Pós-Graduação em Ciência da Computação

Rodrigo Cardoso Amaral de Andrade

PRIVACY AND SECURITY CONSTRAINTS FOR CODE CONTRIBUTIONS

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

Recife
2018

Rodrigo Cardoso Amaral de Andrade

PRIVACY AND SECURITY CONSTRAINTS FOR CODE CONTRIBUTIONS

A P.h.D Thesis *presented to the Informatics Center of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Philosophy Doctor in Computer Science.*

Advisor: *Paulo Henrique Monteiro Borba*

Recife
2018

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB 4-1758

A553p Andrade, Rodrigo Cardoso Amaral de.
Privacy and security constraints for code contributions / Rodrigo Cardoso Amaral de Andrade . – 2018.
104 f.: fig., tab.

Orientador: Paulo Henrique Monteiro Borba.
Tese (Doutorado) – Universidade Federal de Pernambuco. Cin. Ciência da Computação. Recife, 2018.
Inclui referências e apêndice.

1. Engenharia de software. 2. Segurança da informação. I. Borba, Paulo Henrique Monteiro. (orientador). II. Título

005.1 CDD (22. ed.) UFPE-MEI 2018-71

Rodrigo Cardoso Amaral de Andrade

Privacy and Security Constraints for Code Contributions

Tese de Doutorado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Doutora em Ciência da Computação

Aprovado em: 08/03/2018.

_____

Orientador: Prof. Dr. Paulo Henrique Monteiro Borba

BANCA EXAMINADORA

_____

Prof. Dr. Vinicius Cardoso Garcia
Centro de Informática /UFPE

_____

Prof. Dr. Marcelo Bezerra d'Amorim
Centro de Informática /UFPE

_____

Prof. Dr. Fernando José Castor de Lima Filho
Centro de Informática / UFPE

_____

Prof. Dr. João Arthur Brunet Monteiro
Departamento de Sistemas e Computação / UFCG

_____

Prof. Dr. Marco Tulio de Oliveira Valente
Departamento de Computação / UFMG

Eu dedico esta tese à minha esposa, à minha mãe, ao meu pai e ao meu irmão.

Acknowledgements

First, I would like to thank Paulo for his dedication and patience during the last years. Not only for his Ph.D. supervision, but also for many valuable pieces of advice for my professional life. I take him as a model to be followed.

I also want to thank my family support across these tough years. Especially my wife who wants more than me that this Ph.D. is over.

Abstract

Developers must protect privacy and security of sensitive information handled by software systems. In collaborative software development, like GitHub's pull-based environment, developers submit code contributions using pull requests containing commits. These code contributions might carelessly or maliciously violate sensitive information privacy and security. If we do not appropriately detect this problem in the repository, the harmful code could remain unnoticed for a long time. This way, developers must be concerned about how to protect sensitive information from specific code contributions. One potential solution is to execute manual code review. Albeit commonly necessary, it is expensive, time-consuming, and error-prone. There are also automatic analysis tools to find violations throughout the source code. Nonetheless, they could be designed to work only for a specific technical domain, like Android, or they could demand a significant effort to specify policies and constraints. To mitigate these issues, we propose a new policy language named Salvum to allow the specification of constraints that help to protect sensitive information from specific contributions. Salvum allows a non-security specialist to define constraints to safeguard this information, and consequently avoid critical violations. We implement a tool to automatically enforce these constraints in two different ways. The first one allows us to check whether merged code contributions introduced violations of specified constraints. The second one allows us to automatically enforce constraints before integrating code contributions into the repository. Salvum supports the specification of constraints to determine the information that can and cannot flow to or be altered by specific contributions for Java projects. To determine whether there are such information flows, we use an existing set of Information Flow Control analyses called JOANA in our tool implementation. We evaluate our proposal regarding its ability to find violations of sensitive information for real software projects. Thus, we write policies and constraints for each selected project and execute our tool. We also investigate whether developers fix violations before merging code contributions on GitHub and whether unmerged code contributions are related to violations of sensitive information. We conclude that Salvum can indeed find such violations, mainly for poorly-supported projects. Moreover, there is no evidence: (i) that developers solve these issues before merging their contributions and (ii) that the unmerged code contributions are related to violations of sensitive information. We also investigate whether Salvum helps to reduce the effort of reviewing contributions to find violations. Thus, we compare the effort with and without our tool based on metrics like the number of lines of code to be reviewed. We conclude that Salvum can significantly reduce the effort to find violations of specified constraints. Furthermore, we assess the Information Flow Control analysis we use in our tool regarding precision, recall, and accuracy to find issues. Our results indicate that this analysis has high recall, but low precision and accuracy. Thus, the analyses could miss a few issues, but it could present high rates of false-positives. At last, the result of our work can mitigate privacy and security problems in the context of collaborative software development.

Keywords: Information Flow Control. Code Contribution. Policy language. Constraint. Sensitive information.

Resumo

Desenvolvedores devem proteger a privacidade e segurança de informações sensíveis manipuladas por sistemas de software. Em desenvolvimento colaborativo, como o ambiente pull-based do GitHub, desenvolvedores submetem contribuição de código usando pull requests contendo conjuntos de commits. Essas contribuições de código podem violar privacidade e segurança de informações sensíveis acidental ou maliciosamente. Se não detectarmos adequadamente esse problema, o código problemático pode permanecer indetectável por um longo tempo. Neste contexto, desenvolvedores devem se preocupar sobre como proteger informações sensíveis de determinadas contribuições de código. Uma solução é executar revisão manual de código. Embora comumente necessária, ela é cara, consome muito tempo e propícia a erros. Também há ferramentas de análise automática para achar violações espalhadas pelo código fonte. Entretanto, elas podem ser concebidas para funcionar somente para domínios técnicos específicos, como Android, ou podem requerer um esforço significante para especificar políticas e restrições. Para atenuar esses problemas, nós propomos uma nova linguagem de política nomeada Salvum para permitir a especificação de restrições que ajudem a proteger informações sensíveis de determinadas contribuições de código. Salvum permite que um leigo em segurança defina restrições para garantir a confidencialidade e integridade dessas informações. Nós implementamos uma ferramenta para aplicar essas restrições de duas formas diferentes. A primeira nos permite verificar se contribuições de código integradas introduziram violações para restrições especificadas. A segunda forma permite aplicar as restrições automaticamente antes de integrar contribuições no repositório. Salvum suporta a especificação de restrições simples para determinar a informação que pode e que não pode fluir ou ser alterada por determinadas contribuições de código em projetos Java. Para isso, nós usamos uma análise de fluxo de informação para implementar nossa ferramenta. Nós avaliamos nossa proposta com relação a habilidade de achar violações de informações sensíveis para projetos de software reais. Portanto, nós escrevemos políticas e restrições para cada projeto selecionado e executamos nossa ferramenta. Nós também investigamos se desenvolvedores removem violações antes de integrar contribuições de código no GitHub e se contribuições que não integradas estão relacionadas à violações de informações sensíveis. Nós concluímos que Salvum pode achar essas violações, principalmente para projetos que são pouco suportados. Além disso, concluímos que não há evidência: (i) que desenvolvedores resolvem esses problemas antes de integrar as contribuições e (ii) que contribuições que não foram integradas estão relacionadas a violações de informações sensíveis. Nós também investigamos se Salvum ajuda a reduzir o esforço de revisar contribuições de código para achar violações. Por conseguinte, comparamos o esforço com e sem nossa ferramenta baseados em métricas como o número de linhas de código que um revisor tem que revisar. Nós concluímos que Salvum pode reduzir o esforço de achar violações às restrições especificadas significativamente. Adicionalmente, avaliamos a análise de controle de fluxo de informação com relação a precisão, recall e acurácia para achar problemas.

Nossa conclusão é que esta análise tem alto recall, mas baixa precisão e acurácia. Assim, o resultado do nosso trabalho pode atenuar problemas de privacidade e segurança no contexto de desenvolvimento colaborativo de software.

Palavras-chave: Controle de fluxo de informação. Contribuição de código. Linguage de policy. Restrição. Informações sensíveis.

List of Figures

2.1 JOANA's user interface 20
2.2 Program Dependence Graph for Listing 2.3 23
2.3 SDG construction overview 24
2.4 Java program and its corresponding call graph 26
2.5 Example of git checkout 28

4.1 General Idea 44
4.2 The `Contribution` identifier and project versions 45
4.3 `noflow` and `noset` are permissive 50
4.4 Semantics of constructs 52
4.5 Overview of our tool 54
4.6 Five steps 56
4.7 High and Low labels 56

5.1 Metrics for **Q2** 61

A.1 Salvum grammar 104

List of Tables

2.1 Pointer analyis options 27

5.1 GQM to assess Salvum 59
5.2 Selected software projects 63
5.3 Results for violations 67
5.4 Manually analyzed pull request messages 70
5.5 Manually analyzed unmerged pull requests 71
5.6 Results for revisions 73
5.7 Selected software projects 74
5.8 Results for violations 76
5.9 SecuriBench Micro test results 78

Summary

1 Introduction **14**

2 Background **17**

2.1 Manual Code Review . . . 17

2.2 Static Information Flow Control analysis tools . . . 18

2.3 Confidentiality and Integrity . . . 20

2.4 Java Object-sensitive Analysis (JOANA) . . . 21

2.4.1 *System Dependence Graph (SDG)* . . . 22

2.4.1.1SDG creation . . . 24

2.4.2 *JOANA configuration* . . . 26

2.4.3 *Information flow control analysis* . . . 29

2.5 Collaborative software development . . . 31

2.5.1 *Git* . . . 31

3 Problem **34**

3.1 First scenario: Code contribution introducing a violation . . . 34

3.2 Second Scenario: Sensitive information leaking through third-party classes . . . 36

3.3 Third Scenario: Code contribution introducing a potential violation . . . 38

3.4 Fourth Scenario: Untrustworthy developer introducing violations . . . 39

3.5 Summary of existing approaches limitations . . . 40

4 Salvum **42**

4.1 General idea . . . 42

4.2 Language specification . . . 44

4.2.1 *Examples* . . . 44

4.2.2 *Constructs* . . . 48

4.2.3 *Representation of code contribution* . . . 48

4.2.4 *Discussion* . . . 50

4.3 Language implementation . . . 51

4.3.1 *First step - Preprocessing* . . . 53

4.3.2 *Second step - Generating SDG* . . . 54

4.3.3 *Third step - Labeling* . . . 55

4.3.4 *Fourth step - Analyzing.* . . . 56

4.3.5 *Fifth step - Results processing* . . . 57

5 Evaluation **58**

5.1 Goal, Questions, and Metrics . . . 58

5.2 Assessing highly active and well-supported software projects . . . 62

5.2.1 *Selected software projects* . . . 62
5.2.2 *Policies and Constraints* . . . 64
5.2.3 *Results* . . . 66
5.3 Assessing low active and poorly supported software projects . . . 73
5.3.1 *Selected software projects* . . . 74
5.3.2 *Policies and Constraints* . . . 75
5.3.3 *Results* . . . 75
5.4 JOANA evaluation . . . 78
5.5 Threats to validity . . . 81
5.5.1 *Construct validity* . . . 81
5.5.2 *Conclusion validity* . . . 82
5.5.3 *External validity* . . . 82
5.5.4 *Internal validity* . . . 83

6 Related Work **86**
6.1 Manual code review . . . 86
6.2 Information Flow Control analysis tools . . . 86
6.3 Language-based Information Flow . . . 88

7 Conclusions **91**
7.1 Review of main contributions . . . 92
7.2 Limitations . . . 93
7.3 Future work . . . 94

References **97**

Appendix A - Salvum Grammar **104**

1 Introduction

Many software systems handle sensitive information, such as confidential user data. Thus, developers must be concerned about how to protect such information. Due to the increasing popularity of collaborative software development in environments such as GitHub [1], it is necessary to protect sensitive information from specific potentially dangerous code contributions, which consists of a set of commits (e.g., untrustworthy developer pull requests). These code contributions might carelessly or maliciously introduce privacy and security violations [2] of sensitive information in existing projects [3]. It may become worse when inexperienced or untrustworthy developers are involved in the project development. Many issues related to violations introduced by code contributions can remain unnoticed for years, even in open source projects [4]. Therefore, we need mechanisms to find violations in this context. Classical approaches such as cryptography and certificates are insufficient to detect such violations because they do not analyze source code [5]. Thus, we should also consider proposals that identify violations at the source code level.

In collaborative software development, a common approach to do so is to manually review code contributions before integrating them into the repository. Albeit necessary in many cases, manual code review is time-consuming, error-prone, and costly [6, 7, 8]. Static analysis tools might reduce some manual code review drawbacks [9]. However, existing ones also introduce some disadvantages. Some tools are designed for specific technical domains [10, 11, 12], such as Android, which means that we cannot use them for other technical domains without many changes. Other existing tools might demand significant effort to execute an analysis for each code contribution [13, 14]. Additionally, some approaches allow developers to mitigate the significant effort issue, but they do not support code contributions [15, 16].

To reduce these problems, we propose a new policy language named Salvum[1] [17] to allow the specification of constraints that help to protect sensitive information from specific code contributions. For example, we can specify that a user password cannot flow to the execution of code contributions submitted by an untrustworthy developer. We would need to write the following constraint:

```
User {password} noflow Bob where Bob {c | c.author("Bob")}}
```

[1] Salvum means “secure” in Latin.

Additionally, Salvum allows developers to define these constraints for systems of different technical domains (e.g., Java Desktop or Java Web).

To enforce the specified constraints, we developed a tool, which automatically checks Java source code adherence to these constraints. We implement our tool based on an existing Information Flow Control analysis [18]. This analysis identifies whether there is an information flow from one source code location to another.

Roughly, our proposal works as follows: a developer writes Salvum constraints specifying the information and the whole code contribution. Then, she runs our tool to enforce these constraints either before or after merging the code contribution into the repository. Our tool provides a collection of violation warnings, which states the line of code where the potential violation occurs and the commit that introduced it. At last, with this information, the developer can review the code contribution to confirm the violations.

To show evidence that Salvum can indeed find relevant violations in the context of code contributions, we conducted an experiment considering nine highly active and well-supported software projects. Therefore, we studied their source code and wrote constraints for these projects and executed our tool. Our results indicate that our selected projects present a few violations (only three) of the specified constraints. However, even executing manual code review, developers could not detect such violations, so they remained unnoticed for months. We reported these violations on GitHub, and they were accepted and fixed. Additionally, to better understand why we detected a few violations for these projects, we also manually analyzed discussions and code commits attached to 243 pull requests on GitHub. Our first goal was to identify violations that were fixed before being merged into the repository. Our second goal was to establish a correlation between discarded pull requests and violations of sensitive information. Our results show that there is no evidence that developers fix violation before integrating code contributions for our sample. Moreover, we conclude that there is also no correlation between discarded pull requests and violations in our context. Since we found a few issues for well-supported software projects, we also perform an evaluation considering five low active and poorly-supported software projects. We also write a set of constraints for these projects and execute our tool. In contrast to the results regarding well-supported projects, we find a high number of violations of the specified constraints.

Additionally, we conduct an experiment to determine whether Salvum helps to reduce the effort to find violations. Therefore, we compare the lines of code and project versions that a reviewer should consider to find the violations that we identified for the mentioned projects. This way, we compare the effort of this task with and without Salvum (only manual code review). In this assessment, we conclude that Salvum can significantly reduce both lines of code and project version to manually analyze.

We also evaluate the Information Flow Control (IFC) analysis we use in our tool implementation. Our goal was to assess it regarding precision, recall, and accuracy to find issues for the Stanford SecuriBench Micro [19]. This benchmark provides groups of Java-written test

cases that we can use to identify the cases that this analysis misses information flows, which could impact our mentioned results. We conclude that this analysis presents high recall, and low precision and accuracy. The high recall indicates that we should not detect false-negatives, which avoids missed violations regarding the experiments above-mentioned. On the other hand, the low precision and accuracy might lead to a high number of false-positives. Nonetheless, we obtained a few false-positives in our experiment. We provide all the data necessary to reproduce this work in our online Appendix [20].

We organize the remainder of this work as follows:

- Chapter 2 reviews essential concepts used throughout this work;
- Chapter 3 discusses the problem we aim to address in this study. In particular, we illustrate scenarios where the problem that we want to tackle occurs and we explain limitations of existing approaches to solve it;
- Chapter 4 explains our language. It provides a general idea of Salvum as well as details of the language specification and implementation;
- Chapter 5 presents the evaluation we perform for Salvum to answer whether it can find relevant violations for well-supported and poorly-supported projects. Moreover, we discuss another assessment we performed to investigate whether Salvum helps to reduce the effort to review code contributions. Besides that, we discuss the evaluation of the Information Flow Control analysis that we use in our tool implementation. In the end of this chapter, we discuss threats to validity;
- Chapter 6 discusses related work;
- Chapter 7 presents the final considerations of this work.

2 Background

In this chapter, we review essential concepts explored in this work. First, we explain manual code review in Section 2.1. Then, we discuss existing automatic analysis approaches designed to mitigate manual code review drawbacks (Section 2.2). Furthermore, in Section 2.4, we describe the framework we use as a basis for our language implementation; we focus on its primary data structures and analysis. At last, we discuss collaborative software development and the Version Control System we use in Section 2.5.

2.1 Manual Code Review

Manual code reviewing is a form of inspection in which reviewers search for issues in source code. It is a way of ensuring developers are following proposed guidelines to build a system. Thus, we can use manual code review to check many properties, such as proper code indentation or lack of security flaws. We could use this technique to find issues that might compromise privacy and security of sensitive information. Frequently, it demands reviewers to understand the domain and purpose of the application [21].

In this context, available documentation might guide developers in the code reviewing task. For instance, the Open Web Application Security Project (OWASP) provides these guidelines [22] for web systems. Thus, by following them, reviewers can read the source code to find known vulnerabilities, such as those that allow SQL Injection, Cross-site Scripting, Race Conditions, etc. [21]. The code review task consists of reading the source code to determine if it is possible for third parties to maliciously explore the system information. Besides searching for known vulnerabilities, reviewers seek for violations of sensitive information, that is when this information might be exposed. For example, a reviewer can search for points within the source code where a secret token is printed in the user interface. Listing 2.1 of code below illustrates this example. Line 3 appends the token secret to a text area visible to users. Reviewers should detect this problem.

Code 2.1: Program fragment with violation of sensitive information

```
1  void actionPerformed(ActionEvent evt) {
2   String secret = getTokenSecret();
3   textArea.append(secret);
4   ...
5  }
```

Another typical scenario is to review code in collaborative development based on the pull request model [23]. Usually, an experienced developer performs the review task before integrating the code contribution into production repositories. Thus, it is possible to reject the code and demand changes fix to the detected problems.

More recent platforms like GitHub [1] allow developers to easily interact during the manual code review task. For example, a reviewer can comment any line of code changed, added, or removed by other developers in a set of commits. These platforms have increased the popularity of manually reviewing code contributions [24].

Albeit necessary in many cases, manual code review is time-consuming and expensive [6]. To do it correctly, human code reviewers must first know what privacy and security issues look like before they can rigorously examine the code [7, 8]. Additionally, they must reason about the sensitive information that should not be exposed as well as the points where it could leak. This understanding demands knowledge about the system domain, architecture, and source code. Therefore, it is usual to assign only experienced developers to the review task. Considering the experience we acquired on open-source and Java-written projects on GitHub [1] based on our experiments, only one or two developers perform the manual code review task independently of the total number of contributors. Besides that, even experienced reviewers can bypass some problems that might then escape the development environment and reach users, with potentially harmful consequences [4].

2.2 Static Information Flow Control analysis tools

To mitigate the drawbacks of manual code review researchers have proposed the use of static analysis tools. These tools automatically analyze source code to find different types of violations. In general, there are two analysis approaches.

The first analysis approach concerns the set of tools that allow developers to specify flows that configure violations, independently of the application domain. However, they are still dependent on programming languages. Jeeves [16] and PIDGIN [15] provide policy languages for specification of *Sources* and *Sinks*, which allow developers to determine program parts that hold sensitive information and source code location where it cannot flow. For example, we could write the code snippet below in the policy language provided by PIDGIN.

```
let secret = pgm.returnsOf("getRandom") in
 let outputs = pgm.formalsOf("output") in
  pgm.between(secret, outputs) is empty
```

This policy specifies that there must not be information flow between the return of method `getRandom()` and any arguments of method `output`. Thus, by writing it, developers can also reduce the effort of manual code reviewing.

Other tools such as JIF [14], CheckerFramework [25], and JOANA [13] provide an annotation mechanism. Thereby, developers can manually annotate *Sources* and *Sinks* before running the automatic analysis. These tools are also independent of application domains and dependent on programming languages. For instance, the snippet of JIF code below[1] specifies that `x` is ruled by a security policy.

```
int {Alice->Bob} x;
```

It states that the principal Alice controls the information in `x`, and she permits that the principal Bob sees this information. The policy `{Alice->Bob}` means Alice owns the information, and she allows Bob to affect it.

The CheckerFramework has a similar annotation mechanism. This way, developers tangle policy code with system's source code. On the other hand, JOANA provides a different annotation mechanism. Developers use its user interface to annotate program instructions as *Sources* and *Sinks* every time before running the analysis. Figure 2.1 illustrates this scenario. We annotate the `User.passwordHash` attribute as *Source* whereas the `println` instruction called in the `Log.loggingAction(user)` method as *Sink*. In this way, JOANA's analysis is capable of determining whether there is a flow between these two annotated program parts. Nonetheless, JOANA also provides an API to permit implementation of customs annotation mechanisms.

In this work, we use JOANA's analysis and API to implement our policy language. Therefore, in Section 2.4, we explain the JOANA tool, its analysis and data structures.

The second approach concerns the set of tools that analyze systems of specific technical domains, such as Java Web systems. In this context, some tools automatically find source code locations that hold user input. For example, a field that contains a password that the user typed. These tools also search for program parts where the information of these inputs should not flow to or must be sanitized before reaching them [11, 26], for instance, a call to a method that saves user password in the database without encryption. Thus, their goal is to determine whether there is a flow between user input and points where it might leak or cause harm. For the domain of Java Web applications, TAJ [11] can determine whether there is a flow between user entries in a form and database queries. Other tools designed for specific domains like Flowdroid [10] and Scandroid [12] hold a list of *Sources* and *Sinks* that represent sensitive information and leaking points, respectively. Their primary goal is also to determine whether

[1] https://www.cs.cornell.edu/jif/

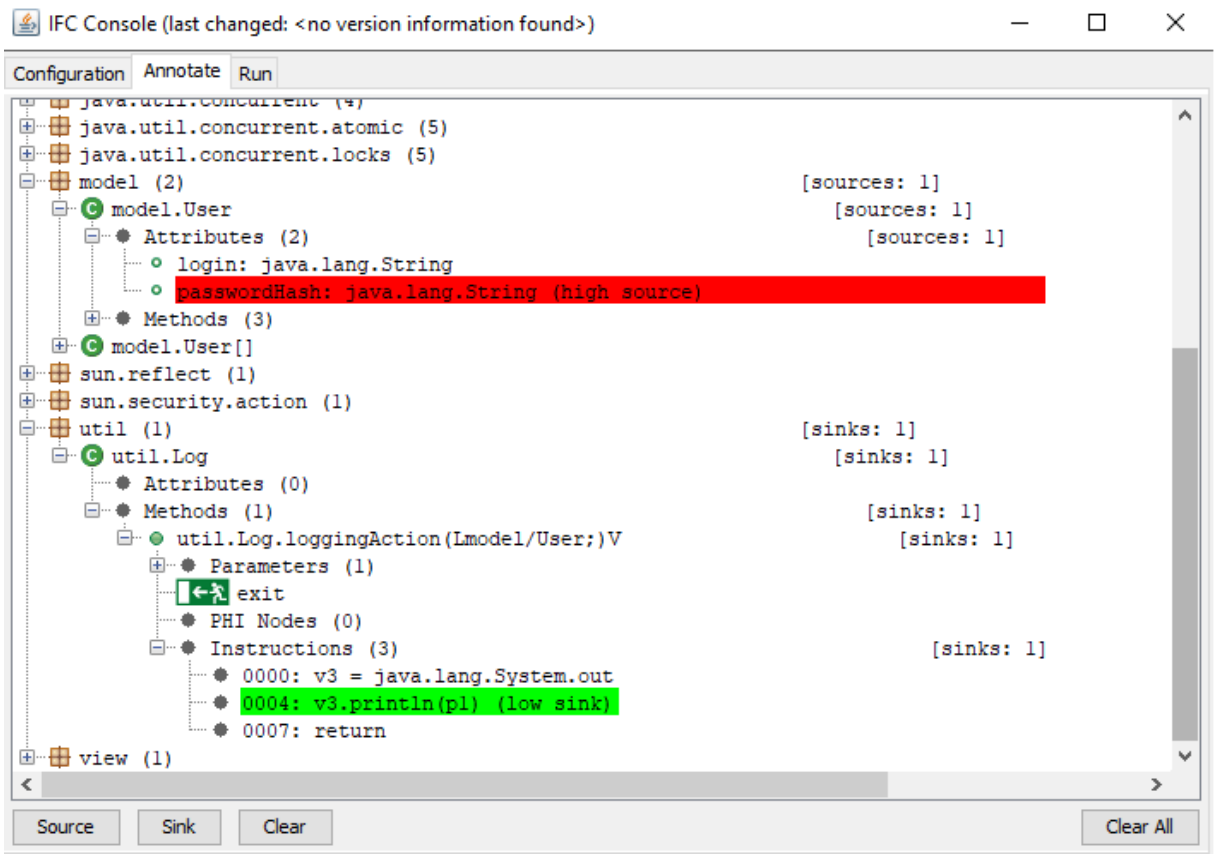

Figure 2.1: JOANA's user interface

there is information flow between a *Source* and a *Sink*. For example, Flowdroid [10] can establish the existence of information flows between a phone's IMEI and a `sendSMS` method for the Android application domain.

These tools allow reviewers to reduce the effort of manually reviewing source code to find issues regarding illegal information flow. On the other hand, such tools might not affect the productivity of manual code review for other tasks we mentioned in Section 2.1, like checking code indentation.

2.3 Confidentiality and Integrity

Before explaining JOANA in Section 2.4, we introduce two important concepts in Section 2.3: confidentiality and integrity properties. In this section, we explain two properties that are important to consider when working with sensitive data.

Confidentiality demands that sensitive information does not leak to public outputs [18]. It also refers to protecting sensitive information from being accessed by unauthorized parties. Only the people who are authorized to do so have access to the information. For example, in Listing 2.1, the sensitive information is the secret token whereas the public output is the user interface text area. In this work, we approach confidentiality regarding information flow. This

way, we can enforce confidentiality by controlling information flow from sensitive information to source code locations where it can leak [27], like the text area we mentioned above. Information flow policies can help us to express confidentiality properties of systems [28].

Integrity demands that public data does not influence sensitive computations [18]. It also refers to ensuring that information is not altered, and that the source of the information is genuine. For instance, Listing 2.2 illustrates a violation of sensitive information integrity.

Code 2.2: Program fragment showing integrity violation

```
1   void registerUser(User u) {
2     DatabaseConnection c = getConn();
3     String login = u.getLogin();
4     u.setPass("123");  // malicious code
5     String password = u.getPass();
6     c.saveUser(login, encrypt(password));
7   }
```

An unauthorized developer introduced the malicious code in line 4. It changes the sensitive information password so that he can easily guess the encrypted password of any user. This example assumes this malicious developer knows the encryption algorithm used in the `encrypt()` method, which is called in line 6. In this example, the public data is the unauthorized call to `setPass()` in line 4 whereas the sensitive computation is the `saveUser()` call in line 6.

We can also express integrity properties of systems using information flow policies. In this manner, we can enforce these properties by controlling information flow from potentially dangerous program parts (eg., a method that changes passwords) to sensitive information.

Lastly, confidentiality and integrity are dual to each other [29]. Thus, to guarantee both, we need to check for information flows from one source code location to another and vice-versa.

2.4 Java Object-sensitive Analysis (JOANA)

There are many Information Flow Control tools [25, 30, 31, 32, 6, 33, 34, 10, 11, 35, 14, 15, 16]. In this work, we implement our policy language using JOANA [13], which is an open-source framework written in Java that supports static analysis of Java systems to identify integrity and confidentiality constraint violations. It is built upon the WALA program analysis framework [30].

We choose JOANA because it is open-source and contains many customizable optimizations that improve analysis precision, which help to reduce the number of false-positives [33]. Additionally, JOANA provides an API that allows us to customize its annotation mechanism. It uses a special kind of graph named System Dependence Graph [36] in its analysis implementation. In Section 2.4.1, we explain this data structure.

2.4.1 System Dependence Graph (SDG)

JOANA backend is based on dependence graphs which capture dependencies between program statements in the form of a graph [33]. It uses two kinds of graph: Program Dependence Graphs [37] (PDG) and System Dependence Graph [36] (SDG). PDGs are intraprocedural whereas SDGs are interprocedural. In this way, an SDG adds edges between different PDGs accordingly to the call graph representation of the program.

In these dependence graphs, graph nodes represent program statements or expressions. Moreover, there are two kinds of edges: data dependence and control dependence [18]. Data dependence regards one statement (node) that assigns a variable which is used by another one (node) without being reassigned underway. Control dependence edge means that the execution of one statement depends directly on the value of a given expression, which is typically a condition clause in an `if` or `while`.

To determine whether an information flow exists from one node to another, JOANA requires that we manually annotate labels of different security levels to these nodes. Security levels are defined in security type systems, which associate levels—coded as types—to variables, fields, expressions, etc. and the typing rules propagate these levels through the expressions and statements of a program [18]. These labels might be *High* or *Low*, which could represent public or sensitive information. Thus, JOANA runs its analysis to identify potential flow between a *High* and *Low* label to check for confidentiality violations, and from *Low* to *High* to check for integrity violations. In Listing 2.3, we show a snippet of code and Figure 2.2 shows the corresponding PDG.

Code 2.3: Program fragment with information leak [33]

```
1 void main() {
2   String password = input();
3   print(password);
4   if (password.contains("123")) {
5     print("password contains 123");
6   }
7   print("not a leak");
8 }
```

Control dependence is shown as dashed edges and data dependence as solid lines. There is a path from node 2 to node 4, indicating that `password` may eventually influence the value of the conditional expression. There is also a data dependence between nodes 2 and 3 because the statement in 3 reads `password`. The control dependence from 1 to 2, 3, 4, and 7 indicates that these statements execute only if we call `main`. Additionally, there is a control dependence between 4 and 5 because the execution of the latter depends on the evaluation of the expression in the former.

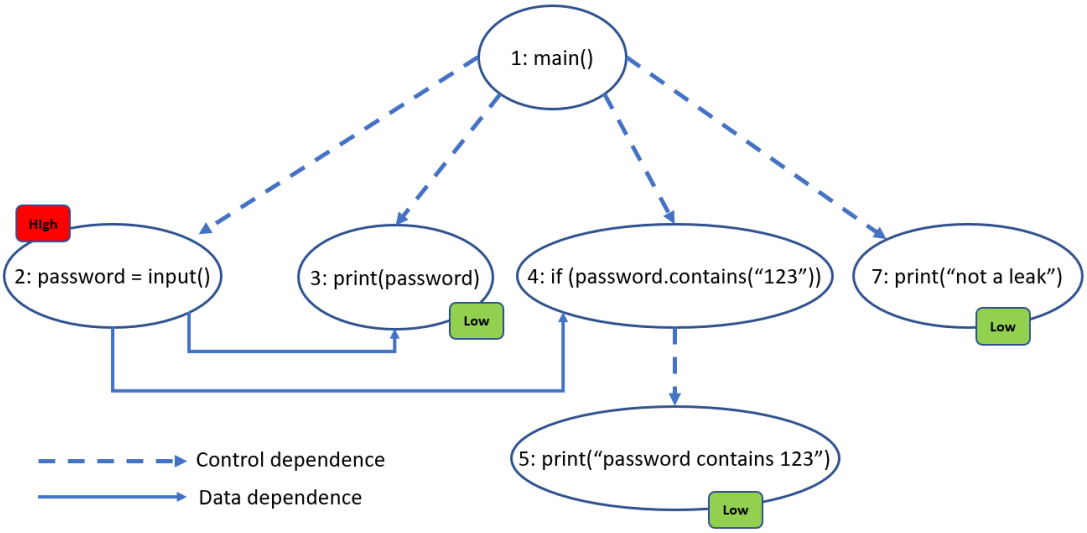

Figure 2.2: Program Dependence Graph for Listing 2.3

After building the graph of Figure 2.2, we should annotate the nodes that represent sensitive information and potential leaking statements. Therefore, we assign the *High* label to node 2 because it holds sensitive information. Furthermore, we assign the *Low* label to nodes 3, 5, and 7 because they can be leaking statements. The *High* label has a different security level from the *Low* label. Therefore, our goal is to determine whether there is a path in the graph from a node annotated with *High* to another node annotated with *Low*.

In this case, there is a path from node 2 to node 3, which indicates a violation of sensitive information confidentiality. Indeed, the statement in node 3 prints the `password`. Since there is no path from node 2 to node 7, sensitive information is not printed through `print("not a leak")`. However, there is a path from node 2 to 5, which consists of a Data dependence edge from 2 to 4 and a Control dependence from 4 to 5. Thus, the `password` value eventually influences the execution of 5.

To detect a violation of the integrity property, we check whether there is a path from a node annotated with *Low* to another one annotated with *High*. As we explained in Section 2.3, these two properties are dual.

We can annotate code instructions represented as PDG nodes with the user interface of Figure 2.1. In this way, we label node 2 with *High* label and nodes 3, 5, and 7 with *Low* label. These nodes correspond sensitive information and potential leaking statements, respectively. This annotating procedure might be time-consuming when we need to annotate many nodes scattered across an SDG. Moreover, we illustrate an PDG instead of SDG for simplicity. An SDG would have additional edges to connect method calls to their corresponding PDGs. For example, node 4 would have a Control dependence edge to the starting node of the PDG representing the method `contains()`. We explain the process of SDG creation next.

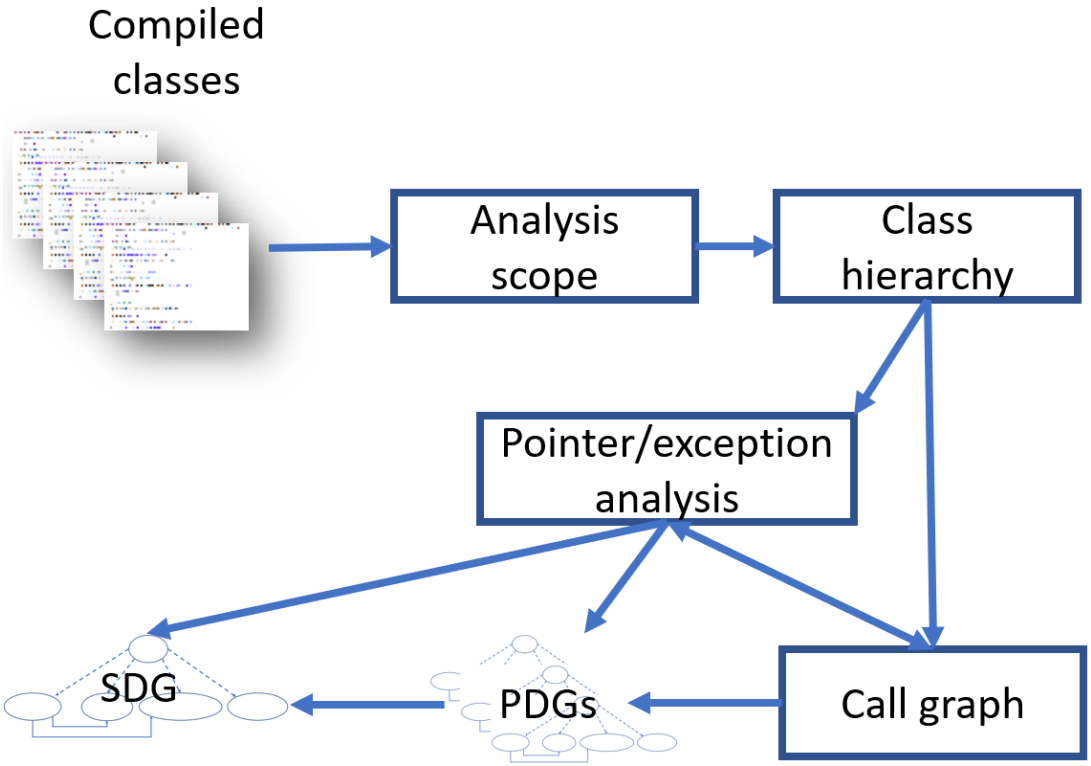

Figure 2.3: SDG construction overview

2.4.1.1 SDG creation

In Figure 2.3, we provide an overview of the process of SDG creation. First, we need a set of compiled Java classes. Then JOANA sets up the analysis scope, which defines what classes will be analyzed. We can classify these classes as Primordial, Application, and Extension class loader references. Primordial represents classes defined in the Java standard library, like ArrayList. Application represents the classes we define for the analyzed system. Extension regards third-party classes that we use in the system, such as those provided by libraries we include in the build path. Additionally, JOANA permits that we provide an exclusion list to the scope. This tool uses the analysis scope mechanism provided by the WALA framework.

Then, JOANA also uses WALA to create a class hierarchy based on the analysis scope. This data structure holds information about relationships between classes. For example, a subclass is related to its superclass. JOANA uses the class hierarchy as an auxiliary data structure to create other structures and to find the entry method we provide as input.

The call graph is, in summary, a structure that relates nodes and edges. The nodes represent methods whereas the edges represent calls to these methods. JOANA executes a pointer and exception analysis, which is used together with the class hierarchy to build a WALA call graph. The call graph uses an entry-point as a starter. From each node in the call graph built, JOANA creates an PDG, which models a given method. Finally, the call graph edges are used to connect the established PDGs, forming an SDG. For each call graph node, JOANA finds the target of the call to establish the connections.

JOANA allows that we configure the SDG creation so that the analysis is less or more precise. This configuration might have an impact on performance. The more accurate, the slower it is. Therefore, we opt for the most precise configuration that does not turn the analysis execution infeasible regarding time. We explain some of these configurations below.

Entry point.

To create a call graph of a program, JOANA requires that we provide an entry-point. For a Java Desktop system, the typical entry-point is the `main()` method. For native methods, the PDGs must be manually supplied or generated from so-called stubs [18]. JOANA supports stubs for two Java Runtime Environments (JREs): 1.4 and 1.5. In particular, these stubs include predefined models of native methods for the Java standard library. Additionally, JOANA supports multi-threaded systems. It is capable to compute interference SDG edges to model dependencies between threads [18]. In Figure 2.4, we show a Java program and the corresponding call graph.

```
public class Main {

    private String pwdHash;

    public String encrypt(String string) {
        return string + "---";
    }

    public void print() {
        String temp = this.encrypt(this.pwdHash);
        System.out.println(temp);
    }

    public static void main(String[] args) {
        Main m = new Main();
        m.print();
    }
}
```

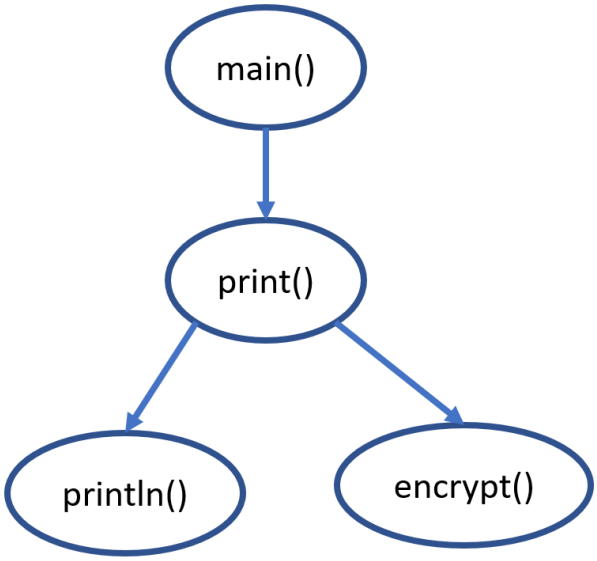

Figure 2.4: Java program and its corresponding call graph

The `main()` method calls `print()`, thus there is an edge between their corresponding nodes. In turn, `print()` calls `println()` and `encrypt()`, thus there are edges from the former to the latter. We can also provide other methods as entry-point. For example, usually Java Web systems do not have a `main` method. Thus, we can provide methods that encompass parts of code that we are interested in analyzing.

Exception analysis.

The SDGs we have shown so far do not include exceptions. Hammer and Snelting [18] discuss that exceptions can alter the control flow of a program and thus may lead to implicit flow[2], in case the exception is caught by some handler on the call-stack, or else in case the

[2]We explain this concept in Section 2.4.3

exception is propagated to the top of the stack yielding a program termination with stack trace.

To avoid missing these implicit flows, JOANA can deal with exceptions in two different ways. First, JOANA can conservatively add control flow edges from bytecode instruction to an appropriate exception handler. Second, it can percolate the exception to the callee which in turn receives such a conservative control flow edge [18]. We can configure different JOANA options regarding exceptions, which influences in the analysis precision. This tool provides four different configurations regarding exception analysis:

1. Ignore all exceptions;
2. Integrate all exceptions without optimization;
3. Integrate all exceptions with intraprocedural optimization;
4. Integrate all exceptions with interprocedural optimization.

The first option acts as if exceptions never occur. Thus, 1 is less precise, but faster. The second option assumes that each statement in a program can potentially throw an exception (e.g., `NullPointerException`). Hence, 2 tends to be more accurate than 1, but also slower. The third option acts like 2, but it applies an intraprocedural analysis that detects statements that definitely cannot throw an exception (e.g., the `this` pointer). For this reason, 3 tends to be more precise than 2, but it is also slower. At last, the fourth option extends the third option with interprocedural analysis. Therefore, 4 tends to be the most precise and the most time-consuming.

The choice for a more precise option affects the SDG creation because JOANA needs to include more nodes and edges for each exception occurrence. Thus, we create different SDGs if we use different options.

Pointer analysis.

The pointer analysis also affects the SDG creation. It is a static program analysis that determines information on the values of pointer variables or expressions. The goal of pointer analysis is to compute an approximation of the set of program objects that a pointer variable or expression can refer to [38]. JOANA inherits nine different pointer analyses provided by WALA. Table 2.1 explains each pointer analysis that we can choose to build an SDG. In Sec-tion 2.4.2, we discuss the configuration we choose to run JOANA in this work.

2.4.2 JOANA configuration

For this work, we set JOANA's configuration to build the SDGs with the object-sensitive pointer analysis and integrating all exception with interprocedural optimization since we notice that this configuration is the most precise and yet feasible (i.e., it finishes executing in a few

Table 2.1: Pointer analyis options

Pointer analysis	Description
Rapid Type Analysis	Protoype for academic purposes
Type based	It differentiates objects by their classes. Different instances of a given class are considered to be the same object
Instance based	It differentiates objects by their allocation points/positions
Object sensitive	It is capable of distinguishing different instances of the same class and methods invoked on different instances. Unlimited receiver context for application code and 1-level receiver context for library code
N1 Object sensitive	It limits the receiver context for both application and library code to 1-level
Unlimited object sensitive	Unlimited receiver context for the application and library code It tends to be very slow
N1 call stack	It differentiates objects by their allocation points/positions. Additionally it differentiates calls to methods of the same object with 1 level of call stack
N2 call stack	It differentiates objects by their allocation points/positions. Additionally it differentiates calls to methods of the same object with 2 levels of call stack
N3 call stack	It differentiates objects by their allocation points/positions. Additionally it differentiates calls to methods of the same object with 3 levels of call stack

days in the worst of our cases). This configuration tends to allow us to detect more violations. However, more precise settings take much more time and hardware resources.

As we explained in Section 2.4.1.1, the object-sensitive analysis is capable of distinguishing two objects from the same class. For example, Listing 2.4 shows a snippet of code in which `"secret"` does not leak to the `print()` method. Indeed, there is no path in the SDG from the node of line 10 to 13 because we create a new instance of `X` in line 11. Therefore, the object-sensitive analysis distinguishes the `value` variables in the two objects of `X`.

Code 2.4: Object-sensitivity example

```
1  class X {
2   String value;
3   void setValue(String v) { this.value = v;}
4   String getValue() {return this.value;}
5  }
6
7  class UsingX {
8   void main() {
9    X p = new X();
10   p.setValue("secret");
11   p = new X();
12   p.setValue("public");
13   print(p.getValue());
14  }
15 }
```

Despite the fact that we believe the scenario of Listing 2.4 is unusual, we still need to detect it. Therefore, we opt for object-sensitivity.

For the sake of precision, we also include the exception analysis that integrate all exceptions with interprocedural optimization. In contrast to the object-sensitive example, Listing 2.5 shows a real information leak we found in our evaluation (Chapter 5).

Code 2.5: Exception example

```
1  IPassword getPassword(String pwd) {
2   int index = this.pwd.indexOf( ':' );
3   if (index == -1) {
4    throw new IllegalArgumentException(pwd);
5   }
6   ...
7  }
```

Depending on the value of `index`, the program throws a new exception with the password as an argument. In case we label `pwd` as *High* and the exception throwing statement in line 4 as *Low*, we would be able to detect a violation even without exception analysis. However, with this option activated, we are also able to identify that there is a Control Dependence between `index` and the exception, which unveils precisely the statement that can throw `IllegalArgumentException`. Otherwise, with exception analysis deactivated, all the lines of the `getPassword()` method are potential throwers. Thus, to be able to identify when these problems occur in the context of exceptions, we also opt to include exception

analysis.

In this context, JOANA implements an SDG based Information Flow Control analysis that checks whether there are paths from nodes annotated with labels of different security level (eg., *High* and *Low*). Therefore, we explain the analyses in Section 2.4.3.

2.4.3 Information flow control analysis

Information Flow Control (IFC) is concerned with the flow of information inside a system. IFC analysis checks whether there is information flow from one point to another within a system or if this information is altered at some point. In our context, we use JOANA analyses to check whether sensitive information flows to or is altered by potentially dangerous operations (e.g., labeled with *Low*). To avoid these problems, we consider IFC as a technique for discovering paths of sensitive information flows. It has two main tasks [18]:

- Guarantee *confidentiality* of information;
- Guarantee *integrity* of information.

In particular, IFC analysis aims to establish noninterference [39], that is, by analyzing compiled source code, it tries to prove that there is no violation of confidentiality and integrity [40, 18].

In Listing 2.6, we illustrate a simple example of a path that IFC analysis detects to check sensitive information confidentiality. In this way, an SDG representation of the program is built, which makes it possible to check for paths from a given node to another. Thus, the information initially stored in `password` flows (has a path) to `print()` in line 6 because the `user` object contains this sensitive information. In this example, IFC analysis is useful for finding a violation of `password` information confidentiality. In this case, we need to provide to IFC tools the statements that contains sensitive information and the statements that might leak it. Thus, the node that corresponds to `password` is labeled as *High* and `print()` node is labeled as *Low*. Therefore, `password` is our *Source* and `print()` is our *Sink* in this example.

Code 2.6: Sensitive information confidentiality

```
1  String login = "public";
2  String password = "secret";    // High
3  User user = new User(login, password);
4  signIn(user);
5  Logger.info("User signed in: " + login);
6  print(User)   // Low
```

Additionally, Listing 2.7 illustrates an example of a path that IFC analysis identifies to determine whether there is a violation of sensitive information integrity. The information

initially stored in `password` is altered by `changeUserPassword()` because this method reassigns the `password` variable. Given the SDG representation of the program, IFC analysis is capable of identifying that there is a path between these two corresponding nodes. In this case, there is a Data dependence between the node that corresponds to `changeUserPassword()` and the node that corresponds to `password`. Different from the confidentiality example, `changeUserPassword()` is labeled as *High* and `password` as *Low*. This annotation switching happens because confidentiality and integrity are dual [29]. Thus, sensitive information labeled as *Low* and dangerous operations labeled as *High* allow us to check information integrity instead of confidentiality.

Code 2.7: Sensitive information integrity

```
1  String login = "public";
2  String password = "secret";    // Low
3  User user = new User(login, password);
4  changeUserPassword(password); // High
```

Static analysis tools can help to reduce the effort of Manual Code Review because they can check systems more frequently, and they encapsulate privacy and security knowledge in a way that does not require the tool operator to have the same level of expertise as a human code reviewer [7]. Another frequent use of these analyses is to guarantee the **Principle of Least Privilege** [41], which states that every program and every user should operate using the least amount of privilege necessary to correctly complete the job.

Additionally, IFC analysis is capable of detecting different kinds of information flows. An explicit flow arises if a secret value is copied to public output whereas an implicit flow arises if a secret value can influence the control flow. Listing 2.8 implements an example that shows both kinds of flow.

Code 2.8: Explicit and implicit flows

```
1  void main () {
2    x = inputPIN(); // secret
3    print(x); // explicit flow
4    if (x < 1234) {
5      print(0); // implicit flow
6  }
```

The value of variable `x` is secret. In this example we have an explicit flows from `x` to a public output (`print()`) in line 3. This program prints `0` if the secret value is less than `1234`. Therefore, the value of `x` influences the program control flow (conditional clause in line 4), which represents an implicit flow. If an attacker has access to the source code, he would know that secret is less than `1234`, which masks it more straightforward to guess the correct secret value.

2.5 Collaborative software development

In collaborative software development, many developers implement code and submit to a common source code repository. We use a term for the code and submissions throughout this work: code contribution. Here a code contribution is a set[3] of commits submitted to a repository managed by a Version Control System (VCS). For example, a code contribution might be:

- A pull request in GitHub [1];
- Commits with messages containing a certain word (e.g., to identify commits related to issues);
- Commits submitted by a specific author;
- Commits related to the result of a task execution;

Collaborative software developers might be talented and experienced, but they can also be untrustworthy or malicious. In particular, an open-source software project can receive code submissions from any developer. To help protecting these systems from harm, we use this code contribution concept in our language specification, as we explain in Chapter 4. Our language implementation currently supports the Git [42] Version Control System.

2.5.1 Git

Git [42] is a distributed Version Control System that allows developers to contribute to repositories in an organized way. By the end of 2017, the most used hosting service using Git, named GitHub [1], has around 20 million users, 57 million repositories, and 100 million pull request [43]. This is the main reason we choose Git for our policy language implementation. Nonetheless, the concepts of our language also apply to other Version Control Systems.

Pull requests let a developer warn others about commits pushed to a repository on GitHub. In this way, experienced developers can manually review and discuss changes with collaborators before merging the changes. Thus, pull-based collaborative software development encourages code review. However, as we mentioned in Section 2.1, developers might bypass some issues.

To be capable of analyzing code contributions, we use four Git commands in our language implementation (Chapter 4): `diff`, `checkout`, `blame`, and `log`.

First, the git `diff` command shows changes between commits. It generates a file indicating the lines added, changed, or removed. In Listing 2.9, we show an example of `diff`'s output. The latest commit adds the line shaded in green (line 2) whereas it removes the line shaded in red (line 3). Notice that before this latest commit, only line 3 existed. We use `diff`

[3]The order does not matter.

outputs in our language implementation to determine the lines of code that correspond to a certain code contribution.

Code 2.9: `diff` output

```
1 void showDiff() {
2     + print("Hello added");
3     – print("Hello deleted");
4 }
```

Second, the git `checkout` command updates file in the working tree to match the specified source code version. For instance, we can `checkout` a source code to a specific commit identified by a commit hash: `git checkout gsd68dh`. The result of this command execution is the repository showing exactly how all its files were when a particular developer made a commit associated with the commit hash `gsd68dh`. Git automatically generates these commit hashes, and they are unique for each commit, independently of repositories. In this way, we use git `checkout` to obtain the system version we have to analyze. Figure 2.5 illustrates an example of Git checkout.

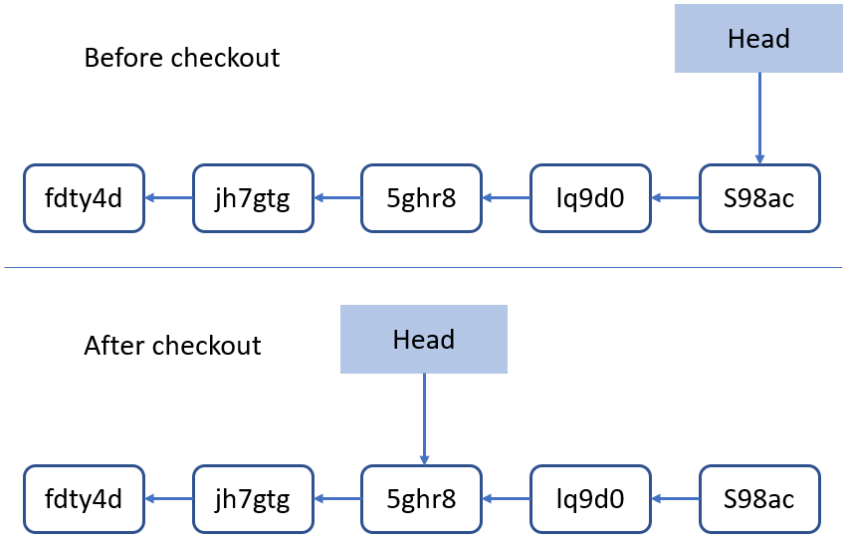

Figure 2.5: Example of git checkout

Before checkout, the current repository state is on the `s98ac` commit. After checkout, all the files downgrade to the state they were on commit `5ghr8`. In this way, we can analyze the source code corresponding to the commit `5ghr8`.

Third, the `blame` command shows the code contribution and author that last modified each line of a file. Thus, it is useful in our context to identify the developer and his code contribution that introduced a potential violation in a system source code. For example, we can run the `blame` command to retrieve information about the last commit that modified line 4. For example, on command line, we can use this: `git blame -L 4,4 - - AuthenticationProvider.java`.

This command identifies the last commit that makes a change to line 4 in the `AuthenticationProvider.java` file.

Code 2.10: Git blame example

```
1   class AuthenticationProvider {
2    void setCookie(UserModel user) {
3     if (StringUtils.isEmpty(user.cookie))
4      user.cookie = user.createCookie();
5    } ...
6   }
```

This command would inform that the commit identified by hash `b453703` was committed by the author Rodrigo Andrade on the 15th of August. Furthermore, we can call this Git feature programmatically.

Fourth, we also use the `log` command to find the set of commit hashes corresponding to a given author or to filter commits containing a keyword in their messages. For example, the command below retrieves the commit history for a given software project where the commit messages include the keyword "leak".

```
git log -grep="leak"
```

For each commit found, this command output consists of its hash, committer, date, and message.

In this chapter, we explain the central concepts that the reader should know to understand this work. Thus, we define Manual Code Review and briefly discuss its limitations. Moreover, we present some real automatic analysis to find issues in the source code. To better understand the kind of analysis we use in this work, we explain the information integrity and confidentiality properties. So, we describe the Information Flow Control analysis and its main structures. We end this chapter explaining collaborative software development and the Version Control System we use in this study. In the next chapter, we illustrate four scenarios of problems we aim to tackle.

3 Problem

The Common Weakness Enumeration (CWE) has a catalog of more than 1000 kinds of software weaknesses [44]. These are different ways that software developers can introduce violations that lead to non-secure projects. Each violation could be subtle, and many are seriously tricky. Software developers are not taught about these issues in school, and most do not receive any training on the job about these problems [21]. Indeed, other researchers have found many problems in Java projects [6, 11, 26, 45].

In this chapter, we explain four scenarios that illustrate privacy and security problems that could be time-consuming and error-prone to detect with existing approaches, such as manual code review and plain IFC tools. We explain the first scenario in Section 3.1, the second one in Section 3.2, the third one in Section 3.3, and the last one in Section 3.4. These scenarios illustrate similar problems regarding violations of sensitive information. However, they differ in the way we could detect such violations. For example, the first scenario demands that we analyze different code contributions whereas the second scenario demands that we analyze potentially dangerous classes in a single software version. At last, we summarize some limitations of manual code review and existing IFC analysis for tackling the problems presented in our scenarios (Section 3.5).

3.1 First scenario: Code contribution introducing a violation

In this scenario, we discuss an example of a real problem in the Gitblit [46] project. In this git repository manager, there is a web user interface that administrators can use to create, edit, rename, or delete repositories, users, and teams. Furthermore, Gitblit implements a cookie mechanism to identify these web interface users. This project is open-source and supported by several developers, who submit their code contributions to a GitHub[1] repository. We provide further details about Gitblit in Section 5.2.1.

One of these contributions resulted from an enhancement task assigned to one developer [47]. His task was to improve the Gitblit code that deals with Java Servlets. Among other code changes in the resulting contribution [47], the developer added the green line (line 5) in

[1] https://github.com/gitblit/gitblit

Listing 3.1 to set a user cookie. In particular, `setCookie` receives the `user` object, which is an instance of `UserModel`.

Code 3.1: Setting an user cookie

```
1  class RootPage {
2   void loginUser(UserModel user) {
3   if (user != null) {
4    ...
5     app().auth().setCookie(request, response, user);
6    }
7   ...
8   }
```

However, we believe that the developer did not notice that `UserModel` instances hold sensitive user information, such as `password`, `email`, and `location`. Since the experienced developer responsible for manually reviewing code contributions in Gitblit did not identify any problems, this `setCookie` call was merged into the production repository. Hence, when we execute the `loginUser` method, sensitive user information unnecessarily flows in an implicit way to `setCookie`, which we call in the new code contribution (line 5). This situation is a violation of the Principle of Least Privilege [41], which states that every program part should operate using the least amount of privilege necessary to correctly complete the job.

Thus, this new code contribution should not have access to sensitive user information contained in the `user` object, such as the one stored in `UserModel.password`. Moreover, the code contribution can expose sensitive user information to any attacker possessing access to user browser data. Indeed, Gitblit developers use a weak encryption algorithm (SHA-1) to deal with this confidential information in cookie implementation [48]. Additionally, an attacker might use tools like HashKiller [49] to decrypt the resulting hash key and expose user sensitive information.

This problematic code contribution was merged on the 3rd of July 2014. We have reported this potential leak, and Gitblit developers requested a pull request. We submitted a fix and it was integrated on 13 December 2016[2]. Thus, it might take a long time to detect and solve these problems [4].

In case we consider existing approaches to find the issue presented in this scenario, the person responsible for reviewing the code changes that generated the illustrated problem would have to read and understand 142 additions and 142 deletions in the Gitblit source code [20] only for this contribution. This person would have to be an expert in the system because she must know the information that should be protected and also the program parts that could leak it. In the case of Gitblit, which contains many pieces of sensitive information, this could be a hard task. Notice that even though Gitblit has a manual code review process, reviewers could not

[2] https://github.com/gitblit/gitblit/pull/1116

identify the issue presented in this scenario.

On the other hand, when using JOANA [13] on the Gitblit system for the code contribution of Listing 3.1, we have to analyze 408 bytecode instructions to annotate *High* and *Low* to run its IFC analysis. They originate from a method with 300 lines of code (each line can contain more than one instruction). Furthermore, we might have to annotate instruction for other classes and methods. Labeling these instructions was time-consuming even using a graphical user interface provided by JOANA. Nevertheless, we could mitigate the effort of this labeling task with a more user-friendly interface, but it would still demand significant work because there are 142 lines of methods and fields added in this code contribution. Jif [14] and Checker Framework [25] might introduce more drawbacks since we would need to change the source code to associate program elements with their types or annotations for each code contribution. For example, we would need to manually annotate additional types for the 142 lines added in this contribution, like line 5 in Listing 3.1. However, the user password did not change, so we would need to do further annotations for it.

3.2 Second Scenario: Sensitive information leaking through third-party classes

Our second scenario illustrates an issue in the Blojsom [50] project, which is an open-source and Java-written blog application. This system calls many external library methods. One of these libraries is the Simple Logging Facade for Java (SLF4J) [51], which is useful to log operations throughout the source code. These logging operations are known to be a potentially dangerous point of information leak [52] as specified in the Common Weakness Enumeration [44] and in The Open Web Application Security Project [22]. Thus, in many situations, we should not log sensitive information. However, untrustworthy or careless developers might ignore this problem, which might compromise information confidentiality and integrity. Thus, we should be able to find these leaks and discover the code contribution that introduced it.

For example, Listing 3.2 illustrates a method of the `AtomAPIServlet` class that checks whether a user is authorized to request a resource. Line 4 obtains the user password and line 8 logs it in case an error occurs. Thus, the `password` value is passed as an argument to `logger.info()`, which leads to sensitive information flowing to the code of an external library (SLF4J).

Code 3.2: Logging passwords

```
1   boolean isAuthorized(HttpServletRequest req) {
2     boolean result = false;
3     ...
4     password = realmAuth.substring(pos + 1);
5     result = _blog.checkAuthorization(username, password);
6     if (!result) {
7       logger.info("Unable to auth user [" + username + "]
8       with password [" + password + "]");
9     } ...
10  }
```

The code in Listing 3.2 might be dangerous because it prints the `password` in the log files, which could expose this information to an attacker in case the user provide the wrong `username`, but the correct `password`. In this context, sensitive information should not flow or be changed by the code of external libraries such as `Logger` methods. We could not report this issue because, differently from Gitblit, the Blojsom project is no longer supported. Thus, the problem we illustrate in Listing 3.2 is still present in Blojsom source code.

To detect this problem, a manual code reviewer would have to localize and understand 58 references to `Logger` and 340 calls to its methods in the most recent version of Blojsom. In contrast to the other scenarios discussed so far, the Blojsom project does not have a code contribution review process. Thus, developers might have integrated many violations. For this reason, we should also be able to detect privacy and security violations to such cases.

Moreover, she would have to identify flows of sensitive information to these methods. For example, she would have to reason whether the password value reaches the `logger.info()` method. In case the `password` is reassigned before reaching `logger.info()`, there is no violation. Identifying such issues could be a hard task because it includes explicit and implicit flows.

At last, JOANA would require that sensitive information and `Logger` methods are manually labeled. This task is not as time-consuming as using these tools for the first scenario because we only consider one Blojsom version instead of many code contributions. However one might forget to annotate instructions across 340 different `Logger` method calls. Additionally, if we consider Checker Framework or JIF, we would have to scatter and tangle policy code with core code, which could make it harder to evolve and maintain the system [53].

3.3 Third Scenario: Code contribution introducing a potential violation

This scenario differs from the first scenario because here the code contribution introduces a potential violation. We consider a real example in the ScribeJava [54] project. It is a simple authentication library for Java programs. This project is also open-source and supported by several developers, who submit their code contributions to its repository on GitHub. We explain ScribeJava with more details in Section 5.2.1.

The OAuth [55] library has representations of access tokens, which hold sensitive information such as the token secret. Many classes implement these representations. Each of them override `toString` methods. In 19th of February 2016, a developer submitted a code contribution with the goal of updating token representations.[3] Among 559 changed lines throughout 33 files, this developer implemented `equals()` and `toString()` methods. Listing 3.3 illustrates one `toString()` implementation introduced by this code contribution.

Code 3.3: Implementing toString method

```
1 class OAuth1AccessToken {
2     String toString() {
3         return "OAuth1AccessToken{"
4           + "oauthtoken=" + getToken()
5           + ", oauthtokensecret=" + getTokenSecret() + "}";
6     }
7 }
```

This developer carelessly introduced the sensitive information token secret in the result yielded by the `toString` method. In case other developers call `OAuth1AccessToken.toString()`, the token secret would be printed, which exposes this sensitive information. ScribeJava developers did not notice this issue during manual code review. Thus, this dangerous code was merged into the central repository.

In this context, the code contribution illustrated in Listing 3.3 should not have access to the token secret in the `toString` method because, in case some developer calls it, sensitive information would be printed. We have opened an issue in the ScribeJava project and one of its maintainers accepted it as a problem.[4] Since it was an easy fix, this developer submitted a new commit excluding the token secret from the `toString` method.[5] In this case, it took 21 months from the submission of the problematic code contribution to the commit fixing it.

To find the problem introduced by this code contribution, a reviewer might need to understand sensitive information flows for 33 different files with 385 additions and 174 deletions.

[3] https://github.com/scribejava/scribejava/commit/d734a4df
[4] https://github.com/scribejava/scribejava/issues/796
[5] https://github.com/scribejava/scribejava/commit/73c29f5

Moreover, she would have to identify the sensitive information that might flow to the code contribution. This review task might be unfeasible if we need to analyze a large number of code contributions manually.

Analogously to the scenario of Section 3.1, the task of manually annotating code contributions might be time-consuming and error-prone if we use JOANA. Notice that there are more changed lines in this scenario which leads to the need for more instruction labeling. At last, if we use other tools such as JIF, we would have the same drawback: to manually annotate *Sources* and *Sinks* throughout Java source code for each code contribution.

3.4 Fourth Scenario: Untrustworthy developer introducing violations

The fourth scenario is a hypothetical example of the Open Refine project [56]. This system works with messy data, cleaning or transforming it from one format into another. Open Refine is a client-server system written in Java. Furthermore, Open Refine implements an authentication mechanism so that users can access their spreadsheets stored in Google Drive. This project is also open-source and supported by many developers. We provide further details in Section 5.2.1.

Some of these developers might be considered untrustworthy, e.g., a new developer, with few commits accepted. Other characteristics might be necessary to define whether a developer is untrustworthy: (i) a high number of rejected or reverted commits, (ii) a low activity profile in GitHub, (iii) the developer is a stranger to a particular software community, (iv) or because the developer was fired.

These developers' code contributions should be checked carefully. For the Open Refine project, we assume that one of its developers is untrustworthy. Among several development tasks, this developer improved the authentication mechanism between Open Refine and Google Drive. Listing 3.4 illustrates a snippet of code of his contribution. The shaded line in green (line 3) represents the added code, whereas shaded line in red (lines 4 and 5) represent removed code.

Code 3.4: Removing public check

```
1  void doInitializeParserUI(ServletRequest request,
2    ServletResponse response) {
3      String token = TokenCookie.getToken(request);
4      boolean isPublic = "true".equals(getProperty("isPublic"));
5      String token = isPublic ? null : TokenCookie.getToken(request);
6      ...
7  }
```

Notice that before this code contribution, there was a verification checking whether a token is public. This verification was removed by the untrustworthy developer, which might compromise user privacy since `TokenCookie.getToken()` returns sensitive information. Hence, sensitive user information of existing code mistakenly flows to an untrustworthy developer code contribution. In this case, it might result in private user spreadsheet access without proper permission.

For instance, if this developer is fired from a company, we should check his code contributions to find violations. In this scenario, the untrustworthy developer has 813 commits with 1027084 additions and 589028 deletions to be manually reviewed [20]. A code reviewer should search for sensitive information leak in each of these commits to check the flow of information to the execution of their lines.

Using JOANA for this scenario would require manually labeling lines added by the untrustworthy developer for his 1027084 additions spread across 813 commits. Besides that, sensitive information needs to be labeled for these different program versions. Thus, using this tool for this scenario would be more time-consuming and error-prone than for the others. In turn, the effort to use Checker Framework or JIF is unfeasible because we would need to manually annotate security types for 813 different versions.

3.5 Summary of existing approaches limitations

Manual code review. By looking at the examples we present in our four scenarios, we can notice that manual code review alone could be time-consuming and error-prone to identify and solve the problems we discuss. Thus, tools that reduce the effort to review code are neces-sary to improve productivity and quality of the task of finding privacy and security violations.

Existing IFC tools. In this work, we attempt to mitigate two IFC analyses limitations: (i) *lack of generality* and (ii) *time-consuming* and *error-proneness*.

The *lack of generality* drawback regards the IFC tools that are designed to work only for certain technical domains, such as Android. Therefore, to use these tools to detect the problems of our scenarios, we would have to do many changes so they would not be specific to their original domain anymore.

The *time-consuming* and *error-proneness* drawback concerns IFC tools that demand developers to manually specify the information to be protected. Developers can manually label instructions as *High* or *Low* and run the IFC analysis to check whether there is a flow between them [13]. Alternatively, developers can manually associate labels to program elements in the form of security types [14] or annotations [25]

On the other hand, some IFC tools demand developers to specify the information to be manually protected. Developers can manually label instructions as *High* or *Low* and run the IFC analysis to check whether there is a flow between them [13]. Alternatively, developers can manually associate labels to program elements in the form of security types [14] or annotations [25].

However, the use of such tools might be *time-consuming and error-prone*.

In this chapter, we illustrate four scenarios that contain problems we want to tackle in this work. Additionally, we discuss some limitations of existing approaches to solve such problems. Therefore, to address the *time-consuming* and *error-proneness* limitation of these approaches, we define a policy language named Salvum and a tool to enforce specified constraints. We go into more details in Chapter 4.

4 Salvum

To solve the problems discussed, we propose Salvum [17] to allow developers to declaratively express constraints to protect confidentiality and integrity of sensitive information from code contributions. To tackle the *lack of generality* problem, we aim at specifying and enforcing constraints for systems of different technical domains. Furthermore, to solve the *time-consuming and error-proneness* problem, we design Salvum to declaratively specify sensitive information and code contributions so that our tool can automatically label them. Moreover, our language helps developers to reduce the effort of manual code reviewing by diminishing the need to read several lines of code through different software versions.

Our goal with Salvum is to find confidentiality and integrity violations of sensitive information when a developer changes the source code (code contribution) or when this information leaks to specific classes (e.g., external libraries). Thus, in these cases, Salvum is capable of determining the lines of source code that a violation of a constraint happens.

To better illustrate how Salvum works, we discuss the general idea of our approach in Section 4.1. Furthermore, we explain Salvum and its main constructs in Section 4.2. Finally, in Section 4.3, we detail Salvum implementation logical steps, which includes how it deals with Git and JOANA, and processes the results.

4.1 General idea

In the context of collaborative software development, it would be useful to write a set of constraints and check them to protect confidentiality and integrity of sensitive information that might flow within a system. This verification process could happen before integrating code contributions or after integration into a central repository. For example, in case we want to prevent violations from happening (e.g., for the scenario of Section 3.1), it would be essential to enforce the specified constraints before merging the code contributions. On the other hand, in case the code contributions have already been merged, like those associated to an untrustworthy developer like in our fourth scenario (Section 3.4), we could detect the constraints after the integration into the central repository.

In this manner, a developer (e.g., code reviewer) could use Salvum to specify a set of

constraints for a system to avoid the problems we discuss in Chapter 3. Our language provides means to reference program elements that might store sensitive information (e.g., password, location) and code contributions. For instance, we can specify that the code contributions of an untrustworthy developer cannot read or write a password. In this context, Salvum automatically identifies whether there are violations of the specified constraints within developers' code contribution.

We can use our tool to enforce specified Salvum constraints before or after merging code contributions, which we call *backward* and *forward*. The former concerns the case in which we should check the history of a software project. For instance, similarly to the scenario of Section 3.4, consider that a company fires a developer due to misbehavior after some time of software development. This developer has contributed with several commits across that period. Thus, it could be interesting to check potential sensitive information violations in all these commits. To do that, one could write constraints in Salvum to specify that sensitive information should not flow to the code added or changed by these commits. Moreover, we could also write constraints to assure that only a given information can flow to or be changed by code contributions. For example, if we assign to a developer a task to include logging operations, we would want to guarantee that her code contributions do not access sensitive information because they might leak through these logging operations. Therefore, Salvum can automatically detect violations of these constraints in the history of this project.

The latter (*forward*) regards the case in which we should check code contributions before being integrated into the production repository. For example, consider an open-source project that receives many code contributions (e.g., pull requests, commits) from different developers, which possibly are naive or untrustworthy. Thereby, a developer could write constraints in Salvum to check whether sensitive information leaks to these code contributions. Then, before integrating the code contribution into the production repository, we can ask the developers to fix any violations found by Salvum. Figure 4.1 depicts this general idea.

Equally to the fourth scenario (Section 3.4), without Salvum, a specialist (eg., experienced developer or reviewer, system architect, etc.) should manually review the commits made by the fired developer. Otherwise, she should manually label sensitive information as well as the commits' code and run an IFC tool for each system version for the *backward* approach. Similarly, this specialist would need to review code contributions manually before merging into the production repository (*forward*).

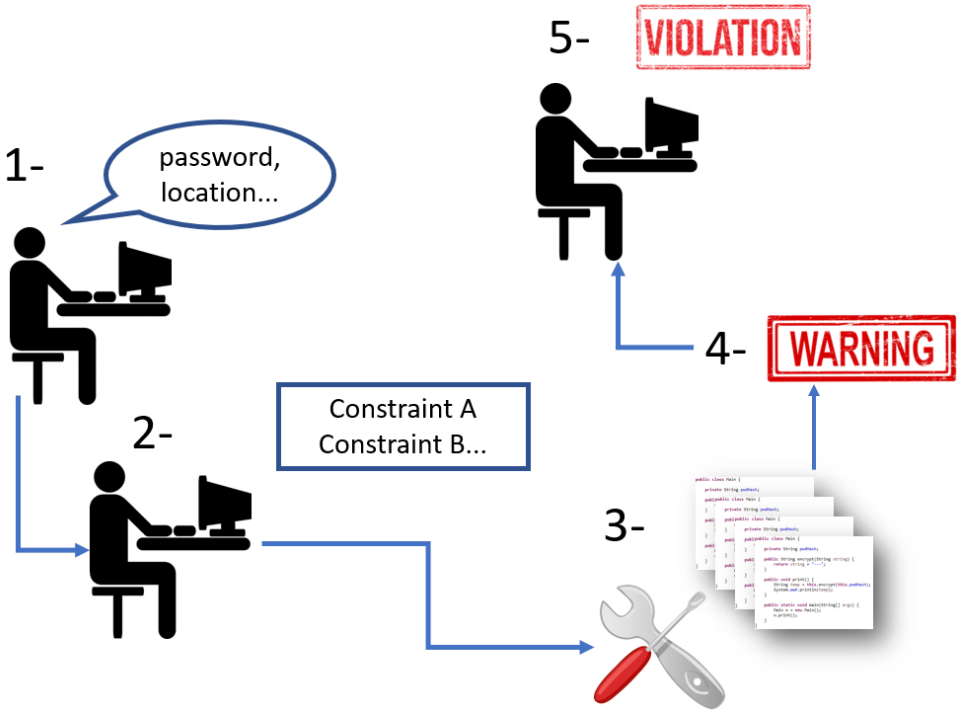

Figure 4.1: General Idea

In summary, our solution works as follows:

1. The specialist identifies sensitive information;
2. The specialist writes constraints;
3. Our tool checks source code adherence to these Salvum constraints;
4. Our tool reports potential violation warnings;
5. The specialist determines which warnings are real violations.

4.2 Language specification

In this section, we discuss examples of constraints that we could use for the scenarios of Chapter 3. Then, we explain the main constructs and identifiers of our language.

4.2.1 Examples

We can use Salvum to specify a constraint with the goal to detect the problem introduced by the code contribution of Listing 3.1 in Chapter 3. In that example, a developer implemented a code contribution that introduces a `setCookie` call. This method implicitly leaks the user password. Since the code contribution was already merged in the Gitblit repository, we con-sider our *backward* approach, that is, we aim at searching for privacy and security violations

throughout the repository history and different versions of Gitblit. Listing 4.1 illustrates a constraint we can write for this scenario. First, we determine a set of existing fields that could store sensitive information, such as the `password` and `locality` defined in `UserModel` class. The `noflow` construct specifies that this sensitive information (and only it) cannot flow to any line changed or added by the code contribution, which is identified as `Contribution`. This identifier is a Salvum keyword and it designates a specific set of code contributions that we want to analyze.

Code 4.1: Protecting sensitive information flows to code contribution

```
UserModel {password, locality} noflow Contribution
```

In Listing 4.1, the `Contribution` identifier represents a set of Gitblit commits we want to analyze. For example, Figure 4.2 depicts three versions in the commit history represented by `Contribution`. Therefore, this constraint enforces that `password` and `locality` do not flow to the code committed throughout these three versions.

Figure 4.2: The `Contribution` identifier and project versions

The constraint in Listing 4.1 helps to detect sensitive information flows to code contributions. However, it determines nothing about code contributions writing to that sensitive information. Thus, we might also define the constraint in Listing 4.2 to detect the points where code contributions make changes to sensitive information. It states that code contributions cannot change sensitive information initially stored in the field like `password` and `locality` fields defined by the type `UserModel`. Analogously to the `noflow` example, this constraint can also reduce the effort to manually review different Gitblit versions.

Code 4.2: Protecting code contribution changes to sensitive information

```
Contribution noset UserModel {password, locality}
```

The example in Listing 4.3 regards a constraint specifying the information that can flow to code contributions defined by a pull request [23]. It states that only the information stored in `login`, `nickname`, or `countrCode` can flow to the commits with messages containing `#921`, which uses the `PullRequest` identifier. Here we assume that only the commits from a given pull request have `#921`. Thus, Salvum might be used to ensure that only this `UserModel` information can flow to the code of these commits.

Code 4.3: Specifying information that can flow to code contributions

```
UserModel{login,nickname,countrCode} flow PullRequest
where PullRequest = {c | c.message.contains("#921")}
```

This constraint demonstrates that Salvum is capable of enforcing it for many different system versions. We explain in detail how Salvum does it in Section 4.3. This constraint is particularly useful to ensure that only a information can flow to the resulting code contribution. The code contributions to be analyzed might resolve an issue [57], a task execution, or a feature implementation, as long as they have an identifier in their commit messages, such as `#921`. By using this constraint, a code reviewer would not need to manually review these different system versions identified by the commits associated with `PullRequest`. Besides that, there would be no need to manually label sensitive information or the code contribution.

Additionally, we can use the `set` construct to specify the information that can be altered by a code contribution. In Listing 4.4, we show an example of a constraint specifying that the code contribution identified by the commit message containing `task87` can alter only the information stored in `login`, `nickname`, and `countrCode`. Therefore, this code contribution cannot write to any other field.

Code 4.4: Specifying information that can be altered by code contributions

```
UpdateUserTask set UserModel {login, nickname, countrCode}
where UpdateUserTask = {c | c.message.contains("task87")}
```

Besides the `contains` clause, Salvum also supports `c.author` to define code contributions. For instance, suppose that Bob was fired, and his company wants to check his commits to ensure that his code only accesses public user information. Therefore, Listing 4.5 defines a constraint to verify that the code Bob committed accesses information stored only in the `login`, `nickname`, and `countrCode` fields of `UserModel` class. Therefore, any access to other fields would represent a violation. The `UntrustworthyBob` identifies the code committed by Bob. Analogously, we can specify a constraint using the `set` construct.

Code 4.5: Specifying information that can flow to code contributions

```
UserModel{login, nickname, countrCode} flow UntrustworthyBob
where UntrustworthyBob = {c | c.author("Bob")}
```

Moreover, we can define constraints using commit hashes directly to represent a code contribution. For example, Listing 4.6 states that only information stored in `login`, `nickname`, and `countrCode` can flow to the code contribution `ServletsTask`, which is defined by the commit hash `efdb2b3d0`. Thus, Salvum helps to ensure that this code contribution only accesses public user information.

Code 4.6: Specifying information that can flow to a specific commit code

```
UserModel{login, nickname, countrCode} flow ServletsTask
where ServletsTask = {efdb2b3d0}
```

Furthermore, we can reference code contributions and use Salvum's negative constructs. In Listing 4.7, we write a constraint specifying the information that cannot flow to code contri-

butions defined by a pull request. It states that the information stored in `password`, `email`, and `location` cannot flow to commits with a message containing `#921`. Thus, Salvum might be used to ensure that this sensitive information cannot flow to the code of these commits.

Code 4.7: Specifying information that cannot flow to code contributions

```
UserModel{password, email, location} noflow PullRequest
where PullRequest = {c | c.message.contains("#921")}
```

When we define this constraint, it is not necessary to review the source code manually to find a potential user password, email, or location leak throughout the different system versions identified by `PullRequest`. Furthermore, our tool automatically labels this sensitive information and these different versions, which reduces the time-consuming task of manually annotating them (Section 4.3).

Salvum also considers constraints specifying that only public information can flow to the methods of a certain class, which means that other information cannot flow. For this purpose, we use the `flow` construct. In Listing 4.8, we illustrate a constraint specifying that public information initially stored in the `login` and `name` fields of `UserModel` class can flow to `Logger` class methods, which uses the `Log` identifier. The `flow` construct means that **only** the information stored in `login` and `name` can flow, that is, no other information can. In turn, this constraint does not specify anything about `login` and `name` flowing to other methods. Therefore, it would not represent a violation to this constraint.

Code 4.8: Specifying information that can flow to `Logger` methods

```
UserModel{login, name} flow Log
where Log = {Logger.error(), Logger.info(), Logger.warn()}
```

Our goal is to consider the `Logger` class and its subclasses in the constraint specification. Moreover, in case of `Logger` is an interface, we would consider the classes that implement it. However, the current version of our language only supports `Logger` as a class. Therefore, one needs to specify `Logger` subclasses in the constraint as well.

This constraint might help to assure that only public information flows to public outputs, such as `Log` methods. In case this constraint is violated, a code reviewer knows precisely where within the source code it happens and also which contribution introduced such violation. We could also use the `set` construct to determine which method or contribution is allowed to write only on a set of fields that store public information.

Analogously, we can use the `noflow` construct to protect sensitive information. Listing 4.9 specifies a constraint to state that a sensitive information like `password` cannot flow to public outputs such as `Logger` methods.

Code 4.9: Specifying sensitive information that cannot flow to `Logger` methods

```
UserModel{password} noflow Log
where Log = {Logger.error(), Logger.info(), Logger.warn()}
```

At last, we can also use the constructs `noset` and `set` with these specifications of potentially dangerous external libraries like the `Logger` methods. We explain Salvum constructs in the next section.

4.2.2 Constructs

Salvum supports four main constructs: `noflow`, `noset`, `flow`, and `set`. The `noflow` construct specifies that sensitive information cannot flow to code contributions. This construct does not determine anything about other information. For instance, if we define that a password cannot flow to a code contribution, we are saying nothing about additional information, such as location. Therefore, this construct goal is to guarantee the confidentiality of information.

In this context, the `noset` construct complements `noflow`. It specifies that sensitive information cannot be altered by code contributions. Again, we are not specifying anything about other information. Thus, we can determine that a password cannot be altered by a set of code contributions. In turn, this construct's goal is to guarantee integrity of information.

In contrast to `noflow`, Salvum supports the `flow` construct. It specifies that only a specific information can flow to code contributions. Therefore, no other information can flow to these contributions. For example, we can specify that only user login can flow to a code contribution that implements logging operations.

Also in contrast to `noset`, we provide the `set` construct. It specifies that only a specific information can be altered by code contributions. For instance, we can specify that only an email message can be altered by a code contribution that implements an indentation feature for an email system. Thus, this code contribution cannot alter other information (i.e., user email address).

The goal of `flow` and `set` is to enforce the Principle of Least Privilege [41]: a code contribution should use the least amount of privilege necessary to correctly complete its execution. Salvum also supports different identifiers that represent code contributions. We explain them in the next section.

4.2.3 Representation of code contribution

Salvum allows the representation of code contributions in three distinct ways: explicitly, implicitly, and through operations listing.

Explicitly. The first representation defines a code contribution as a set of commits that contains a particular message keyword. For example, if we want to enforce a constraint for all

the commits that solve a specific issue, we can use a `where` clause followed by the condition, as shown below:

```
IssueFixing where IssueFixing = {c | c.message.contains("issueId")}
```

The `IssueFixing` identifier represents the code contributions. The `where` clause establishes a condition for these contributions to be identified. In this case, the condition represents the set of commits that contains messages with the `"issueId"`. The `c` identifier is a keyword that we must use when expressing declarations regarding commits. The `message.contains()` statement represents each commit that contains a message with a particular keyword (`"issueId"`, for example). In turn, we must assume that only the commits related to a specific issue contain the `"issueId"` in their messages.

Salvum also supports other representations of code contributions for the `where` clause. For example, we can define a code contribution as being the set of commits submitted by a particular author, as follows:

```
FiredAuthor where FiredAuthor = {c | c.author("AuthorId")}
```

The `FiredAuthor` identifier represents the code contributions. In this case, the condition determines they are the set of commits submitted by the author identified by `"AuthorId"`. Besides that, we can directly declare that a code contribution represents only one particular commit:

```
HarmfulCommit where HarmfulCommit = {5odle482}
```

In this case, the `HarmfulCommit` represents one commit identified by the hash `5odle482`. We could have represented this particular commit using only `HarmfulCommit = {5odle482}`. However, we intend to reuse the `HarmfulCommit` definition for other constraints. Therefore, we opted to represent a particular commit using the `where` clause. Further, we can implicitly specify code contributions.

Implicitly. We can specify code contributions via the comprehension of a commit set. Thus, we implicitly define a code contribution with the identifier `Contribution`. This identifier represents a set of commits that we want to analyze. This identification is up to the person who is examining the system. Therefore, `Contribution` does not explicitly restrict a particular set of commits like using the `where` clause, as we explained above.

```
User {password} noflow Contribution
```

In this constraint, the `password` represents the sensitive information whereas the `Contribution` identifier implicitly defines the set of commits to be analyzed.

Operations listing. Salvum also supports that we specify a list of potentially dangerous operations for the `where` clause. For example, we can identify a code contribution as a list of `Logger` methods, as shown below:

```
Log where Log = {Logger.error(), Logger.info(), Logger.warn()}
```

The `Log` identifier represents all the calls to these three `Logger` methods.

4.2.4 Discussion

In the current version of our language, we do not support the specification of sensitive information that is stored only on local variables. As we show in the examples of Section 4.2.1, Salvum only allows the specification of information that is initially stored on class variables. We intend to solve this limitation in future work by allowing that developers write constraints specifying local variables. Additionally, Salvum does not support the specification of method parameters and the return of a method. We also intend to address this drawback in future work. To be able to detect violations of sensitive information regarding these situations in our evaluation (Chapter 5), we could turn a local variable into a class variable. We could also assign the return of a method or its arguments to class variables.

We might use `noflow` and `noset` to guarantee the Principle of Least Privilege and sensitive information confidentiality and integrity. However, these constructs are permissive. Figure 4.3 illustrates an example of their limitation. Notice that in case we write only the constraint of Listing 4.7, Salvum enforces that any information stored in user `password`, `email`, `location` does not flow to the specified pull request's code. However, this constraint (using `noflow`) does not restrict `phone` and `address` from flowing to the pull request code, that is, this sensitive information can flow without any restriction.

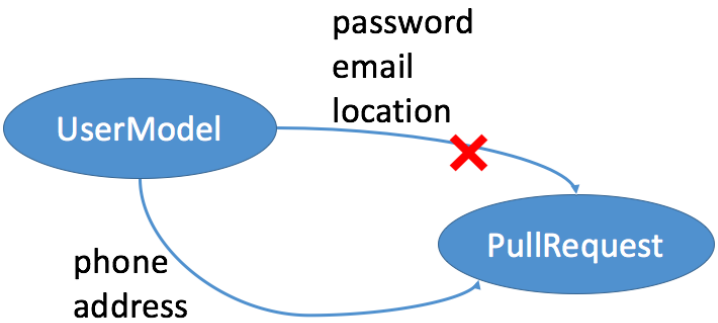

Figure 4.3: `noflow` and `noset` are permissive

This issue might be mitigated if experienced developers write the constraints in Salvum. Indeed, this kind of developer usually is responsible for manually reviewing code [6]. Moreover, forgetting to specify sensitive information is also a drawback of Information Flow Control analysis tools that require developers to label program parts manually [33, 14]. However, we acknowledge that despite the restriction to specific technical domains, such as Android, a tool like Flowdroid [10] does not introduce this issue since it automatically identifies sensitive information using auxiliary tools [58].

Due to advantages and disadvantages of the presented constructs, we provide both positive (`flow` and `set`) and negative (`noflow` and `noset`). Figure 4.4 shows a summary of these constructs and their respective semantics.

By writing the constraints presented in this section using our policy language, develop-

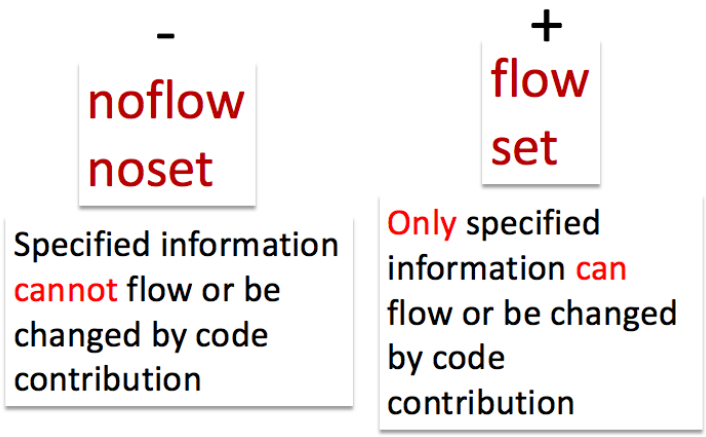

Figure 4.4: Semantics of constructs

ers can specify exactly the only information that can flow or be changed, or illegal information flow to systems of different domains, which helps to circumvent the *lack of generality* problem we point out in Section 3.5. Furthermore, developers might have some effort to write the constraints in Salvum, but they do not need to manually associate labels to program parts before reviewing code to find violations. This helps to mitigate the *time-consuming and error-proneness* problem of tools like JOANA [33] and Jif [14], as we also explain in Section 3.5. Thus, using Salvum might increase productivity for the execution of finding violations tasks. We provide more details in Chapter 5.

Finally, our constraints define boundaries for code contributions or classes. Salvum sees these elements as software modules [59]. In Section 4.3, we explain it in more detail as well as Salvum's implementation.

4.3 Language implementation

An additional contribution of this work is the development of a tool to automatically enforce the Salvum constraints that we specify. To achieve that, we use the JOANA [33] tool, which provides IFC analysis, as we explained in Section 2.4. This way, we can use these analyses to detect information flows that might violate the specified constraints. Figure 4.5 illustrates our tool components and their connections.

We implement a parser for our language using ANTLR [60], which is a tool to generate parsers that can build and walk parse trees. This parser validates the syntax of the constraints. Furthermore, it receives a text file where we specified the constraints and generates a corresponding JSON file, which we use as input to perform the annotation procedure and analysis.

Besides that, we implement a mapping generator to execute a preprocessing for the source code of the system to be analyzed. The resulting mapping holds information about relevant lines of code, such as those that code contributions change or those that declare a variable that contains sensitive information. Our tool uses the information of the line of code to

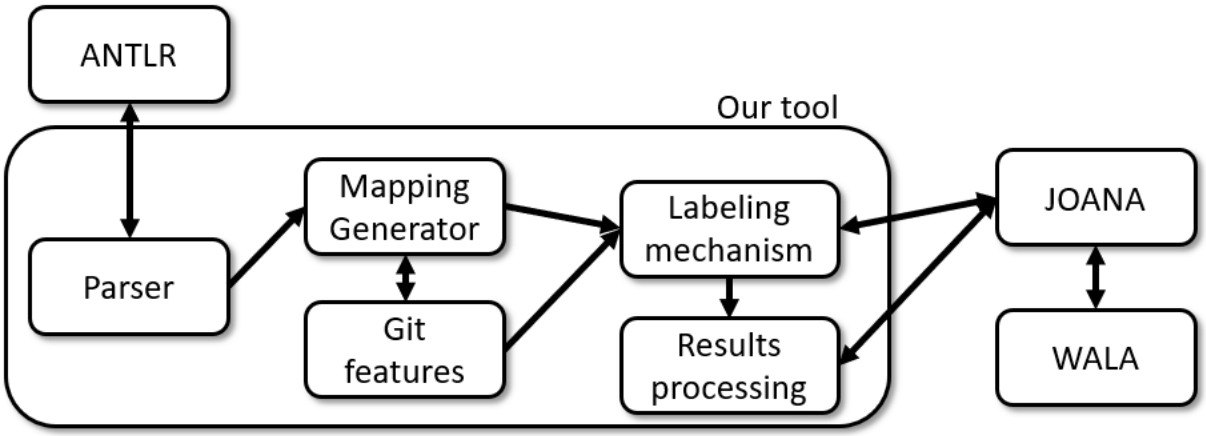

Figure 4.5: Overview of our tool

correctly annotate it as *High* or *Low* depending on the specified constraint.

Additionally, we implement a mechanism that allows us to automatically annotate sensitive information and code contribution. This mechanism mitigates the *time-consuming and error-proneness* problem of plain IFC tools because we do not need to manually associate instruction with labels, as discussed in Section 3.5.

As we explained in Section 2.5.1, we use four Git features in our language implementation: `diff`, `blame`, `checkout` and `log`. Hence, we implement a module to integrate Git with our tool, and consequently provide support for these Git features.

For each project version that we analyze, we have to build a System Dependence Graph. Thus, we implemented a custom generator that is capable of automatically creating at least one SDG for each version that contains the code contributions we want to analyze. Since JOANA permits that we provide only one entry method, we implemented support for multiple entry methods at once. Therefore, we can use JOANA to create an SDG from a list of entry methods. This implementation allows us to build larger SDG, which consequently encompasses a more substantial part of the source code. For example, in case a code contribution is scattered across many classes and methods, we would not be able to enforce constraints for the whole code contribution because the unique entry method would result in a small SDG that would not have nodes and edges corresponding to that code contribution. Thus, in these cases, we provide multiple entry methods. However, we acknowledge that we do not have a mechanism to check whether the full code contribution is represented by the SDG created. We discuss this threat in Section 5.5.

In the next sections, we explain Salvum implementation steps: *Preprocessing*, *Generating SDG*, *Labeling*, *Analyzing*, and *Results processing*. Figure 4.6 illustrates these steps, their inputs and outputs. Our tool uses JOANA API to generate the SDGs and to execute the Information Flow Control analysis (*Analyzing* step).

Differently from a typical JOANA execution, we have the Preprocessing, Labeling (automatically), and Result Processing steps. Nonetheless, we observe that the execution time

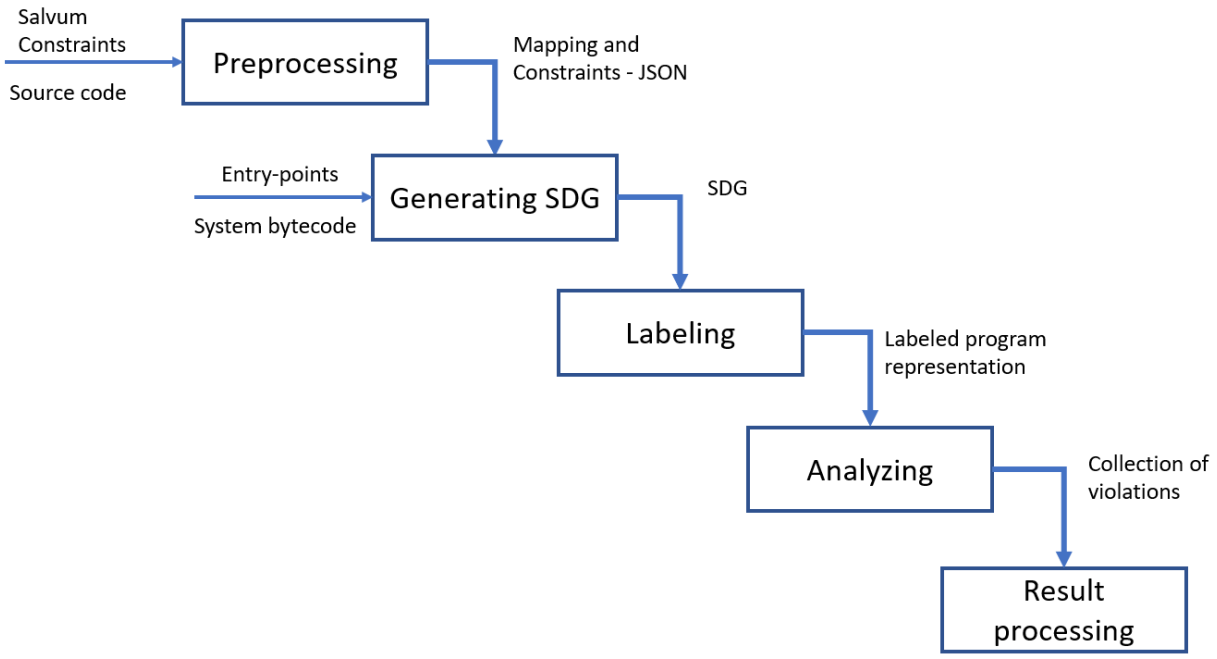

Figure 4.6: Five steps

overhead is insignificant. To execute the additional steps, our implementation needs a few seconds regarding the projects we selected for our evaluation (Chapter 5). In contrast, Generating an SDG might take hours.

4.3.1 First step - Preprocessing

As shown in Figure 4.6, this step receives as inputs the specified constraints and the source code of the system we want to analyze. There are two different preprocessing procedures, which depend on the constraints.

The first one regards the specification of code contributions as **operations listing** (Section 4.2.3). In this case, we define an algorithm to find occurrences in the source code of calls to methods that are specified by the constraint. These calls could be specified as in Listing 4.8. In this context, our algorithm holds information about the lines of code of these occurrences throughout the source code. Therefore, the output of this step is a mapping of classes and line numbers. For example, if there is a call to a `Logger` method in a class called `Main` in lines 11 and 12 plus another call in a class called `Repository` in lines 6 and 16, this step would create the following mapping: `[Main:[11,12]],[Repository:[6,16]]`. This mapping is useful for automatically labeling the source code as a sink or source and mitigate the *time-consuming and error-proneness* problem, which we explain in the *Labeling* step.

The second preprocessing procedure concerns the specification of explicit and implicit code contributions (Section 4.2.3). Therefore, we need to deal with program history in Version Control Systems [61]. For now, Salvum supports Git [42]. In this context, the Git `diff` command generates a `diff` file, which contains the differences between existing code and

code contribution. We define an algorithm that analyzes the `diff` file to obtain the classes and their source code line numbers where changes occur. Thus, the output of this step is a mapping of classes and line numbers. For example, a mapping for the snippet of code in Listing 3.1 is [`RootPage:5`]. This mapping is also useful for automatically labeling source code to execute an IFC analysis and detect illegal flows. This step is the same for the two main approaches for finding violations: *backward* and *forward*. The main difference between the mapping we generate for code contributions to the one we do for classes is that the former contains lines of code changed within a commit whereas the latter contains lines of code that represent calls to methods we want to stop from leaking sensitive information.

In particular, there is an additional difficulty for code contributions. Since we need to analyze different software project versions, we need to obtain the target version before building the output mapping. To achieve this goal, we use the `checkout` Git command either automatically or manually depending on the project and constraint.

Additionally, our parser validates the syntax of the constraints and generates a corresponding JSON containing the specification of sensitive information and code contributions.

4.3.2 Second step - Generating SDG

In this step, we create the necessary SDGs. We provide as inputs the set of entry methods and the compiled system. A typical entry method of a Java system is its main method. However, some systems do not define a main or they define many mains. Therefore, for these cases that do not present a unique main, one has to manually choose a set of entry methods. This task could be done by identifying the set of methods that encompasses both code contribution and sensitive information. These inputs are mandatory to correctly create an SDG. In the current version of our tool, we manually choose a set of entry methods with the goal to encompass the code contribution. This selection procedure depends on the system we want to analyze and its code contributions. For example, consider the following constraint:

```
User {password} noflow Contribution
```

In Listing 4.10, we illustrate an example that would demand us to provide two entry-methods to correctly generate an SDG. A developer added the lines 4, 5, 13, and 14 in her code contribution. Since there are changes in the `authUser()` and `editUser()` methods, we provide them as entry-points to the JOANA SDG generation mechanism. Consequently, we can create an SDG that encompasses the changes introduced by this code contribution. However, we acknowledge that manually picking entry-points is a time-consuming task that could be at least, partially, automated. We provide more details in Sections 3.5 and 7.3.

Code 4.10: Entry method selection

```
1   class AuthenticationManager {
2     void authUser(String login, String password) {
3       ...
4++       Logger.info(" User " + login + " with pwd "
5++        + password + " logged successfully");
6     }
7     ...
8   }
9
10  class UserDAO {
11    void editUser(String login, String password) {
12      ...
13++      Logger.info(" User " + login " + with pwd "
14++       + password + " updated successfully");
15    }
16    ...
17  }
```

Depending on the constraint, we could need to generate many SDGs in this step. For instance, recalling the fired developer e xample: she could have submitted many code contributions, and consequently, there are many different system versions to analyze. Thus, for each version, we automatically create a different SDG. At last, this step output is a set of one or more SDGs.

4.3.3 Third step - Labeling

This step is essential to reduce the *time-consuming and error-proneness* problem. Our implementation automatically annotate information and code contributions so that it can check whether there is a flow between them in either direction depending on the specified constraints. JOANA provides an annotation mechanism that allows us to label a set of program parts with different security levels (e.g., *High* and *Low*).

This way, JOANA is capable of annotating the instructions that correspond to a particular program part, which in our case is identified by its source code line n u mber. Since JOANA is built upon the WALA framework [30], it holds a list of intermediate representations that represents instructions of a program. Mainly, this data structure represents the instructions of a method. Therefore, JOANA is capable of identifying in which SDG node an intermediate representation is included.

Based on the constraints written in Salvum and the first step output mapping explained

above, our implementation automatically labels SDG nodes that represent sensitive information for `noflow` construct, as *High*. On the other hand, SDG nodes originated from source code lines identified in the mapping as part of code contributions are automatically labeled as *Low*. To deal with positive constructs (`flow` and `set`), the labeling strategy is the opposite of `noflow` and `noset` constructs since they are dual [29]. Thus, code contributions are labeled as *High* and the public or sensitive information is labeled as *Low*.

For instance, consider again Listing 3.1 and Listing 4.1. This constraint uses the `noflow` construct. Therefore, the *High* label is automatically associated with `UserModel` sensitive information stored in `password` and `locality` and the *Low* label is associated with code contribution. Notice that if we use the `noset` construct for this constraint, the *High* label would be associated with the code contribution and the *Low* label would be associated with `UserModel` information. Figure 4.7 depicts this scenario.

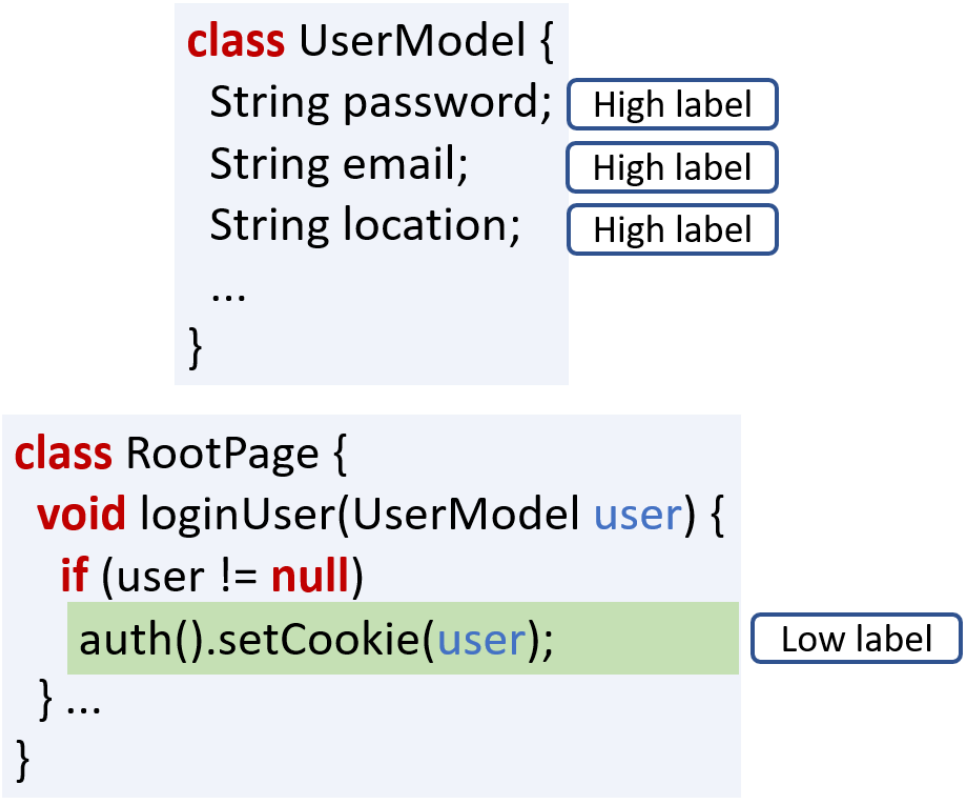

Figure 4.7: High and Low labels

4.3.4 Fourth step - Analyzing.

In this step, we execute JOANA [13] IFC analyses. Thus, Salvum automatically runs these analyses to verify source code's adherence to the specification of Salvum constraints. More specifically, it checks whether there are possible paths from an SDG node labeled as *High* to another one labeled as *Low*.

As a result of the analyses execution, we obtain a collection of potential violations, which we provide as an input to the last step: *Results processing*.

4.3.5 Fifth step - Results processing

Salvum creates a report to inform violations occurrences. For instance, consider Listings 3.1 and 4.1 again. The resulting report would inform a violation, like the following:

```
Illegal flow from UserModel.password, UserModel.locality
          to RootPage:27 at commit hsf354ks
```

In this way, we know that there is a violation of `UserModel` sensitive information confidentiality to the line 27 of `RootPage` class in the system project version identified by the commit hash `hsf354ks`. Therefore, we can determine that this code contribution introduces a violation in a *backward* approach, that is, in the actual project history. On the other hand, we could either reject this code contribution or ask for a fix in a *forward* approach. We manually process these results to find false-positives.

In this chapter, we present the specification of our policy language as well as a tool to enforce the constraints we could write. Moreover, we provide Salvum implementation in our online Appendix [20]. In the next chapter, we explain the evaluation we performed considering the use of Salvum for selected software projects.

5 Evaluation

To evaluate our policy language presented in Chapter 4, we performed an empirical assessment focusing on finding violations by writing Salvum policies and constraints for selected software projects and checking whether it can help developers to reduce the effort of manual code review. Furthermore, we evaluate JOANA concerning precision, recall, and accuracy of its IFC analyses.

In Section 5.1, we discuss the Goal-Question-Metric design [62] that drives our evaluation in Section 5.1. Additionally, in Section 5.2, we perform the first e valuation considering highly active and well-supported software projects. To evaluate Salvum for other kinds of projects, in Section 5.3, we explain our second evaluation with low activity and poorly supported software projects. We make this distinction between highly and low active projects because we believe that we should have different results. For the former, we expect to detect a few violations whereas more violations considering the latter. In Section 5.4, we investigate whether JOANA's IFC performs well when we run it against an existing benchmark. This JOANA analyses evaluation brings us insights about the kinds of violations that we can identify or miss. At last, we discuss the threats to validity in Section 5.5.

5.1 Goal, Questions, and Metrics

We use the Goal-Question-Metric (GQM) [62] design to better drive the evaluation process. Our goal is to check whether we can detect privacy and security violations for real software projects. Besides that, we aim to determine if Salvum can reduce the effort to find these violations. We also have the additional goal of measuring JOANA [13] precision, recall, and accuracy. Table 5.1 summarizes our GQM approach.

We define three major and two minor research q uestions. **Q1** is crucial to bring evidence that Salvum can, indeed, find violations of sensitive information for existing software projects. This answer would help developers to prevent violations that could be harmful to sensitive information. We investigate software projects of different technical domains to check whether Salvum is useful in this context. We write some constraints in Salvum (e.g., passwords cannot flow t o c ode c ontributions) a nd r un o ur t ool t o fi nd vi olations on so ftware pr oj ects. To better

Table 5.1: GQM to assess Salvum

Goal	
`Purpose`	Evaluate Salvum regarding
`Issue`	privacy and security violations
`Object`	for code contributions
`Viewpoint`	from a code reviewer viewpoint
Questions and Metrics	
Q1- Can Salvum detect code contribution violations to sensitive information?	`-Number violations warnings (NVW)` `-Number of violations (NV)`
Q1.1- Do developers solve violations before code contribution merging on GitHub?	`-Number of violations before merging code contributions (NvC)`
Q1.2- Are unmerged code contributions on GitHub related to violations of sensitive information?	`-Number of unmerged code contributions containing violations (NrC)`
Q2- Can Salvum reduce the effort to find violations of specified constraints in a set of code contributions?	`-Source lines of code to analyze (SA)` `-Project versions to analyze (PA)`

Q3- How does JOANA [13] perform when being applied to IFC issues regarding precision, recall, and accuracy of its analyses?	`-Number of true-positives (TP)`
	`-Number of false-negatives (FN)`
	`-Number of false-positives (FP)`
	`-Number of true-negatives (TN)`

understand the answer to **Q1**, we define two minor questions: **Q1.1** and **Q1.2**. The former aims to explain whether developers fix violations before merging code contributions. The latter intends to unveil whether code reviewers found violations and discarded code contributions before merging them into the repository.

Answering **Q2** might help developers to decide whether it is more efficient to perform (i) only a manual code review, (ii) to use only Salvum, or (iii) to use both to bring evidence of the violations introduced by a set of code contributions. Therefore, we analyze the source lines of code and the number of different software versions we would need to review for finding the violations. We approximate effort in terms of the lines of code and project versions we need to manually review. Thus, more lines and project versions mean more revision effort. We acknowledge that lines of code do not approximate well the programming effort. Thus, it is likely that the same occurs to an approximation of code reviewing effort because we could take more time to review one line that contains complex computations than many other more uncomplicated lines. However, we are not measuring the effort to review one line comparing to another one. In this context, we do not exactly take that risk in our assessment.

On the other hand, **Q3** concerns JOANA precision, recall, and accuracy. The answer to **Q3** is useful to determine the cases JOANA can detect violations. For example, if JOANA analyses present low rates of recall, we can conclude that we may have missed violations due to limitations of this tool. Thus, we run this IFC tool against the SecuriBench Micro [19], which is a set of micro-benchmarks intended for evaluating security of web-based applications.

To answer our research questions, we define a set of simple metrics, as shown in Table 5.1. Regarding **Q1**, we use the `Number of violation warnings (NVW)` and `Number of violations (NV)` metrics. The former regards the number of violation warnings that Salvum finds among different project versions whereas the latter represents the subset of found violations which were confirmed by the selected systems developers. To better understand **Q1** results, we manually analyze a set of submitted pull requests for each selected project to check whether our approach misses violations. Thus, we use the `NvC` metric to answer **Q1.1**. Additionally, we manually check rejected pull requests to know if they were not accepted due to violations in our context. We provide further details in Section 5.2.3.

To answer **Q2**, we consider two metrics:

- `Source lines of code to analyze (SA)`
- `Project versions to analyze (PA)`

By using our tool, a reviewer needs to manually check whether a warning represents a real violation. Therefore, we use the `SA` metric to measure the number of lines of code that this reviewer should manually analyze to determine whether the code contribution indeed introduces the warned violation.

Besides that, our tool warns the presence of potential violation for each project version. Therefore, a reviewer would not need to manually review all the versions, but only those with

violation warnings. Thus, we use the `PA` metric to measure the number of projects versions we need to manually analyze to confirm violations of the specified constraints. For example, if we use our tool to analyze ten project versions, and it warns violations only for two versions, we would not need to manually examine the other eight versions to confirm the warnings. Figure 5.1 illustrates an example that explains our intent with these metrics.

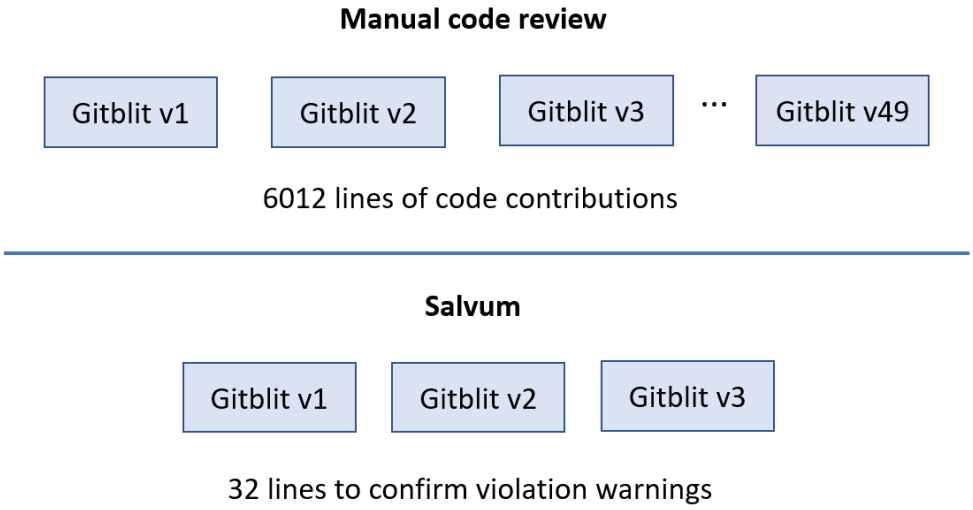

Figure 5.1: Metrics for **Q2**

In the case a reviewer needs to consider the full code contributions, she would have to manually review 6012 lines of code scattered across 49 different project versions to find violations of the specified constraints. On the other hand, Salvum detects warnings in three different project versions. Since our tool brings evidence that there are no violations present on the other 46 project versions, a reviewer would need to manually review 32 lines of code contribution across these versions to confirm that the problem happens. However, JOANA analyses have some limitation, which we discuss in Section 5.5. For instance, it does not support Java reflection.

Regarding **Q3**, we take into account four metrics. The `Number of true-positives (TP)` represents real identified information flows that are violations of the specified constraints. The `Number of false-negatives (FN)` represents the unwarned existing violations whereas the `Number of false-positives (FP)` represents warnings that are not a violation. The `Number of true-negatives (TN)` is the number of violations that JOANA correctly establishes are not a real treat. We provide more details in Section 5.4.

5.2 Assessing highly active and well-supported software projects

In this section, we consider nine highly active software projects. Thus, we explain these projects in Section 5.2.1. Furthermore, we define a set of constraints for them in Section 5.2.2. At last, we describe the results we obtained in Section 5.2.3. Our goal here is to answer **Q1**, **Q1.1**, **Q1.2**, and **Q2**. To do that, we use the *backward* approach (Section 4.1). In this case, it is more interesting than the *forward* approach because the existing project history has much more commits, and consequently merged code contributions.

5.2.1 Selected software projects

For this assessment, we consider the following open-source Java software projects hosted on GitHub: Gitblit [46], Open Refine [56], Voldemort [63], ScribeJava [54], Solo [64], HikariCP [65], Apache Kafka [66], Teammates [67], and Crawler4J [68]. Table 5.2 summarizes some of their numbers, which are approximations because they are rapidly changing. We found these applications by using a simple tool we developed to filter GitHub projects based on their number of stars and fork [20].

Our tool uses the GitHub API [69] to generate a list of GitHub repositories URLs. Each one contains at least 100 stars and 50 forks. We also set that the main programming language is Java and the project README must have the keyword "java". This last condition helps to decrease the number of Android projects. JOANA developers only added support to Android after we started this experiment and it only works for small toy applications. We provide more details in Section 7.2.

In this context, we obtained a list of the first 300 hundred projects returned by the GitHub API. We visited each GitHub project repository webpage to choose the first nine that returned in our searching procedure. Initially, we discarded the projects that are Android applications. Furthermore, we put away projects that we could not find sensitive information by navigating through their source code in GitHub. Hence, the first nine projects that we did not discard are considered in this section. We provide source code and further details in out online appendix [20]. We explain each project below.

Gitblit is a system for managing, viewing, and serving Git repositories. There are two main products within the same project: client and server. This application has approximately 12300 lines of code. Moreover, Gitblit deals with much sensitive information, such as private user data. It is an active project on GitHub containing around 3000 commits and 100 con-tributors. A GitHub contributor is a developer that has committed at least one time within the repository. This project includes around 400 open and closed pull requests.

Open Refine is an application for working with messy data: cleaning it, transforming it from one format into another, and extending it with web services and external data. This

Table 5.2: Selected software projects

Project	KSLOC	Commits	Contributors	Total Pull requests	GitHub stars	GitHub forks
Gitblit	123	3000	100	400	1400	500
Open Refine	75	2700	60	250	5000	900
Voldemort	225	4300	60	350	2000	500
ScribeJava	5	800	90	350	4300	1500
Solo	41	2000	15	190	3900	1700
HikariCP	17	2600	80	250	6000	900
Apache Kafka	113	4700	400	4500	7400	4400
Teammates	145	16600	350	3100	500	1200
Crawler4J	8	300	25	100	2600	1500

system might deal with sensitive data stored in sheets, and it can communicate with Google API informing user login and password. Besides that, Open Refine has roughly 75000 lines of code, and it is an active project with around 2500 commits, 60 contributors, and 250 pull requests.

Voldemort is a distributed key-value storage system. It provides functionalities to allow distribution of data across a set of databases, which could be useful to replicate data in the cloud. This application handles sensitive information such as database password and administrator client instance. Voldemort has around 4300 commits, 60 contributors, and 350 pull request. It is the largest application we selected with 225000 lines of code.

ScribeJava is an OAuth library for Java. It is designed to provide authentication functionalities which means ScribeJava handles sensitive information such as user API key and secret. ScribeJava project has approximately 800 commits, 90 contributors, 350 pull requests, and 5000 lines of code.

Solo is a blogging system with nearly 2000 commits, 15 contributors, and 190 pull requests. It handles sensitive information that contains, user password, email, cookie, and email messages.

HikariCP provides a JDBC connection pool that supports most popular databases. It is capable of creating and maintaining a collection of JDBC connection objects. This application deals with sensitive user information such as his password. It presents around 17000 lines of code, 2600 commits, 70 contributors, and 250 pull requests.

Apache Kafka is a distributed streaming platform to publish, subscribe, process, and store streams of data. It handles passwords, credentials, and keys for security layer. This applications has around 113000 lines of code, 4700 commits, 400 contributors, and 4500 pull requests.

Teammates is a tool for managing peer evaluations and to collect feedback from students. It works as a cloud-based service for educators and students, and many universities currently use its functionalities. This application handles sensitive information like student reg-

istration keys, administrators email, and account. This project is supported by approximately 350 contributors that have already committed 16600 times in a total of 3100 pull requests. Teammates has around 145000 lines of code.

Crawler4J implements a web crawler. Users can provide a set of initial URLs and this tool crawls for linked pages. It needs to handle sensitive information such as proxy passwords and authentication data. This implementation has more than 8000 lines of code. Nearly 25 contributors support Crawler4J with 300 commits and 100 pull requests.

We noticed in the pull requests review on GitHub that there are code contribution review processes for these nine software projects. We also noticed that two or three experienced developers manually review the changes that other developers submit via pull requests. Therefore, we might find few critical violations within the repository history. However, these experienced developers might have spent a long time to manually review all code contributions, and they might miss violations.

The number of selected projects is limited due to the time-consuming task of understanding the systems, which is necessary to write relevant constraints. We discuss this threat in Section 5.5.2. Next, we define Salvum constraints for these projects.

5.2.2 Policies and Constraints

To answer **Q1** and **Q2**, we analyze the selected software projects focusing mainly on their technical domain, source code, the sensitive information they hold, and their contributors. We need to understand their technical domain and source code to correctly write policies and constraints. Identifying sensitive information that we should protect also helps to specify the policies and constraints. Additionally, we check the number of commits a contributor submitted because low numbers might indicate that the developer is either inexperienced or not evolved with the project. Thus, we can write constraints to protect sensitive information for these con-tributors.

A developer that has experience on a particular project could play this role since he would have this knowledge to write constraints in Salvum. As we mentioned in Section 4.3.2, we have to manually choose the entry-points for each project. We might introduce bias because only one author validated these entry-points. Thus, we discuss this threat in Section 5.5.

Moreover, we define policies and constraints with the goal to enforce them for the selected projects. For simplicity, we explain only those policies and constraints that we could detect real violations for the selected projects. However, the complete list can be found in our online Appendix [20]. We discuss the violations that we found by enforcing the constraints presented below in Section 5.2.3.

GITBLIT:

Policy P1: *User password, locality, token, access information,*

and permission cannot flow to code contributions

Constraint C1:

```
ChangePasswordPage {password,confirmPassword}, UserModel {password,locality},
        FederationModel {token}, FederationProposal {token},
AccessPermission {permission}, RepositoryModel {accessRestriction,authControl},
    RepositoryUrl {permission}, TeamModel {permissions} noflow Contribution
```

Constraint C1':

```
ChangePasswordPage {password,confirmPassword}, UserModel {password,locality},
        FederationModel {token}, FederationProposal {token},
                        AccessPermission {permission},
RepositoryModel {accessRestriction,authControl}, RepositoryUrl {permission},
                TeamModel {permissions} noflow WrittingOps where
      WritingOps = {Logger.info(),Logger.error(),Logger.warn(),
                    System.out.println(),setCookie()}
```

In **P1** we specify sensitive information handled by Gitblit that we aim to protect. Thus, our goal is to detect whether there is a flow between such information and code contributions. The constraints **C1** and **C1'** written in Salvum help us to achieve this goal. We identify the fields that initially store the sensitive information specified in **P1**, such as `password` and `locality` from `UserModel` class. The `noflow` construct determines a violation occurs in case there is a flow between the sensitive information and code contributions. Notice that we use the `Contribution` identifier, which implicitly defines the contributions to be analyzed. To obtain the results we show in Section 5.2.3 for **C1**, we consider a set of 49 different Gitblit versions. Therefore, the `Contribution` identifier represents 49 code contributions. For **C1'**, we consider a unique Gitblit version. This variation is useful because we can find violations introduced by a code contribution already integrated into the repository without building the corresponding Gitblit version.

SCRIBEJAVA:

Policy P2: *User token secret cannot flow or be altered by code contributions*

Constraint C2:

```
OAuth1Token {tokenSecret}, ServiceBuilder {apiSecret},
        ServiceBuilderAsync{apiSecret},
OAuth1AccessToken {tokenSecret}, OAuth1RequestToken {tokenSecret},
  OAuthConfig {apiSecret}, RSASha1SignatureService{privateKey} noflow
                        Contribution
```

Constraint C2':

```
Contribution noset OAuth1Token {tokenSecret}, ServiceBuilder {apiSecret},
        ServiceBuilderAsync{apiSecret},
OAuth1AccessToken {tokenSecret}, OAuth1RequestToken {tokenSecret},
    OAuthConfig {apiSecret}, RSASha1SignatureService{privateKey}
```

We define **P2** to state that sensitive information cannot flow to code contributions and code contributions cannot alter sensitive information. We write **C2** and **C2'** in Salvum to enforce **P2**. The constraint **C2** is similar to **C1**, which determines that sensitive information cannot flow to code contributions. On the other hand, **C2'** uses `noset` construct to state that code contributions cannot alter sensitive information initially stored in the specified fields.

We also write more policies and constraints for ScribeJava, but we did not find any violations. So, we focus only on the constraints that helped us to detect violations.

CRAWLER4J:

Policy P3: *User proxy password cannot flow to code contributions or writting operations*

Constraint C3:

```
CrawlConfig {proxyPassword} noflow Contribution
```

Constraint C3':

```
CrawlConfig {proxyPassword} noflow WrittingOps where
WrittingOps = {Logger.info(),Logger.error(),Logger.warn(),
                        System.out.println()}
```

The **P3** policy not only specifies that user proxy password cannot flow to code contributions but also to writting operations. Therefore, we write **C3** and **C3'**. The former is similar to **C1** and **C2** whereas the latter states that `proxyPassword` cannot flow t o w riting o perations called throughout Crawler4J source code, such as `Logger.info()`. At last, we explain the violations these constraints helped to find in Section 5 .2.3. Additionally, in our online Appendix [20], we provide a complete list of constraints we have written in Salvum.

5.2.3 Results

In this section, we show the results we obtain running Salvum for the constraints and software projects explained. First, we answer **Q1**.

Answering Q1.

In particular, Table 5.3 illustrates the number of different software versions we analyzed and the number of violations we have found for all considered versions of each selected software project.

For the cases where Salvum detects a violation, we still need to manually review the code contribution. There are two possibilities: (i) we acknowledge that there is indeed a vio-lation, so we open an issue for the corresponding project on GitHub, or (ii) we recognize that there is a sensitive information flow to the code contribution, but it is not h armful. For exam-ple, a developer might submit commits containing a new authentication implementation. In this case, Salvum detects a sensitive information flow, then we check the code contribution, and we might conclude that the developer correctly implemented this functionality.

Table 5.3: Results for violations

Project	Versions	Number of Violation Warnings	Number of Violations
Gitblit	49	2	1
Open Refine	10	0	0
Voldemort	52	0	0
ScribeJava	27	3	3
Solo	10	0	0
HikariCP	10	0	0
Apache Kafka	43	0	0
Teammates	10	0	0
Crawler4J	10	1	1
Total	221	6	5

Thus, we do not report a violation. This scenario is typical when analyzing initial commits. In summary, the procedures to obtain the results for this evaluation is as follows:

- We write constraints for a software project;
- We run our tool to detect these constraints;
- We manually analyze the set of violation warnings;
- We decide whether the warnings are real violation;
- In case there are violations, we report them for the project developers on GitHub.

Gitblit, Voldemort, Teammates, Open Refine, and Apache Kafka present a more significant number of commits than the other projects. Hence, we consider more versions to encompass around three years of development time as well as different contributors. ScribeJava presents less commits, but it has more developers contributing than Voldemort, for instance. Therefore, we consider 27 different versions. Our results could be biased because of the difference in the number of versions that we consider for each project. Thus, we discuss this threat in Section 5.5. The remaining projects either handle less sensitive information or present fewer contributors or commits. Thus, we take into account fewer software versions for them.

These nine projects adopt manual code review for their pull based software development through GitHub pull requests. For each project, there is at least one developer that reviews code contributions. Additionally, we also see that many code contributions fix only user interface presentation, typos, and configuration files. To conclude that, we manually reviewed the code attached to all Gitblit pull requests and at least five for the other projects. These kinds of commits do not add or change lines of Java code, which avoids introducing violations in our context.

Nonetheless, we expect that the problems we might find should be critical because they encompass sensitive information. In Table 5.3, the NVW metric results show that we found six violation warnings for the selected projects of which five were confirmed as violations by experienced developers from each project, as we can see in NV results. One violation warning was discarded by Gitblit developers because it was not a real threat. We opened an issue explaining this warning, but Gitblit developers concluded that it was not a real problem.

In the following, we explain the violations we found for Gitblit, ScribeJava, and Crawler4J.

Gitblit. For this project, we found two violations of the constraints we explained in Section 5.2.2. In particular, we illustrate the output Salvum provides for constraint **C1** below. We use this violation as motivating example in Chapter 3. This output indicates that there is an illegal flow of the information initially stored in `password` to line 284 of the `RootPage` class at the Gitblit version associated with commit hash `efdb2b3d`.

```
Illegal flow from UserModel.password
      to RootPage.loginUser() at line 284 in commit efdb2b3d
```

We reported this violation, and Gitblit developers accepted it as an issue[1]. Afterward, we submitted a pull request to fix this issue. Listing 3.1 illustrates the line that caused this problem. We solve it by implementing a new way of generating a hash for the cookie mechanism. Instead of using password and username, we produce a random number. The pull request was accepted, and its code was integrated into production repository. This issue remained unnoticed for 655 days. Furthermore, Gitblit developers also opened an issue to change the implemented hash function after we warned them about the weakness ot he previous method.[2]

ScribeJava. Among the 27 Scribe Java versions, we found one violation to constraint **C2**. Thus, we manually analyzed the code contribution introduced by commit `d734a4df`. Indeed, a developer added one flow from `tokenSecret` to `toString()` method, which might leak sensitive information in case other developers carelessly call it.

```
Illegal flow from OAuth1Token.tokenSecret
      to OAuth1AccessToken.toString() at line 48 in commit d734a4df
```

We also reported this violation to Scribe Java developers. After one experienced developer reviewed our report, he confirmed the issue and fixed it himself by removing the `toString()` method.[3] Therefore, we did not need to submit a pull request. This issue remained unnoticed for 573 days.

Crawler4J. The violation we found for this project is similar to the one explained for

[1] https://github.com/gitblit/gitblit/pull/1116#pullrequestreview-12289367
[2] https://github.com/gitblit/gitblit/issues/1166
[3] https://github.com/scribejava/scribejava/issues/796#event-1250752673

ScribeJava. There is a violation of the constraint **C3**. We manually analyzed the code contribution that introduced this violation, and we identified that a developer also added one flow from `proxyPassword` to a `toString()` method. Crawler4J developers accepted this violation, and they asked for a pull request to fix it[4]. This issue remained unnoticed for 2091 days.

```
Illegal flow from CrawlConfig.proxyPassword
      to CrawlConfig.toString() at line 399 in commit 1c23e320
```

> Based on `NVW` and `NV` results shown in Table 5.3, we answer **Q1** stating that Salvum can detect proper violations of sensitive information and code contributions in the context of software projects that have a defined code review process.

However, due to the low number of problems for this set of software projects, we aim to answer two secondary research questions: **Q1.1** and **Q1.2**.

Answering Q1.1.

The answer to **Q1.1** might help us unveil the reason we have found only a few violations. Thus, we read developer discussions of at most 25 merged pull request containing at least two messages attached considering the nine projects. We choose 25 because while performing this analysis, we noticed the reasons of discussions started to be repetitive, which indicates that we would have similar results for more pull requests. Additionally, We observed that two messages indicate an initial debate on the code contributions whereas only one message suggests that the pull request developer explains something without any replies. Indeed, most messages discuss change requests and responses. On the other hand, pull requests without messages suggest that developers did not review it or there was no problem and it was merged. The former case is not compelling to answer **Q1.1** because there is no change request. The latter case means that the code contribution is integrated, and consequently, we could analyze it using Salvum. Table 5.4 depicts the number of pull request we manually analyzed for each project.

Following Lientz and Swanson [70] and Gousios, Storey, and Bacchelli [24], we divide the types of contributions in three categories: Perfective (new functionalities), Corrective (fixing issues), and Preventive (refactoring). We aimed at checking a correlation between these categories and security-related issues, but we did not find a connection.

Our goal was to analyze 25 pull requests for each project, but Solo has only six merged pull request with at least two messages whereas Crawler4J has only 10. Thereby, we analyzed a total of 191 pull requests messages.

The main reason for discussions throughout these 191 pull requests in GitHub is to: (i) include or run tests, (ii) obey development guidelines like code indentation, (iii) change

[4] https://github.com/yasserg/crawler4j/issues/260#issuecomment-346919742

Table 5.4: Manually analyzed pull request messages

Project	Perfective	Corrective	Preventive	Total	Security related
Gitblit	11	5	9	25	0
Open Refine	8	2	15	25	0
Voldemort	7	6	12	25	0
ScribeJava	8	9	8	25	2
Solo	0	5	1	6	2
HikariCP	5	10	10	25	1
Apache Kafka	6	8	11	25	1
Teammates	9	9	7	25	0
Crawler4J	5	2	3	10	0
Total	59	56	76	191	6

behavior of new functionality, (iv) fix rebase, branch, or merge issues, (v) explain how the proposed new functionality works and why it is useful, and (vi) fix compilation problem. The nine projects present discussions mostly about these six reasons.

We can observe in Table 5.4 that security-related pull requests are rare for our sample. Three of them are Perfective pull requests, and the other three are Corrective. The discussion attached to these six pull requests are related to (i), (ii), and (vi). However, there are no comments about privacy and security violations of sensitive information. To confirm that they do not concern violations, we analyzed the code of these six pull requests. Indeed, none of them introduces violations.

In this way, these manually analyzed pull requests ratifies our previous findings with Salvum. Only a small amount of code contributions are associated with privacy and security of sensitive information. Thus, regarding our sample, it is not unusual that we find a few violations of our specified constraints for projects that have an organized way of contribution.

> In summary, we did not find violations that developers solved before merging code contributions for our sample. Thus, the `NvC` metric is zero which leads us to answer **Q1.1** stating that we did not find on the analyzed projects that developers solved these violations for our sample.

Answering Q1.2.

To answer **Q1.2**, we manually analyzed the source code of at most five unmerged pull requests that have already been closed. Hence, developers will not merge them in the future. As we explained in Section 3.5, manually reviewing code contributions is a time-consuming task. For this reason, our sample is small. However, we plan to extend this analysis, as we discuss in Section 7.3.

The corresponding results might bring evidence that we missed violations because the code contributions that would introduce them was not integrated into the repository. In this case, we do not need to analyze the discussions associated to pull requests because a developer can close them without demanding changes from the submitter.

We search for violations of the constraints we define in Section 5.2.2. Thus, while manually reviewing, we have in mind mainly where and how the code contribution deals with sensitive information. The procedure to collect data to answer **Q1.2** for each project is as follows:

- We search for the five most commented pull requests that are closed and unmerged;
- We read the last comments to check why developers rejected the pull request;
- We search for occurrences of sensitive information on the resulting `diff` file;
- We decide whether those occurrences configure a violation.

We performed this procedure for 45 pull requests, which leads to 39517 lines added, changed, or removed. However, we skipped non-Java code and test implementations because they cannot introduce violations in our context. For this reason, the total number of Java code lines that we read was reduced. Table 5.5 summarizes our manual analysis results. Despite the fact that we could not find a connection between privacy and security pull requests and the categorization of Lientz et al. [70], we use it to keep the pattern equal to Table 5.4.

Table 5.5: Manually analyzed unmerged pull requests

Project	Perfective	Corrective	Preventive
Gitblit	4	1	0
Open Refine	2	1	2
Voldemort	3	0	2
ScribeJava	3	0	2
Solo	2	2	1
HikariCP	3	2	0
Apache Kafka	3	0	2
Teammates	2	1	2
Crawler4J	3	0	2
Total	25	7	13

As we discussed before, security-related pull requests are rare for our sample. Only two ScribeJava rejected pull requests approach privacy and security properties. They implement changes that print access tokens on the console. However, We did not classify them as violations because these operations are implemented in classes that exist only as examples, that is, real systems that use ScribeJava do no use

them. Therefore, the `Number of unmerged code contributions containing violations (NrC)` is zero for our sample.

> Then, we answer **Q1.2** stating that the unmerged code contributions on GitHub are not related to violations of sensitive information for our sample.

This answer brings evidence that we did not miss violations because the code contribution that could have introduced them was not merged. Regarding the analyzed sample, we found a few violations. Therefore, we do not have evidence that these violations existed on GitHub project, but they were detected and fixed during the code review process. This way, one possibility is that the developers of these projects are experienced, so they introduce a small number of these kinds of violations. Another possibility is that these violations are rarely introduced even for inexperienced developers, or they do exist, but they are fixed at the local repositories.

At last, the main reasons for the rejected pull requests were: (i) submitter did not respond a change request for a long time, (ii) another pull request replaces the rejected one, and (iii) the new functionality implemented in the pull request is invalid.

Answering Q2.

Moreover, to assess whether Salvum reduces the effort (regarding project versions and line of code to review) to review manually these violations that Salvum detects, we consider **Q2**. Table 5.8 illustrates the results for `PA` and `SA` metrics. For instance, we needed to perform a manual code review for only three versions of the Gitblit project. The other 46 versions do not present violations as Salvum established. Therefore, in the worst case, a manual code reviewer would need to consider the 49 selected versions. This scenario could happen for cases that we need to analyze all the contributions submitted by a particular developer, as it was the case for our untrustworthy developer scenario in Section 3.4. Table 5.8 also illustrates the percentage of total `PA`. For example, `PA` with Salvum for Gitblit is three, which means that we need to analyze only 6,12% of the total project versions.

Additionally, there are 6012 lines of code contribution scattered across the 49 versions. These numbers represent lines of Java code added, changed, or removed in considered code contributions. Thus, the total number of lines to initiate a review is 23053. Besides that, a reviewer would need to review lines of code that were not introduced by code contributions, but are related to them. However, we focus on this set of initial lines to start a code review. Since Salvum shows precisely where the violations occur, we reduce the number of lines to read and confirm or discard the issue.

Thus, the `SA` with Salvum for Gitblit is 32. This measure represents the number of initial lines of Java code that we consider to review to determine Salvum indeed detected a

Table 5.6: Results for revisions

Project	PA without Salvum	PA with Salvum	Percentage of total	SA without Salvum	SA with Salvum	Percentage of total
Gitblit	49	3	6,12%	6012	32	0,53%
Open Refine	10	0	0%	578	0	0%
Voldemort	52	0	0%	5040	0	0%
ScribeJava	27	1	3,7%	4496	4	0,08%
Solo	10	0	0%	952	0	0%
HikariCP	10	0	0%	216	0	0%
Apache Kafka	43	0	0%	3402	0	0%
Teammates	10	0	0%	1863	0	0%
Crawler4J	10	1	10%	494	22	4,45%
Total	221	5	-	23053	58	-

violation of the constraints we specified. In summary, we needed to consider 0,53% of code contribution lines for Gitblit, 0,08% for ScribeJava, and 4,45% for Crawler4J.

However, an experienced reviewer could decrease the number of lines to be reviewed. For example, there might be lines of code corresponding to tests or configuration files. In these cases, there is no change for Java code, and consequently, no violation is introduced. On the other hand, our tool has the advantage to show precisely from and to the sensitive information flows. In this context, even an experienced reviewer would have to put lots of effort to find inter-procedural dependences. For instance, the task of identifying violations of sensitive information that flows throughout many methods of different classes.

Thus, we answer **Q2** stating that Salvum reduces the effort to find violations of the specified constraints under JOANA IFC analysis limitations.

5.3 Assessing low active and poorly supported software projects

For the second part of our evaluation, we consider five software projects used in previous study [26, 11]. We selected the projects that are still available either in SourceForge [71] or GitHub. From a total of 36 projects, we discarded the unavailable ones and also those that we could not correctly build or identify sensitive information.

Different from the nine projects in Section 5.2, the current ones do not have an organized code review process. In this way, we expect to find more violations than the previous evaluation. We explain these projects in Section 5.3.1. Additionally, we define a set of constraints for them in Section 5.3.2. At last, we describe the results we obtained in Section 5.3.3. Our goal here is to answer **Q1** considering the latest version of each project.

5.3.1 Selected software projects

In this assessment, we select the following open-source and Java written software projects: Blojsom [50], Personalblog [72], GridSphere [73], SnipSnap [74], and Lutece [75]. In Table 5.7, we illustrate some of their characteristics.

Table 5.7: Selected software projects

Project	Commits	SLOC	Technical Domain
Blojsom	3600	6500	Blog
Personalblog	-	4300	Blog
GridSphere	6000	40400	Web portal
SnipSnap	1500	22800	Wiki
Lutece	3000	69700	Web portal

Blojsom is a multi-user blogging system. It has nearly 6500 lines of Java code and 3600 commits. Developers have supported this project for ten years until 2013. However, the last stable version is still available for download. Blojsom handles sensitive information like owner authorization and email as well as many passwords.

Personalblog is also a blogging application. It has approximately 4300 lines of code. Equally to Blojsom, this project has been supported for ten years until 2013, and its last version is available for download. Sensitive information such as hibernate connection properties, password, cookies, and sessions flows throughout Personalblog source code. We could not retrieve the list of commits for this project because it uses an old control version system, which is not supported for migration to GitHub.

Gridsphere is a web portal developed from 2002 to 2010. During this time, it was highly active containing three mailing lists and issue tracker. Thus, it has around 6000 commits and 40400 lines of code. This system holds passwords and certificates as sensitive information. SnipSnap is a collaborative system that executes a wiki, which allow users to create and manage pages via a browser. This project has roughly 1500 commits and 22800 lines of code. It also handles sensitive information like user password.

The last selected project, Lutece, is a web portal that allows users to create websites or intranets. Developers still support this project. They have submitted 137 pull requests. However, their code contribution review is not as organized as the projects of our first assessment. For example, the pull requests do not have many comments before being accepted. In turn, this project is more active than the other four we selected. Thus, we expect to find fewer violations.

To evaluate these projects, we define policies and constraints for each one in Section 5.3.2, and we explain the results in Section 5.3.3.

5.3.2 Policies and Constraints

To answer **Q1** for a set of selected projects with different characteristics, we write policies and constraints. We explain only policies and constraints that helped us to find violations and also because the others are similar. However, the complete list can be found in our online Appendix [20].

BLOJSOM:

Policy P4: *Blog owner email, authorization, and any password cannot flow or be altered by writing operations*

Constraint C4:

```
Blog {blogOwnerEmail,blogOwner,authorization}, AtomAPIServlet {password},
    BlojsomXMLRPCServlet {password}, BloggerAPIHandler {password},
        MetaWeblogAPIHandler {password} noflow Writeops where
    Writeops = {Logger.info(),Logger.error(),Logger.warn(),
                System.out.println(),setCookie()}
```

Constraint C4':

```
Writeops noset Blog {blogOwnerEmail,blogOwner,authorization},
AtomAPIServlet {password},BlojsomXMLRPCServlet {password},
BloggerAPIHandler {password}, MetaWeblogAPIHandler {password} where
    Writeops = {Logger.info(),Logger.error(),Logger.warn(),
                System.out.println(),setCookie()}
```

P4 defines sensitive information that Blojsom handles and the kind of operations that they should not flow. Thus, our goal is to detect an information flow between these elements. Therefore, we specify two constraints in Salvum to enforce **P4**. The first one, **C4**, aims to identify a flow from sensitive information to writing operations, which would represent a potential leak. Thus, a violation occurs if there is a flow. This constraint guarantees confidentiality. On the other hand, **C4'** detects a flow from writing operations to sensitive information (the opposite), which would mean that the source code violates integrity.

The other policies and constraints for the remaining four selected projects are similar to **P4**, **C4**, and **C4'**. The differences regard the set of sensitive information, which we briefly explained in Section 5.3.1, and also the potential leaking operations. For example, the Personalblog implements a `sendMessage()` method to send an email. Therefore, we define policy and constraints to detect a violation through this method because we do not want it to leak passwords. Thereby, we discuss the results for this evaluation in Section 5.3.3.

5.3.3 Results

In this section, we explain the results we obtain running Salvum for the constraints and software projects introduced in Section 5.3.1. In particular, Table 5.8 illustrates the number of

violation warnings for each selected software project. Differently from Section 5.2, we consider only the last version of each system. Blojsom and Personalblog are not pull-based development. Thus, there is no pull request to analyze. Additionally, GridSphere and SnipSnap were migrated from SVN to Git. Therefore, they do not have GitHub pull requests. In turn, Lutece is currently developed on GitHub, and it has pull requests. However, this project does not have an organized code review process. Thereby, we assess it in this section.

Table 5.8: Results for violations

Project	Number of Violation Warnings	Number of Violations
Blojsom	34	33
Personalblog	44	43
GridSphere	4	2
SnipSnap	1	1
Lutece	1	1
=	86	80

Blojsom. Salvum warned 34 violations for this project of which we confirmed 33. In this case, we looked carefully to verify whether it is a violation because we could not report them since contributors do not develop Blojsom anymore. We did not confirm one reported warning because it was a Java reflection usage, and JOANA does not support it [33]. Thus, it generates a false-positive because JOANA conservatively establishes that there is a flow for every reflection usage [18]. Below we show one of the 34 outputs that our tool provided:

```
Illegal flow from BloggerAPIHandler.password
      to BloggerAPIHandler.editPost() at line 452 in commit 71e96cb
```

Line 452 contains a call to a `Logger.error()` method with `userid` and `password` arguments. Therefore, it represents a violation because developers should not print sensitive information such as passwords in log files. It is a well-known problem documented in the OWASP guides.[5] The majority of problems we found in Blojson is similar or related to leaking sensitive information through writing operations like logging. Thus, we do not discuss them here because it would be tedious and repetitive. Nonetheless, we provide the full report in our online Appendix [20].

Other tools have also found many privacy and security issues for Blojsom [26, 11]. However, this project is not developed for a long time, thus developers did not fix its problems.

Personalblog. This project is the only one we could not migrate to GitHub because its last version was still using CVS, which is not supported for migration by that platform. Hence, Salvum could not show the commit that originated the issues. We found 44 violations of which

[5] https://www.owasp.org/index.php/Logging_Cheat_Sheet

we could confirm 43. The other one was also a false-positive due to JOANA limitation regarding reflection. We show one of Salvum's outputs below:

```
Illegal flow from InitializationManager.hibernate_properties
      to InitializationManager.createHibernateConfigFile() at line 80
```

The `hibernate_properties` field holds the password that grants access to the database. The code of line 80 writes the database password in a configuration file, which is also a well-known problem explained in the OWASP guides.[6] Hence, this violation represents a leak of sensitive information that we should avoid.

GridSphere. We found four violations for this project, but two of them are false-positives, both due to JOANA limitation concerning reflection. The following violation represents a different sensitive information leak.

```
Illegal flow from UserManagerPortlet.passwordBean
      to UserManPortlet.mailUserConfirmation() at line 653 in commit 81219e5a
```

Line 653 contains a call to the `sendMail()` method, which sends email messages with the goal of user confirmation. The problem is that they include the `passwordBean` in their body, and this password is not temporary. In case an attacker has control over the user browser or email account, he can also access the GridSphere account. Thus, we should avoid sending permanent passwords through email introduced in commit `81219e5a`.

SnipSnap. This system also implements a cookie mechanism similarly to Gitblit. We found one violation regarding a user password flowing to this mechanism.

```
Illegal flow from User.passwd
      to DefaultSessionService.getUser() at line 153 in commit 38d6902
```

Line 153 calls the `setCookie()` method, which receives a `user` argument. It is an instance of the `User` class, and it contains sensitive information like a password. Analogously to the Gitblit example in Section 3.1, we should not save passwords in cookies. Therefore, it represents a real violation in the SnipSnap system introduced in commit `38d6902`

Lutece. This project also handles user passwords. As shown in the Salvum output below, we detect one violation regarding a password.

```
Illegal flow from PasswordFactory.strStoredPassword
      to PasswordFactory.getPassword() at line 82 in commit 0b8670f
```

Line 82 throws an exception passing `strStoredPassword` as argument. Therefore,

[6] https://www.owasp.org/index.php/Use_of_hard-coded_password

if the system throws this exception, it could print the password on the stack trace. Thus, an attacker that has access to the stack trace could see passwords from different users, and consequently, access their web portals.

> In summary, we answer **Q1** stating that Salvum can detect a higher number of proper violations of sensitive information and code contributions in the context of poorly-supported software projects.

Our results here are different from the study in Section 5.2 because we consider poorly-supported software projects. Thus, since there is no code review process, it is natural that many violations are introduced by code contributions. Additionally, only Lutece is still in development. Thus, the issues that our work and other studies found are not fixed.

5.4 JOANA evaluation

To assess how well JOANA [13] perform its analysis in our work, we evaluated it against the Stanford SecuriBench Micro [19]. This benchmark is a set of many Java-written programs intended to be used as a testing ground for security tools. Other authors have used it [10, 26, 11, 15, 76] to test their approaches. These programs are organized following the data structure or operations where the problems might occur. We obtain the results to answer **Q3** by running JOANA for 73 SecuriBench tests to check its precision and recall.

Our primary goal is to determine the cases that JOANA misses information flows, which could impact in our results for **Q1** and **Q2**. Thus, we generate a System Dependence Graph (SDG) for each test case using JOANA's API, and we manually annotate the necessary instructions with *High* and *Low* for each program using JOANA's user interface. In Table 5.9, we illustrate the results for this assessment.

Table 5.9: SecuriBench Micro test results

Test Case Group	TP	TP + FN	FP	FN	TN	Recall	Precision	Accuracy
Aliasing	12	12	0	0	2	100%	100%	100%
Arrays	9	9	5	0	1	100%	64%	62%
Collections	14	14	6	0	1	100%	70%	71%
Datastructure	5	5	3	0	0	100%	62%	62%
Factories	3	3	3	0	0	100%	50%	50%
Inter	16	16	8	0	3	100%	66%	70%
Pred	5	5	3	0	1	100%	62%	66%
Reflection	3	4	6	1	0	75%	33%	30%
Session	3	3	1	0	0	100%	75%	75%
Strong Updates	1	1	4	0	0	100%	20%	20%

JOANA was able to detect all the violations of 9 test case groups out of 10. The `Number of true-positives (TP)` and the number of existing violations, which is represented by the sum of `TP` and `FN`, are different only for the Reflection group. As expected, JOANA missed one violation regarding reflection since it does not support it. Therefore, the only recall below 100% is for the reflection group. Since JOANA conservatively indicates that there is information flow for reflection code, we believe this false-negative happened due to a bug. We calculate the recall based on the metrics of `TP` and `FN`. We use the following formula:

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

Surprisingly, JOANA results for this benchmark shows that for most examples the `Number of false-positives (FP)` is not zero, which indicates high rates of false-positives. Thus, we use the following formula to calculate precision:

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

As shown in Table 5.9, only the Aliasing group has 100% of precision. Since we enable object-sensitivity in JOANA configuration, we could correctly detect aliasing examples like the one illustrated in Listing 5.1. Lines 7 and 8 leak `"secret"` to `println()`, but line 9 does not leak it because of the assignment in line 4.

Code 5.1: Aliasing example

```
1  String name = "secret";
2  Object o1 = name;
3  Object o2 = name.concat("abc");
4  Object o3 = "anc";
5
6  PrintWriter writer = resp.getWriter();
7  writer.println(o1);        /* BAD */
8  writer.println(o2);        /* BAD */
9  writer.println(o3);        /* OK */
```

However, JOANA presented higher rates of `FP` on the other groups, which compromised its precision. For example, the Strong Updates group result has 20% of precision with four false-positives. In Listing 5.2, line 12 does not leak `"secret"` because line 10 assigns `"abc"` to `w.value`. However, JOANA detects a violation in line 12. Conceptually, JOANA should not introduce such false-positive. Thereby, we opened an issue on GitHub to report this potential problem.

Code 5.2: Strong Update example

```
1  class Outter{
2   class Widget {
3    String value = null;
4   }
5
6   void m() {
7    String name = "secret";
8    Widget w = new Widget();
9    w.value = name;
10   w.value = "abc";
11   PrintWriter writer = resp.getWriter();
12   writer.println(w.value);    /* OK */
13  }
14 }
```

Additionally, we calculate the accuracy for our sample. We obtain the accuracy with the following formula:

$$accuracy = \frac{TP+TN}{TP+FP+FN+TN}$$

Table 5.9 shows that the Aliasing group has 100% of accuracy. Thus, JOANA performs better for these kinds of scenarios regarding recall, precision, and accuracy. Due to the high number of false-positives, the Strong Updates group has only 20% of accuracy, which makes him even worse than Reflection. However, the Reflection group does not have true-negative examples in this benchmark. Thus, the `TN` metric measures zero. Since JOANA does not support reflection, we would have lower accuracy rates in case the Reflection group had true-negative examples.

The SecuriBench Micro examples exercise a large variety of IFC scenarios, which in many times are not common in real systems. Even though the higher recall, precision, and accuracy, the better to detect these examples, we can still find real problems considering 100% recall. The results of Table 5.9 shows that JOANA presents a false-negative for Reflection, as expected. For the other cases `FN` is zero, which allow us to conclude that if there is a violation, JOANA is capable of finding it.

However, these results also show that precision and accuracy rates are lower than we expected, which might reduce productivity to find violations in real systems. For example, if we have cases similar to the programs of the Strong Updates group, we might have more effort to understand, and consequently decide whether JOANA detected a real violation. Therefore,

this limitation is a drawback also for our work because Salvum might find high rates of false-positives for other examples.

Other IFC analysis implementations present lower rates for false-positives [10, 15]. The Flowdroid tool [10] presents a total of only nine false-positives. However, its developers only evaluate it for eight test case group (we consider ten). In turn, the PIDGIN tool [15] consider 12 test case group and present only 15 false-positives. Unfortunately PIDGIN is not open-source. Nonetheless, due to this experiment results, we plan to use another IFC analysis implementation in our tool to check whether we enhance precision and accuracy. We opted not to use the Flowdroid tool because we would need to implement several changes so that we could run its analysis for software projects of technical domains different from Android applications. Since PIDGIN is not open-source, we also could not use it in this work.

In real scenarios that we show in Sections 5.2 and 5.3, we only found false-positives regarding reflection. Thus, the low rates of precision and accuracy did not bias our previous results. However, if we used another IFC analysis implementation, we would be able to decrease even more the number of lines of code that we need to review.

Besides that, as we can see in Section 5.2.3, we could significantly reduce the number of lines of code and project versions to be reviewed.

We answer **Q3** stating that JOANA has a good performance regarding recall. Although, we expected that it would have higher rates for precision and accuracy.

5.5 Threats to validity

In this section, we discuss the threats to validity of our evaluation. By following Wohlin et al, we organize the threats as Construct validity (Section 5.5.1),Conclusion validity (Sec-tion 5.5.2), External validity (Section 5.5.3), and Internal validity (Section 5.5.4).

5.5.1 Construct validity

Threats to construct validity cover issues related to the design of the assessment and its capacity to answer the research questions [77].

The selected metrics limitation. In Section 5.2, we evaluate effort using approxima-tions: lines of code and project versions to analyze. However, we do not consider other metrics such as the time to learn and define the policies and constraints as well as the time to learn how to perform a manual code review.

To mitigate this threat, we could execute a controlled experiment in which we assign tasks to reviewers. These tasks would be related to finding v iolations. T herefore, w e could

measure the difference of time to execute these tasks with and without Salvum. We might bring more insights about how our tool improves productivity.

Another threat related to the metrics we use in this work is related to the fact that they are approximations. For example, an experienced developer could need to review a lower number of lines of code to find the same violations comparing to an inexperienced developer. This way, the aforementioned controlled experiment could also mitigate this threat if we select developers with different levels of experience.

Also because the metrics are approximations, we might have introduced a bias in the number of lines of code that we need to review to find the violations. Only one author of this work manually reviewed the code contributions that Salvum identified violation warnings. Although we have academic experience in code review, it would be interesting to have many reviewers to decide the total lines of code that we need to consider for identifying the violation.

5.5.2 Conclusion validity

Threats to conclusion validity are concerned with issues that affect the ability to draw the correct conclusion [77].

Sample size limitation. To answer the research questions **Q1**, **Q1.1**, **Q1.2**, and **Q2**, we used small samples for selected projects and reviewed pull requests. This limitation might introduce a bias in our conclusions.

The small sample of software projects is due to the time-consuming task of studying the systems to be able to write relevant constraints. We tried to contact developers of some of these projects, but we did not receive replies. The only way to receive feedback from developers was to open an issue regarding the violations we found and tag the developer that introduced it. Thus, we could not select a much higher number of projects.

The small sample of pull request is also because of the time-consuming task of manually understand the discussions and source code. Additionally, only one author was responsible for both tasks.

5.5.3 External validity

Threats to the external validity concern a generalization of the results [77].

The selected software projects, specified constraints, and analyzed pull requests. Due to the analysis limitations explained in Section 5.5.2, our set of selected software projects, specified constraints and analyzed pull request is small. Thus, we acknowledge that we might have a small sample to answer **Q1** and **Q2**. Additionally, the projects are only Java-written and open-source. Therefore, we cannot guarantee that our results apply to other systems written using different programming languages or proprietary software. However, we still could find real violations, and we could reduce the effort to review code contributions manually.

The automated static IFC analysis. In this work, we implement our language using static IFC analysis. Therefore, we cannot generalize our results to other types of analysis, such as dynamic [78, 32, 79] or Bayesian-based [80]. However, implementing policy languages using these alternative techniques could open an interesting avenue of research. We should consider these alternatives as future work.

5.5.4 Internal validity

Threats to the internal validity concern the fact that the assessment affects the re-sults [77].

The procedure of manually reviewing pull requests. To answer **Q1.1** and **Q1.2**, we manually analyzed many pull requests code and associated comments. However, we might have introduced bias because we also defined the policies and constraints for the code contributions. Thus, we might have missed other violations that were not related to our constraints. Despite the fact that **Q1.1** and **Q1.2** are secondary research questions in this work, our results comply with the results we obtained using Salvum to answer **Q1**. Anyway, it would be interesting to have more reviewers to perform the procedure of manually reviewing the set of pull requests we selected.

The procedure to select software projects. For the projects we consider in Section 5.2, we developed a tool [20] to find repositories in GitHub that contain a set of characteristics: Java as the primary language, at least 100 GitHub stars, and 50 forks. We intend to select highly active projects to show evidence that Salvum can find violations in this c ontext. Although, we cannot assure that these values are ideal to obtain highly active projects. For this reason, we might introduce bias in this procedure.

Moreover, we selected the projects that we consider in Section 5.3 with the intent to show evidence that Salvum finds a higher number of violations for low active p rojects. Hence, we searched for them by reading the evaluation of related work (Chapter 6). Since we knew that other studies had found privacy and security problems on this set of projects, we could be biased to write the policies and constraints. In turn, these related work are designed to find different types of issues, like SQL Injection [11].

Time to execute analysis. JOANA analysis execution might be slow for large software projects, such as Voldemort. Depending on the number of lines of a code contribution, it takes up to a day to finish e xecution. This delay happens even using our powerful hardware (Intel Xeon E7 with 64 gigabytes of RAM). It occurs mainly because the System Dependence Graph (SDG) [36] gets very large for such cases. Thus, this limitation on execution time affects our results because we might have missed violations for the cases that we could not execute JOANA.

The set of pull requests we manually reviewed. We establish parameters to select the code contributions to review in Section 5.2.3. For example, pull requests with at least two messages attached to answer **Q1.1**. We intended to select pull requests that developers discussed

merging so that they could have solved violations. However, we did not review pull requests with other characteristics for this case. Hence, we cannot generalize that developers do not address privacy and security violations before merging pull requests.

Size of source code. Static analysis using SGDs do not support projects larger than 100KLOC [81]. So, it is unfeasible to run JOANA analyses providing an entry point that the resulting SDG would encompass more than 100KLOC. Indeed, we noticed that "Out of memory errors" happen for such cases. However, we also run the IFC analysis for larger projects (e.g., Voldemort) because the code contributions do not encompass the whole source code. Therefore, this limitation about choosing the entry points affects our ability to conclude whether a massive project presents violations. To mitigate this issue, we generate different SDGs for the same project by providing different entry points. In turn, we chose these entry points with the goal to encompass the code contributions that we aim to analyze.

IFC analysis limitation. Currently, Salvum uses JOANA as an IFC approach. Albeit we conclude it has a high recall, JOANA does not support reflection so that we might miss other kinds of violations. Therefore, this limitation might affect the results we discuss in Sections 5.2 and 5.3. However, we developed Salvum to be as independent as possible from the IFC implementation. Therefore, we can change to other IFC tool in case we observe many violations regarding reflection for future work.

Specified policies and constraints. In this work, we studied the selected projects to understand their purpose, the sensitive information they handle, architecture, and how developers contribute. Thus, we know to define some policies and constraints for these projects. However, we cannot guarantee that all the sensitive information is specified in our constraints. To be able to define other relevant constraints, we tried to contact developers of the selected projects, but we got no response. This limitation might lead to losing some violations, which affects our conclusions. When we manually reviewed the pull requests to answer **Q1.1** and **Q1.2**, we also tried to observe additional sensitive information that we missed. In this way, we could define more policies and constraints to mitigate this drawback. Nonetheless, developers with vast experience in a particular software project could reduce this issue by establishing their Salvum constraints.

Number of project versions. In Section 5.2.3, we obtain our results from a different number of software projects for each project. Therefore, we could have found more violations for some projects due to a higher number of versions. For example, we consider 49 Gitblit versions and only 10 Open Refine. We found one violation regarding Gitblit and none regarding Open Refine. This way, we could have found more violations if we considered more Open Refine versions.

Decisions whether there is a violation. In Section 5.3, only one author of this work manually decides whether a violation warning is a real problem. Ideally, at least two researchers should make these decisions. This way, we could reduce bias because there would be a discussion before determining a violation occurrence. In contrast, this issue does not happen in

Section 5.2 because we report the violations on GitHub, directly to the project developers. Therefore, these developers decide whether we found a real violation.

Entry-points. For the results we obtained in this work, we manually chose the entry-methods necessary to generate an SDG. We might have introduced a bias in case the author responsible for this task misses methods that contain code contribution changes. This limitation could lead to missing violations for our results. To mitigate this threat, we implemented a mechanism to allow us to provide multiple entry methods to generate one SDG. Therefore, we tried to choose a set of methods that encompasses all code contribution changes. Additionally, in case developers adopt Salvum to their projects, they can define a simple tool to identify all the methods that contain code contribution changes, which results in the set of entry-methods.

6 Related Work

In this chapter, we present related work. First, we discuss manual code review in Section 6.1 and some existing Information Flow Control analysis tools in Section 6.2. Lastly, we approach related language-based information flow in Section 6.3.

6.1 Manual code review

There are some documented guides to improve the execution of manual code review. These guides provide information on how to find specific kinds of vulnerabilities. Thus, a reviewer can focus on user inputs and access to databases, for example. The Open Web Application Security Project (OWASP) [22] provides a number of these guides. Thus a reviewer can follow it to find privacy and security violations for Java projects, .NET projects, etc. They also provide a list of standard vulnerabilities detailing how they might appear in source code and how to fix them. Another widely used guide is the Common Weakness Enumeration (CWE) [44]. It provides a set of software weaknesses to better understanding and management of software weaknesses related to architecture and design. Thus, reviewers can consult this guide to under-stand and discuss privacy and security violations, which could also improve their manual code review tasks.

Nonetheless, as we explain throughout this work, manual code review is costly and time-consuming. These guides help to reduce these factors, but automated tools could improve productivity even more. Thus, our solution does not replace manual code review tasks, but we decrease its effort.

6.2 Information Flow Control analysis tools

TAJ [11] is a static analysis tool designed to detect four of the well-known security vulnerabilities: Cross-site scripting, Injection flaws, Malicious file executions, and Information leakage and improper error-handling attacks. Each of these vulnerabilities can be cast as a problem in which information associated with a label *High* flows to another program part associated

with label *Low* without being endorsed, which means that the information is not validated or corrected. This tool works for Java Web projects and is also a WALA [30] client.

In this context, TAJ defines implicit constraints, which are transparent to its users, to detect occurrences of these four vulnerabilities in Java Web projects. However, this tool would need several modifications to allow developers to specify constraints as we show in Chapter 4, and to for other technical domains than Java Web projects.

Another related static analysis tool is the Flowdroid [10], a static analysis system specifically tailored to the Android platform. It is designed to analyze Android applications bytecode and configuration files to find potential privacy leak. To label program parts as *High* and *Low*, it uses an auxiliary tool named Susi [58], which sweeps Android API to find where potentially sensitive information is stored and potentially dangerous methods, such as `sendSMS()`. Therefore, Flowdroid automatically defines implicit constraints and run its analysis to find violations.

Likewise TAJ, Flowdroid is designed to work for a specific platform. Also, we would need to make several adaptations to use Flowdroid for code contributions. Other tools also present similar drawbacks for our context.

Scandroid [12] is a tool to check Android applications. It extracts security specifications from manifests that accompany such applications, and checks, whether information flows through those applications are consistent with those specifications. Thus, Scandroid automatically defines constraints based on an auxiliary specification, such as manifest files. Mainly, this tool is used to guarantee that an Android application obeys user permissions.

Different from TAJ and Flowdroid, Scandroid constraints might vary depending on these auxiliary specifications. However, we cannot specify our constraints to detect the problems we introduced in Chapter 3. To use Scandroid analysis, we would need to drastically change its source code. Thus, like Flowdroid, this tool is useful for auditing Android applications, but they do not work for other technical domains.

Taintdroid [32] is another related tool. It is a dynamic IFC analysis system capable of simultaneously tracking multiple *Source*s of sensitive information for the Android environment with the cost of performance overhead. Different from the other presented tools, Taintdroid adopts a dynamic analysis. However, it also defines implicit constraints to find illegal information flow when applications are executing. As its main drawback, Taintdroid only supports explicit flows. Thus, its analysis would not be able to identify the problem introduced in Listing 3.1. Furthermore, Tripp et al. propose a Taintdroid new approach to address privacy enforcement as a learning problem, relaxing binary judgments into a quantitative or probabilistic mode of reasoning [80].

Salvum does not provide dynamic analysis to check whether constraint violations occur at runtime. This limitation could be an exciting study for future work. However, it is out of the scope of this thesis because we would have to ultimately implement a new dynamic IFC analysis so that it could work for our scenarios.

Moreover, ANDROMEDA [26] statically detects flows wherein information returned by *Source* reaches a *Sink* without being adequately endorsed by a "downgrader", that is, the information has not been validated. ANDROMEDA is capable of detecting typical information flow related vulnerabilities [22], such as Cross-site scripting and SQL Injection, for a web application. To work for different web frameworks and libraries, ANDROMEDA supports the definitions of extensions so that it works for Struts, Spring, JSF, etc. The purpose and use of this tool are similar to JOANA.

However, JOANA allows more flexibility regarding precision and performance configuration. Furthermore, ANDROMEDA is not as precise as JOANA is [5] since the latter guarantees noninterference [18, 39]. Anyway, Salvum is not high attached to JOANA's implementation. So, we can change our IFC analysis to ANDROMEDA in the case we need it.

The F4F (Framework For Frameworks) tool [82] is also a related work. It is designed to execute IFC analysis of framework-based web applications using a specification language called WAFL for describing the behavior of such frameworks. Thus, to identify privacy and security violations for the scenario, we present in Chapter 3, we would have to add a WAFL generator for each framework the selected software projects (Section 5.2.1) use. For example, Gitblit uses the Apache Wicket [83] framework and Open Refine uses Java Servlet API. On the other hand, Salvum requires only the specification of simple constraints, instead of writing a whole specification language for each selected system.

6.3 Language-based Information Flow

There are languages designed to support IFC. Jif [14, 34] is a security-typed programming language that extends Java with support for information flow control. Thus, developers should modify the system source code to add security types. Therefore, Jif implementation can analyze type errors. For instance, if one variable with security type *High* is assigned to another variable with security type *Low* a compilation error occurs. In this context, it would be hard to attribute security types for different program versions, as we automatically do (Chapter 4). Additionally, Jif tangles constraint code with program code, which might hinder code maintenance and understanding since according to the principle of Separation of Concerns [59] one should be able to implement and reason about each concern independently. The Checker Framework [25] extends Java's type system to let software developers detect and prevent errors in their Java programs. It provides some checkers for different pur-poses, like Nullness, Format String, and Constant value checkers. One of them is related to our work: Tainting checker. It defines a type system that uses mainly two types: `@Untainted` and `@Tainted`. The `@Untainted` annotation is equivalent to JOANA's *Low* label, whereas `@Tainted` is equivalent to *High*. Thus, if we try to assign an `@Tainted` variable to a `@Untainted` one, a compilation error occurs.

Although fast, this tool would demand a significant effort from developers to manually

apply Checker Framework types in systems' source code. This task would be even more difficult when we need to refer to code contributions because we would use these types to different versions of the same system. Besides that, constraint code gets tangled with and spread across system core code, which might harm program maintenance [59].

Joe-E [52] is based upon the Java programming language. It adds some restrictions (taming Java API) to define a subset of Java that provides privacy and security properties of an object-capability language, which says that program state in objects cannot be read or written without a reference. This mechanism is used to guarantee the Principle of Least Privilege [41]. However, to use Joe-E to solve the problems presented in Chapter 3, we would have to make several changes in the source code so that it could compile. Notice that some wide used classes like `java.io.File` have restrictions in this language, such as the exclusion of some methods. Thus, it would be unfeasible to change the source code of several projects, primarily when we work with code contributions since it deals with different source code versions. Caja [84] is also a similar object-capability language, but it is designed to work with JavaScript.

Yang et al. developed a functional constraint language, named Jeeves, to protect sensitive information [16]. It is a Scala programming language extension for privacy policies. Its goal is to enforce these policies to filter the output of any function in the system based on context. For example, if Alice is friends with Bob, she can see she can see his exact location (e.g., Informatics Center, UFPE, Recife). On the other hand, people that are not friends with Bob can only see part of this location information (e.g., Recife). Moreover, Jeeves emphasizes the separation of policies and system core code.

Salvum does not support filtering the output, which, indeed, is useful for some situations. Instead, Salvum currently allows negative and positive constructs. It means that the information cannot flow at all, or that only specific information can flow, which does not specify how fine-grained this information is. However, Jeeves works only at runtime. Thus, to prevent the problems we present in Chapter 3, Jeeves would have to support references for code contributions similar to Salvum.

Johnson et al. provide a generic query language for program dependence graphs to find different kinds of vulnerabilities [15] called PIDGIN. They check if the query resulting program dependence subgraph is not empty for a particular constraint, which configures a violation. We also allow developers to write policies to detect illegal flows from sensitive information to specific operations like logging, as we showed in constraint **C1'** in Section 5.2.2. However, we found violations of policies considering code contributions. PIDGIN does not support these kinds of policies. Indeed, the authors did not find any violation of their case studies [15].

Finally, Apel et al. [85] investigate some shortcomings of object-oriented modifiers, such as `public` or `private`, that limits expressiveness of feature-oriented languages [86]. Thus, they propose three new modifiers: `feature` to limit access to the feature in which the variable is defined, `subsequent` to limit the access to the feature containing the variable and all features composed subsequently [87], and `public` to make the variable globally available.

However, this solution does not support Salvum's flexibility to define different kinds of policies.

7 Conclusions

This work introduces a policy language named Salvum to protect sensitive information confidentiality and integrity from code contributions. Our language allows the specification of constraints that should be enforced for systems of different technical domains. Developers can use Salvum to enforce these constraints either before or after integrating the code contributions. By checking before, we can prevent violations from being introduced in the system code. This scenario is adequate for cases that we must analyze the existing project history. For example, it could be necessary to enforce Salvum constraints to detect violations of sensitive information confidentiality and integrity for code contributions that an untrustworthy developer submits. On the other hand, by checking after, we can identify violations that were introduced by code contributions, that is, they are already merged into the repository. For instance, it could be necessary to enforce Salvum constraints to detect violations of sensitive information confidentiality and integrity for code contributions submitted by a fired developer.

Our language has four main constructs: `noflow`, `noset`, `flow`, and `set`. They specify how code contributions can access sensitive information. For example, we can determine that a `passord` cannot flow to a code contribution using the `noflow` construct. Salvum also supports explicit, implicit, and operations listing declarations of code contributions. For instance, we can specify that the code contributions we want to analyze are represented by all the commits submitted by a particular developer.

Salvum implementation includes four logical steps: *Preprocessing, Labeling, Analyzing,* and *Processing results*. In the *Preprocessing* step we identify the contributions in the source code and build a mapping that we use in the second step. In turn, *Labeling* automatically annotate sensitive information and code contributions as *High* or *Low* security levels depending on the specified constraint. Then, we run the JOANA IFC analysis to identify violations. Finally, we process the results to determine whether it is a real violation in the *Processing results* step. The current version of our tool supports the Java language, although we might use its concepts as a basis for its implementation to other languages.

We evaluate our approach in three different ways. First, we specify Salvum constraints to detect violations of sensitive information in nine highly active and well-supported selected software project. Our results show that even to these projects, which present an organized code

review process, we still can find code contribution violations of sensitive information. Second, we perform a similar assessment with five project that are not highly active nor well-supported. In this way, our results indicate that we can find a higher number of violations for these kinds of projects. We also evaluate the JOANA tool regarding precision, recall, and accuracy of its analyses. JOANA presented a lower precision and accuracy than we expected, but it has a high recall.

7.1 Review of main contributions

This work makes the following contributions:

- A new concept of policy languages;

  To our knowledge, no policy language in the literature deals with code contributions. Therefore, we propose a new concept to protect sensitive information. It might also allow researchers to define other policy languages in this context.

- A policy language named Salvum;

  In this work, we propose Salvum to allow the specification of privacy and security constraints for collaborative software development. Its goal is to protect sensitive information from potentially harmful code contributions.

- A prototype tool based on IFC analysis;

  To enforce the specified constraints, we developed a tool. Therefore, developers can write constraints and check code contribution adherence, which turns it possible to find violations of such constraints.

- Auxiliary tools to perform the *Preprocess* and *Processing results* steps;

  For these steps, we developed simple tools that automatize some work. For the *Preprocess* step, we defined a tool to find projects on GitHub with the characteristics we need. For the *Processing results* step, we developed a tool to filter the IFC analysis results. For example, we remove duplicated illegal flow warnings. This duplication happens because JOANA could detect two illegal flows for the same source code line, but to different instructions.

- Policies and constraints specification for 14 software projects;

  To evaluate our work, we defined a set of policies and constraints for the 14 selected software projects we explained in Chapter 5.

- An empirical assessment considering highly active and well-supported selected software projects;

We use our tool to enforce a set of specified constraints to find violations of sensitive information introduced by code contributions on these projects. To organize our assessment, we define four research questions and use six metrics to answer them.

- An empirical assessment regarding low active selected software projects;

  We use our tool to enforce a set of specified constraints to find violations of sensitive information introduced by code contributions on these projects. We define two research questions and use four metrics to answer them.

- Empirical evidence that our tool can find violations regarding real projects;

  Since we find violations on both empirical assessments, we conclude that our language and tool can indeed be useful in collaborative software development.

- An empirical study concerning JOANA precision, recall, and accuracy.

  By running JOANA against the SecuriBench Micro, we could conclude that it presents high recall, but low precision and accuracy for the test cases.

7.2 Limitations

Our work has many limitations:

- Salvum supports a small set of representations of code contributions. This way, there might be relevant violations that we cannot identify with the current version of Salvum. For example, we could support a list of Java String methods in addition to `contains` (Section 4.2.3). Thus, we could express more policies;

- We use only static Information Flow Control analysis to identify violations. However, we can use other techniques like Dynamic IFC or mutation-based analysis [88];

- IFC analysis demands bytecode as input. Therefore, we have to manually build project versions we aim at analyzing to enforce the specified constraints. For this reason, it is very time-consuming to analyze a higher number of system versions;

- We have not considered cases where the constraints could be conflicting. For instance, we can write that `A noflow B` and `A flow B`. There is no mechanism to alert a conflict. For this case, the analysis result could state that there is a violation of the first constraint independently of the second constraint. Therefore, the current version of our tool enforces each constraint separately;

- We do not have a tool to automatically decide the most suited set of entry-points for a particular system version. This feature would be useful so that we could assure that

we build SDGs that encompass all the code related to the target code contribution. Therefore, we have to manually decide the set of entry-points;

- We use Git commands in our language implementation, which makes the current version of Salvum dependent on this software. Hence, the use of Salvum is limited to this version control system. Nonetheless, Git has become one of the most popular version control solutions. Besides that, GitHub hosts more than 78 million projects supported by 28 million people, and it uses Git;[1]

- The usual high cost of generating SDGs might hinder the use of our *forward* approach. However, we could circumvent this issue by providing entry points that encompass only the code changes and sensitive information. This solution would reduce the size of the SDG, which turns its generation less costly. An alternative solution is to change the IFCS analysis for others that are faster.

7.3 Future work

We intend to extend this study with the following future work:

- In this thesis, we only enforce constraints for code contributions. However, we could also specify policies and constraints for features [89]. We believe that this extension could identify harmful feature interactions or configurations. For example, we could check, based on IFC, whether one feature depends on information that flows within another feature code;

- We intend to conduct more experiments regarding the practical use of Salvum. For instance, one idea is to empirically study developers using our language during the evolution of a real project. This would provide us a better understanding of advantages and disadvantages of our language;

- Customizing SDGs to optimize their creation is also a future work. Instead of creating a whole new SDG for each system version, we could change only nodes and edges corresponding to the differences introduced by a code contribution. This feature would probably decrease the time to analyze multiple system versions. Indeed, we did not find in the literature related work that investigates this optimization;

- Recently, JOANA developers added support for Android library. However, it only works for small toy applications since they still need to implement more optimization for the SDG creation. We tried to consider Android applications for this work, but we could not generate an SDG even for a small Google Play application.[2]

[1] https://github.com/about
[2] https://preview.tinyurl.com/yauwyvsj

Nonetheless, when this support becomes more robust, we plan to use Salvum to investigate Android applications;

- We plan to add a declassification feature for our language. This feature allows the security level (e.g., *High* or *Low*) of sensitive information to be lowered as means to relax the specified constraint. For example, we can specify a constraint that determines user password cannot flow operations that save it in a database unless it is encrypted before. With this feature, we can evaluate our language by comparing it against tools that seek for different kinds of problems such as SQL Injection. Additionally, we could use the OWASP benchmark named WebGoat [90] as an interesting sample;

- We plan to extend our evaluation by including at least another researcher to review the set of pull requests that we manually analyzed. This extension could reduce bias in our results;

- Another point to explore is to consider an alternative Information Flow Control analysis. Since the one we adopt presents low precision and accuracy, we could use another analysis for our tool implementation to mitigate this issue. We believe that we could reduce false-positives for additional results. One possibility is to consider incremental analysis [91] to increase precision. This approach allows developers to interactively decide whether warnings are related to real violations.

- An interesting path of future work is to extend Salvum specification to support more general constraints. Our idea is to allow that developers write constraints and use it among different projects. For instance, we could write this constraint for MVC (Model-View-Controller) projects: `Model.Userpassword noflow View.Page.textArea`. This constraint would help to detect whether there is a flow from a password defined in `User` class on the `Model` layer to a text area defined in the `Page` class on the `View` layer;

- Inspired by AspectJ wildcards [92], we plan to extend Salvum to support this feature. Therefore, developers could write constraints like this one: `Authentication {*} noflow Log where Log = {Logger.*(..)}`. This constraint specifies that any information initially stored in any `Authentication` class variables cannot flow to any method defined in `Logger` class;

- We plan to extend our language specification to allow developers to declare variables that could hold definitions used in more than one location. This improvement would reduce constraint code duplication. For example, we could as-

sign `WriteOps = {Logger.info(), Logger.error()}` to the `logger` variable and reuse it across many constraints;

- Another interesting language extension would be to support the specification of interfaces or superclasses for the operations listing constructs. Therefore, we could allow a reduction in the number of methods that we need to declare. For instance, if `Logger` is an interface in `WriteOps = {Logger.info(), Logger.error()}`, our tool would automatically identify the classes that implement `Logger` so that we could also detect violations for them without explicitly declaring their methods.

References

[1] GitHub: Social Coding. [Online]. Available: http://github.com/

[2] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, 1977.

[3] S. Saghafi, K. Fisler, and S. Krishnamurthi, "Features and object capabilities," in *AOSD*, 2012.

[4] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013.

[5] G. Snelting, "Understanding probabilistic software leaks," *Science of Computer Programming*, vol. 97, Part 1, pp. 122–126, 2015.

[6] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *USENIX Security*, 2005.

[7] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, no. 6, pp. 76–79, 2004.

[8] G. McGraw, "Automated code review tools for security," *Computer*, vol. 41, pp. 108–111, 2008.

[9] Sufatrio, D. J. J. Tan, T.-W. Chua, and V. L. L. Thing, "Securing Android: A survey, taxonomy, and challenges," *ACM Computing Surveys*, vol. 47, pp. 58:1–58:45, 2015.

[10] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Programming Language Design and Implementation*, 2014, pp. 259–269.

[11] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TaJ: effective taint analysis of web applications," *ACM Sigplan Notices*, vol. 44, pp. 87–97, 2009.

[12] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," in *IEEE Symposium on Security and Privacy*, 2010.

[13] JOANA - Java Object-sensitive ANAlysis. [Online]. Available: http://joana.ipd.kit.edu

[14] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. [Online]. Available: http://www.cs.cornell.edu/jif

[15] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," in *Conference on Programming Language Design and Implementation*, 2015, pp. 291–302.

[16] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," in *Proceedings of the Symposium on Principles of Programming Languages*, 2012.

[17] R. Andrade, "Privacy and security constraints for code contributions," in *SPLASH Doctoral Symposium*, 2015.

[18] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, pp. 399–422, 2009.

[19] Securibench Micro. [Online]. Available: http://suif.stanford.edu/~livshits/work/securibench-micro/

[20] Online Appendix. [Online]. Available: https://sites.google.com/site/rodrigocaa/programming2018

[21] OWASP - code review guide. [Online]. Available: https://www.owasp.org/images/2/2e/OWASP_Code_Review_Guide-V1_1.pdf

[22] OWASP - open web application security project. [Online]. Available: https://owasp.org/

[23] Using Pull Requests. [Online]. Available: https://help.github.com/articles/using-pull-requests/

[24] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *International Conference on Software Engineering*, 2016, pp. 285–296.

[25] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. B. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative verification of information flow for a high-assurance app store," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1092–1104.

[26] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Fundamental Approaches to Software Engineering*, 2013.

[27] S. Chong, K. Vikram, and A. C. Myers, "Sif: Enforcing confidentiality and integrity in web applications," in *USENIX Security Symposium on USENIX Security Symposium*, 2007, pp. 1–16.

[28] S. Chong and A. C. Myers, "Security policies for downgrading," in *Conference on Computer and Communications Security*, 2004, pp. 198–209.

[29] K. J. Biba, "Integrity considerations for secure computer systems," Tech. Rep. ESD-TR-76-372, 1977.

[30] S. Fink and J. Dolby. WALA, The T.J. Watson Libraries for Analysis. [Online]. Available: http://wala.sourceforge.net/

[31] J. Newmsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of Network and Dist. System Security Symp.*, 2005.

[32] W. Enck, P. Gilbert, B. g. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones," in *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[33] J. Graf, M. Hecker, and M. Mohr, "Using JOANA for information flow control in Java programs - a practical guide," in *Working Conference on Programming Languages*, 2013.

[34] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *ACM Symposium on Principles of Programming Languages*, 1999.

[35] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *International Journal of Information Security*, vol. 6, pp. 67–84, 2007.

[36] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 26–60, 1990.

[37] S. Horwitz, J. Prins, and T. Reps, "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 345–387, 1989.

[38] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends in Programming Languages*, vol. 2, pp. 1–69, 2015.

[39] J. Graf, M. Hecker, and M. Mohr, "Security policies and security models," in *Proc. of IEEE Symposium on Security and Privacy*, 1982.

[40] L. Zheng and A. C. Myers, "A language-based approach to secure quorum replication," in *Workshop on Programming Languages and Analysis for Security*, 2014.

[41] J. H. Saltzer, "Protection and the control of information sharing in multics," *Communications of the ACM*, vol. 17, pp. 388–402, 1974.

[42] Git. [Online]. Available: https://git-scm.com

[43] Github Popularity. [Online]. Available: https://github.com/blog/2345-celebrating-nine-years-of-github-with-an-anniversary-sale

[44] CWE - common weakness enumeration. [Online]. Available: https://cwe.mitre.org/

[45] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security," in *USENIX security symposium*, 2011.

[46] Gitblit: Open-source, pure Java stack for managing, viewing, and serving Git repositories. [Online]. Available: http://gitblit.com

[47] Gitblit commit. [Online]. Available: http://tinyurl.com/q9xlueq

[48] SHA1 collision. [Online]. Available: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html?m=1

[49] Hashkiller. [Online]. Available: https://hashkiller.co.uk/sha1-decrypter.aspx

[50] Blojsom. [Online]. Available: https://sourceforge.net/projects/blojsom/

[51] SLF4J -simple logging facade for java. [Online]. Available: http://www.slf4j.org

[52] A. Mettler, D. Wagner, and T. Close, "Joe-e: A security-oriented subset of java," in *Network and Distributed System Security Symposium*, 2010.

[53] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *International Conference on Software Engineering*, 2008, pp. 311–320.

[54] Simple OAuth library for Java. [Online]. Available: https://github.com/scribejava/scribejava

[55] D. Hardt, "The oauth 2.0 authorization framework," 2012.

[56] Open Refine. [Online]. Available: http://openrefine.org

[57] Issues in Github. [Online]. Available: https://guides.github.com/features/issues/

[58] S. Arzt, S. Rasthofer, and E. Bodden, "SuSi: A tool for the fsully automated classification and categorization of android sources and sinks," Tech. Rep. TUD-CS-2013-0114, 2013.

[59] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," in *Communications of the ACM*, 1972.

[60] ANTLR. [Online]. Available: http://www.antlr.org/

[61] W. F. Tichy, "Rcs - a system for version control," *Software: Practice and Experience*, vol. 15, pp. 637–654, 1985.

[62] V. Basili, G. Caldiera, and D. H. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, J. J. Marciniak, Ed. Wiley, New Jersey, 1994, pp. 528–532.

[63] An open source clone of Amazon's Dynamo. [Online]. Available: https://github.com/voldemort/voldemort

[64] A blogging system written in Java. [Online]. Available: https://github.com/b3log/solo

[65] A solid high-performance JDBC connection pool. [Online]. Available: https://github.com/brettwooldridge/HikariCP

[66] Apache Kafka. [Online]. Available: https://github.com/apache/kafka

[67] Feedback management tool for education. [Online]. Available: https://github.com/TEAMMATES/teammates

[68] Open Source Web Crawler for Java. [Online]. Available: https://github.com/yasserg/crawler4j

[69] GitHub API. [Online]. Available: https://developer.github.com/v3/

[70] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[71] SourceForge. [Online]. Available: https://sourceforge.net/

[72] Personalblog. [Online]. Available: https://sourceforge.net/projects/personalblog/

[73] GridSphere. [Online]. Available: https://github.com/brandt/GridSphere

[74] SnipSnap. [Online]. Available: https://github.com/thinkberg/snipsnap

[75] Lutece. [Online]. Available: https://github.com/lutece-platform/lutece-core

[76] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," in *International Symposium on Software Testing and Analysis*, 2011, pp. 177–187.

[77] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslen, *Experimentation in software engineering*. Springer, 2012.

[78] V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: Automatic security analysis of smartphone applications," in *Conference on Data and Application Security and Privacy*, 2013, pp. 209–220.

[79] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis," in *USENIX Security Symposium*, 2012, pp. 569–584.

[80] O. Tripp and J. Rubin, "A bayesian approach to privacy enforcement in smartphones," in *USENIX Security Symposium*, 2014, pp. 175–190.

[81] D. Binkley, M. Harman, and J. Krinke, "Empirical study of optimization techniques for massive slicing," *ACM Transactions on Programming Languages and Systems*, vol. 30, 2007.

[82] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4f: taint analysis of framework-based web applications," *ACM SIGPLAN Notices*, vol. 46, pp. 1053–1068, 2011.

[83] Apache Wicket. [Online]. Available: http://wicket.apache.org

[84] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized javascript. [Online]. Available: https://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf

[85] S. Apel, S. S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich, "Access control in feature-oriented programming," *Science of Computer Programming*, vol. 77, pp. 174–187, 2010.

[86] C. Prehofer, "Feature-oriented programming: A fresh look at objects," in *European Conference on Object-Oriented Programming*, 1997.

[87] S. Apel, C. Kastner, and C. Lengauer, "Feature house: Language-independent, automated software composition," in *International Conference on Software Engineering*, 2009.

[88] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller, "Detecting information flow by mutating input data," in *IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 263–273.

[89] K.-C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Tech. Rep. CMU/SEI-90-TR-21, 1990.

[90] OWASP WebGoat. [Online]. Available: https://github.com/WebGoat/WebGoat

[91] X. Zhang, R. Grigore, X. Si, and M. Naik, "Effective interactive resolution of static analysis alarms," in *Proceedings of Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2017, pp. 25–55.

[92] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An overview of aspectj,” in *European Conference on Object-Oriented Programming*, 2001, pp. 327–354.

Appendix A - Salvum Grammar

The grammar for Salvum is shown in Figure A.1. A program represents a set of constraint or constant declarations. The former represents a constraint itself specifying the information to be protected either using the `noflow`, `noset`, `flow`, or `set` constructs. The latter defines a constant that can be used in many constraints. A module might be an identifier (e.g., a `Contribution`), parts of a program (e.g., a method call), or a sequence of fields that store sensitive information.

```
<program> ::= (<constraint-declaration>  (";" <program>)*) | (<constant-declaration> | (";" <program>)*)
<constraint-declaration> ::= <module> "noflow" <module> <where-clause>?
   | <module> "noset" <module> <where-clause>?
   | <module> "flow" <module> <where-clause>?
   | <module> "set" <module> <where-clause>?
<where-clause> ::= "where" <constant-declaration> (","constant_declaration)*
<constant-declaration> ::= <ID> "=" <sensitive-info> | <module>
<module> ::= <ID> | "{" <program-parts> "}" | <sensitive-info>
<sensitive-info> ::= <ID> | <sensitive-fields> ("," <sensitive-fields>)*
<sensitive-fields>  ::= <class> "{" <fields> "}"
<program_parts> : <single_method_call> ("," <single_method_call>)*  | ID "|" <contribution_expression>
   | <commit_hash> ("," <commit_hash>)*
<contribution_expression> : <contribution_spec> | <contribution_expression> "&&" <contribution_expression>
   | <contribution_expression> "||" <contribution_expression> | "!" <contribution_expression>
<contribution_spec> ::= <method_call>
<method_call> ::= <single_method_call> ("." <method_call>)*
<single_method_call> ::=  <full_name> "(" <argument_list>? ")"
<argument_list> ::=  <argument> ("," <argument>)*
<argument> ::= <author_id> |<contribution_id>
<author_id> ::= <STRING>
<contribution_id> ::= <STRING>
<commit_hash> ::= <STRING>
<clazz> ::= <full_name>
<full_name> ::= <ID> ("." <ID>)*
<fields>  ::= <ID> ("," <ID>)*
<ID> ::= ("_" | "$" | <LETTER>)( "_" | "$" | <LETTER> | <DIGIT> )*
```

Figure A.1: Salvum grammar

Salvum also supports combinations of expressions. For instance, we could declare that a set of sensitive information cannot flow to contributions made by a particular author and contain a specific commit message. Finally, we can also specify constraints considering just commit hashes. For example, in case we want to enforce a constraint for a given project version.