMAN, Torbjörn, HEDIN, Görel, et al. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming*, 2013, vol. 78, no 10, p. 1809-1827.

[13] SMELIK, Ruben, RENSINK, Arend, et KASTENBERG, Harmen. Specification and construction of control flow semantics. In : *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 2006. p. 65-72.

[14] Fabian van den Broek, *Static Code Analysis in Java*

[15] Splint Manual, `http://www.splint.org/manual/manual.pdf`

[16] SCHNEIDER, Fred B. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 2000, vol. 3, no 1, p. 30-50.

[17] BURKART, Olaf et STEFFEN, Bernhard. Composition, decomposition and model checking of pushdown processes. *Nordic Journal of Computing*, 1995, vol. 2, no 2, p. 89-125.

[18] *ARM Security Technology*, (n.d.).

[19] ZAKINTHINOS, Aris et LEE, E. Stewart. A general theory of security properties. In : *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*. IEEE, 1997. p. 94-102.

[20] ANISETTI, Marco, ARDAGNA, Claudio A., DAMIANI, Ernesto, et al. Web service assurance: The notion and the issues. *Future Internet*, 2012, vol. 4, no 1, p. 92-109.

[21] SINHA, Saurabh, HARROLD, Mary Jean, et ROTHERMEL, Gregg. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In : *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. IEEE, 1999. p. 432-441.

[22] CLARKE, Edmund M. et EMERSON, E. Allen. D*esign and synthesis of synchronization skeletons using branching time temporal logic*. Springer Berlin Heidelberg, 1982.

[23] Checkmarx Vulnerability Coverage, n.d.

[24] `http://www.checkmarx.com/technology/application-security-testing/`

[25] Checkmarx, `http://www.checkmarx.com/technology/static-code-analysis-sca/`

[26] Checkmarx CxEnterprise CxQuery API Guide V7.1.4

[27] Hyunji Kim, Fortify Source Code Analaysis Tools, n.d.

[28] HP Fortify Software Security Center v3.60, System Requirements

[29] GANESAN, Aarthi et BODDUPALLI, Aparna. *Static Analysis by Abstract Interpretations: For detection of security* vulnerabilities.

[30] WÖGERER, Wolfgang. A survey of static program analysis techniques. *Vienna University of Technology*, 2005.

[31] http://www.grammatech.com/research/technologies/codesurfer

[32] Code Surfer, *Dependence Graphs and Program Slicing*

[33] BACA, Dejan. *Automated static code analysis: a tool for early vulnerability detection*. 2009.

[34] ALLEN, Frances E. Control flow analysis. In : *ACM Sigplan Notices*. ACM, 1970. p. 1-19.

[35] SABELFELD, Andrei et MYERS, Andrew C. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 2003, vol. 21, no 1, p. 5-19.

[36] HORWITZ, Susan, REPS, Thomas, et BINKLEY, David. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1990, vol. 12, no 1, p. 26-60.

[37] BOEHM, Barry W. Software engineering economics. 1981.

[38] NOVAK, Jernej, KRAJNC, Andrej, et ZONTAR, R. Taxonomy of static code analysis tools. In : *MIPRO, 2010 Proceedings of the 33rd International Convention*. IEEE, 2010. p. 418-422.

[39] MATSUBARA, Masahiro, SAKURAI, Kohei, NARISAWA, Fumio, et al. Model Checking with Program Slicing Based on Variable Dependence Graphs. *arXiv preprint arXiv:1301.0041,* 2013.

[40] GOMES, Ivo, MORGADO, Pedro, GOMES, Tiago, et al. An overview on the static code analysis approach in software development. *Faculdade de Engenharia da Universidade do Porto, Portugal*, 2009.

[41] KONG, Deguang, ZHENG, Quan, CHEN, Chao, et al. Isa: a source code static vulnerability detection system based on data fusion. In : *Proceedings of the 2nd international conference on Scalable information systems*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007. p. 55.

[42] HAMMER, Christian, KRINKE, Jens, et SNELTING, Gregor. Information flow control for java based on path conditions in dependence graphs. In : *IEEE International Symposium on Secure Software Engineering*. 2006. p. 87-96.

[43] DENG, Fang et JONES, James A. Weighted system dependence graph. In : *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 2012. p. 380-389.

[44] WEISER, Mark. Programmers use slices when debugging. *Communications of the ACM*, 1982, vol. 25, no 7, p. 446-452.

[45] SMITH, Geoffrey. Principles of secure information flow analysis. In : *Malware Detection*. Springer US, 2007. p. 291-307.

[46] ROY, Arnab, DATTA, Anupam, DEREK, Ante, et al. Inductive trace properties for computational security. *Journal of Computer Security*, 2010, vol. 18, no 6, p. 1035-1073.

[47] GRAF, Jürgen, HECKER, Martin, et MOHR, Martin. Using JOANA for Information Flow Control in Java Programs-A Practical Guide. In : *Software Engineering (Workshops)*. 2013. p. 123-138.

[48] SHROFF, Paritosh, SMITH, Scott, et THOBER, Mark. Dynamic dependency monitoring to secure information flow. In : *Computer Security Foundations Symposium*, 2007. CSF'07. 20th IEEE. IEEE, 2007. p. 203-217.

[49] WEISER, Mark. Program slicing. In : *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981. p. 439-449.

[50] HICKS, Boniface, RUEDA, Sandra, JAEGER, Trent, et al. From Trusted to Secure: Building and Executing Applications That Enforce System Security. In : *USENIX Annual Technical Conference*. 2007. p. 34.

[51] CLARKE, Justin, ALVAREZ, Rodrigo Marcos et al. *SQL Injection Attacks and Defense, n.d.* `http://adrem.ua.ac.be/sites/adrem.ua.ac.be/files/sqlinjbook.pdf`

[52] ROUNTEV, Atanas, MILANOVA, Ana, et RYDER, Barbara G. Points-to analysis for Java using annotated constraints. In : *ACM SIGPLAN Notices*. ACM, 2001. p. 43-55.

[53] SCHMIDT, David et STEFFEN, Bernhard. Program analysis as model checking of abstract interpretations. In : *Static Analysis*. Springer Berlin Heidelberg, 1998. p. 351-380.

[54] ZHIOUA, Zeineb, SHORT, Stuart et ROUDIER, Yves. Static Code Analysis for Software Security: Problems and Approaches. In : *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*, 2014. p. 102-109.

[55] GOGUEN, Joseph A. et MESEGUER, José. Security policies and security models. In : *2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1982. p. 11-11.

[56] HAMMER, Christian et SNELTING, Gregor. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 2009, vol. 8, no 6, p. 399-422.

[57] QUAN, Gang. Data Flow Graphs Intro. `http://web.cecs.pdx.edu/~mperkows/temp/JULY/data-flow-graph.pdf`.

[58] Jeanne Ferrante, KARL J. OTTENSTEIN, and JOE D. WARREN, *The Program Dependence Graph and Its Use in Optimization*

[59] Dependence Graphs and Program Slicing ,n.d, `http://www.grammatech.com/images/pdf/dependence-graphs-and-program-slicing.pdf`.

[60] `http://143.132.8.23/cms/tues/docs/CSC450-Sp2012/4-FortifySCAtools.pdf`