Received: 10 June 2019 | Revised: 27 December 2019 | Accepted: 1 June 2020

DOI: 10.1002/spe.2872

RESEARCH ARTICLE

WILEY

Privacy and security constraints for code contributions

Rodrigo Andrade[1] | **Paulo Borba**[2]

[1]Unidade Acadêmica de Garanhuns, Universidade Federal Rural de Pernambuco, Pernambuco, Brazil

[2]Informatics Center, Federal University of Pernambuco, Pernambuco, Brazil

Correspondence
Rodrigo Andrade, Unidade Acadêmica de Garanhuns, Universidade Federal Rural de Pernambuco, Avenida Bom Pastor, s/n, Pernambuco, Brazil.
Email: rodrigo.caandrade@ufrpe.br

Funding information
Conselho Nacional de Desenvolvimento Científico e Tecnológico, Grant/Award Number: 141590/2013-0

Summary

In collaborative software development, developers submit their contributions to repositories that are used to integrate code from various collaborators. To avoid privacy and security issues, code contributions are often reviewed before integration. Although careful manual code review can detect such issues, it might be time-consuming, expensive, and error-prone. Automatic analysis tools can also detect privacy and security issues, but they often demand significant developer effort, or are domain specific, considering fixed framework specific vulnerability sources and sinks. To reduce these problems, in this paper we propose the Salvum policy language to support the specification of constraints that help to protect sensitive information from being inadvertently accessed by specific code contributions. We implement a tool that automatically checks Salvum policies for systems of different technical domains. We also investigate whether Salvum can find policy violations for a number of open-source projects. We find evidence that Salvum helps to detect violations even for well-supported and highly active projects. Moreover, our tool helps to find 80 violations in benchmark projects.

KEYWORDS

collaborative software development, information flow control, policy language, privacy, security

1 | INTRODUCTION

Many software systems handle sensitive information such as confidential user data. Developers must then be concerned about how to protect such information. This especially holds in collaborative software development projects, which rely on version control repositories that are used to integrate code contributions (sets of changes expressed as commits, pull requests, etc.) from a number of collaborators. Unless project leaders take appropriate measures to protect sensitive information, integrated code contributions might carelessly or maliciously introduce privacy and security violations.[1,2] The situation is worse when inexperienced or untrustworthy developers are involved in the project. The consequences are also aggravated by the fact that introduced violations can remain unnoticed for years, even in open source projects.[3]

Classical approaches such as cryptography and certificates might possibly reduce the effects of violations, but do not actually avoid or detect them.[4] To avoid privacy and security issues, code contributions in collaborative software development projects are often reviewed before integration. Although careful manual code review can detect such issues, it might be time-consuming, expensive, and error-prone.[5-7] Automatic analysis tools can also detect privacy and security issues, and reduce some of the manual code review drawbacks.[8] But some of the tools demand significant developer effort to prepare each code contribution for analysis.[9,10] Other tools are domain specific, considering fixed framework specific vulnerability,[11-13] requiring significant adaptation for other technical domains. Additionally, some approaches allow developers to mitigate the significant effort issue, but they do not support code contribution analysis.[14,15]

To reduce the discussed problems, we propose the Salvum policy language to support the specification of constraints that help to protect sensitive information from being inadvertently accessed by specific code contributions. For instance, Gitblit is a git repository manager that uses sensitive information such as password to its authentication mechanism. A logical policy for Gitblit states that the user password cannot flow to code contributions that add code related to a cookie mechanism since it could leak the password. Therefore, Salvum allows us to write constraints specifying that a user password cannot flow to the execution of code contribution that uses the cookie mechanism. In other words, this code contribution cannot access the field that holds user password information. Salvum allows developers to define these constraints for systems of different technical domains, as it does not rely on a fixed set of domain-specific potential vulnerability sources and sinks. To automatically check Salvum policies, we implement an information flow control (IFC)-based tool[9,16] that statically analyzes Java source code adherence to the underlying policy constraints; the same concepts and architecture could be used to implement similar tools for other languages and static analysis.

Roughly, our proposal works as follows: a developer writes Salvum policies specifying sensitive information and the code contributions of interest. Then, she runs our tool to check for policy violations. Our tool detects a collection of violation warnings, indicating the lines of code involved in each potential violation, and the contributions (commits) that introduced each violation. At last, the developer can review the code contributions to confirm the violations.

To investigate whether Salvum can indeed find relevant violations in the context of code contributions, we first conduct an experiment considering nine highly active and well-supported open-source software projects. We study the projects' source code, write project-specific constraints using our policy language, and execute our checking tool. We find evidence that, even in this kind of well-curated projects, and with limited knowledge of requirements, architecture, and source code, Salvum helps to detect violations. Additionally, to better understand the results, we manually analyze discussions and code commits in a number of GitHub pull requests associated with the nine projects. We further evaluate Salvum effectiveness to detect violations in five benchmark projects, finding 80 violations. Finally, we informally assess whether Salvum helps to reduce the effort to find violations, when compared with a potential manual code review.

We structure the remainder of the paper as follows. Section 2 discusses four scenarios that illustrate the problem that we want to tackle. Section 3 explains our language, giving an overview of Salvum and details of the language specification and implementation. In Section 4, we present the evaluations of our approach to answer a number of research questions related to Salvum's potential to find relevant violations and reduce detection effort. Section 5 discusses related work, and Section 6 presents the final considerations of this work. We provide all artifacts necessary to reproduce this work in our online Appendix.[17] At last, we previously reported our results in the first author PhD thesis.[18]

2 | PROBLEM

The common weakness enumeration (CWE) has a catalog of more than 1000 kinds of software weaknesses.[19] These are different ways that software developers can introduce violations that lead to nonsecure 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.[20] Indeed, other researchers have found many problems in Java projects.[5,12,21,22]

In this section, 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 2.1, the second one in Section 2.2, the third one in Section 2.3, and the last one in Section 2.4. 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.

2.1 | First scenario: Code contribution introducing a violation

To better illustrate the problem we want to mitigate, in this section we explain an integration scenario from the Gitblit[1] open-source Java project, which is supported by several developers who submit their code contributions to a GitHub repository. Gitblit is essentially a git repository manager with a web user interface that administrators can use to create,

[1] http://gitblit.com

edit, rename, or delete repositories, users, and teams. To identify the web interface users, Gitblit implements a cookie mechanism. This mechanism was actually affected by one of the project contributions that resulted from an enhancement task assigned to one developer.[2] Her task was to improve the Gitblit code that deals with Java Servlets. Among other code changes in the resulting contribution,[2] the developer added Line 5 in Listing 1 to set a 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  }
```

Listing 1: Setting a user cookie

As the `setCookie` method receives the `user` object, which is an instance of `UserModel`, the developer responsible for this contribution likely has not noticed that `UserModel` instances hold sensitive user information such as `password`, `email`, and `location`. Since the developers responsible to manually review code contributions in Gitblit have also not identified any problem, this `setCookie` call has been merged into Gitblit's main repository. As a consequence, when one executes the `loginUser` method, sensitive user information unnecessarily flows in an implicit way to `setCookie`, which is called in the new code contribution (Line 5). This violates the Principle of Least Privilege,[23] which states that every program part should operate using the least amount of privilege necessary to correctly complete the job.

In this case, the mentioned violation is critical because the new code contribution, likely created and submitted by an inexperienced developer, should not have access to the sensitive user information contained in the `user` object, especially the one stored in `UserModel.password`. The code resulting from integrating the new contribution could actually expose sensitive user information to any attacker possessing access to user browser data.

Although a more rigorous manual code review process could have found the illustrated violation and avoided integration into the main repository, this would have demanded the reviewer to read and understand 142 additions and 142 deletions in the Gitblit source code only for this not particularly large contribution[2]. The reviewer would also have to assess the potential interactions among each of the additions and deletions, which might be hard even for smaller contributions. This person would have to be an expert in the system in order to know not only the information that should be protected, but also the program parts, and contributions, that could access it. In the case of Gitblit, which contains a lot of sensitive information, this could be a hard task.

To mitigate the mentioned drawbacks of purely manual code reviews, we could explore the potential of IFC analysis[16] as implemented in frameworks such as JOANA.[9] Although that is promising, we would still have to manually identify and label bytecode instructions that correspond to sensitive information and code contributions of interest. This task might be *time-consuming* and *error-prone*, even using a graphical user interface such as the one provided by JOANA. Other static analysis-based approaches like Jif[10] and Checker Framework[24] would require us to annotate the source code to specify program elements that represent sensitive information, and they have no mechanisms for easily handling the abstractions of code contributions. Android-based static analysis[11,13] avoid the annotation effort but *lack generality* because they are designed to work only for specific technical domains, considering fixed framework specific vulnerability sources and sinks.

To handle the existing automated approaches limitations, in Section 3, we present Salvum and its checking tool, which detected the illustrated violation. The problematic code contribution of Listing 1 was integrated to the main Gitblit repository on July 2014. We found this problem by analyzing Gitblit's project history using our tool, and we have immediately reported the potential leak. Gitblit developers requested a pull request. We submitted a fix, and it was integrated on December 2016. As illustrated, it might be difficult to detect violations, and they might exist for a long period before they are detected and fixed.[3]

[2]http://tinyurl.com/q9xlueq

2.2 | Second scenario: Sensitive information leaking through third-party classes

Our second scenario illustrates an issue in the Blojsom project,[3] 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),[25] which is useful to log operations throughout the source code. These logging operations are known to be a potentially dangerous point of information leak[26] as specified in the CWE[19] and in The Open Web Application Security Project.[27] 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 1 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 lines 7 and 8 log 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).

```
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 }
```

Listing 2: Logging passwords

The code in Listing 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 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[9] 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[24] or Jif,[10] we would have to scatter and tangle policy code with core code, which could make it harder to evolve and maintain the system.[28]

2.3 | 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 project.[4]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.

[3]https://sourceforge.net/projects/blojsom/
[4]https://github.com/scribejava/scribejava

The OAuth[29] 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. On February 2016, a developer submitted a code contribution with the goal of updating token representations.[5]Among 559 changed lines throughout 33 files, this developer implemented `equals()` and `toString()` methods. Listing 3 illustrates one `toString()` implementation introduced by this code contribution.

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

Listing 3: Implementing toString method

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 (lines 2, 3, and 4) 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.[6]Since it was an easy fix, this developer submitted a new commit excluding the token secret from the `toString`method.[7]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. 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 2.1, the task of manually annotating code contributions might be time-consuming and error-prone if we use JOANA.[9] 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,[10] we would have the same drawback: to manually annotate sources and sinks throughout Java source code for each code contribution.

2.4 | Fourth scenario: Untrustworthy developer introducing violations

The fourth scenario is a hypothetical example of the Open Refine project.[8]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.

Some of these developers might be considered untrustworthy, for example, 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 from a company.

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 4 illustrates a snippet of code of his contribution. The line 3 represents the added code, whereas the line 4 represents the removed code.

[5]https://github.com/scribejava/scribejava/commit/d734a4df
[6]https://github.com/scribejava/scribejava/issues/796
[7]https://github.com/scribejava/scribejava/commit/73c29f5
[8]http://openrefine.org

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

Listing 4: Removing public check

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.[17] 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[9] 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 the Checker Framework[24] or Jif[10] could be unfeasible because we would need to manually annotate security types for 813 different versions.

3 | THE SALVUM POLICY LANGUAGE

To mitigate the problems discussed in the previous section, we propose the Salvum policy language to support the specification of constraints that help to protect sensitive information from being inadvertently accessed by specific code contributions. Our main goal with Salvum is to find sensitive information confidentiality and integrity violations. By using our language, we can check whether changes submitted by a developer (a code contribution), or even specific classes (like in unreliable external libraries), access sensitive information they should not access. To better illustrate how Salvum works, we first overview its main expected usage scenarios, and later we explain its main features and implementation.

3.1 | General idea

In a collaborative software development context, 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 way, a developer (architect, code reviewer, etc.) could use Salvum to specify a set of application-specific constraints to avoid the problems we discuss in Section 2. Our language constructs allow a developer to refer to program elements associated with sensitive information (like password, location, medical data, etc.), and parts of the code expressed either as program elements or as code contributions (like commits, pull requests, etc.), specifying restrictions on how they are related. For example, Salvum allows us to specify that a user password cannot flow to the execution of parts of the code contributed by a set of untrustworthy developers.

We can use Salvum in two kinds of usage scenarios: *backward* and *forward*. The *backward* scenario is useful when we are supposed to check project history to make sure constraints have not been violated. For instance, consider that a company fires a developer due to misbehavior. Assuming this developer has contributed with several commits across the employment period, it could be useful to check potential sensitive information violations in all these commits. To do that, one could write Salvum constraints to specify that sensitive information should not flow to the code added or changed by these commits. In a *forward* scenario, we are supposed to check code contributions just before they are integrated into a team repository. For example, consider an open-source project that receives many code contributions from different

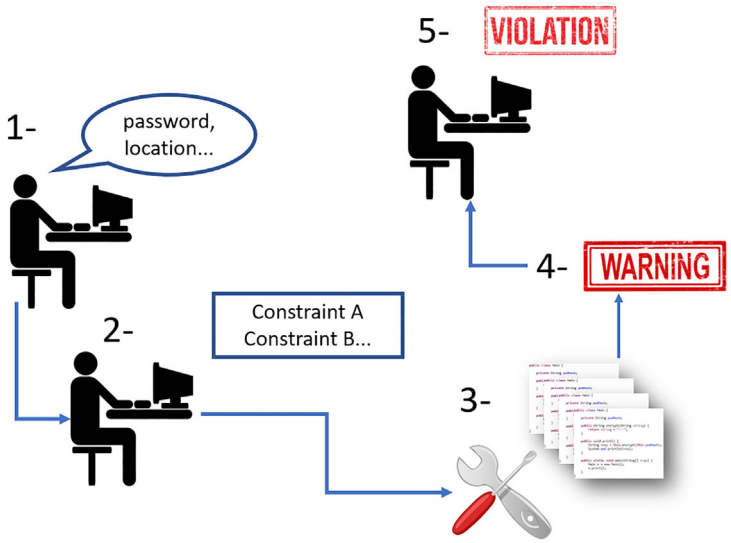

FIGURE 1 General idea [Colour figure can be viewed at wileyonlinelibrary.com]

developers, some of which are either naive or untrustworthy. So, a project developer leader could write Salvum constraints to check whether sensitive information leaks to these code contributions. This way, before integrating code contributions into the repository, we can ask developers to fix detected violations.

Figure 1 depicts how our solution works. First, a specialist identifies system specific sensitive information. Second, she writes Salvum constraints that restrict how this information can be accessed by different parts of the code or contributions. Third, Salvum automatically checks source code adherence to these constraints, and reports violation warnings. At last, a specialist determines which warnings correspond to relevant violations and take appropriate measures to fix them.

3.2 | Language features

Salvum supports four main constructs: `noflow`, `noset`, `flow`, and `set`. They all relate program elements that declare sensitive information to parts of the code expressed either as program elements or sets of contributions. Different from other approaches,[10-12,15,21] our language supports the expression of these sets of code contributions. The `noflow` construct, in particular, specifies that a set of sensitive information cannot flow to certain parts of the code. For example, the declaration below specifies that the `password` information cannot be accessed by the class `CookieManager`.

```
UserModel {password} noflow CookieManager
```

In this simple case, we specify a single sensitive information by referring to its identifier (`password`) and the class that declares it (`UserModel`). Similarly, we specify a single part of the code by referring to a class identifier (`CookieManager`). Although simple, such constraint could help to prevent the problem introduced in Section 2. Constraints like this, based on `noflow`, impose no other constraint on information (such as name or login) not declared in the constraint. By making sure sensitive information does not leak, the `noflow` construct guarantees information *confidentiality*.

Complementing `noflow` and guaranteeing information *integrity*, the `noset` construct specifies that sensitive information cannot be altered by certain parts of the code. For example, the declaration below specifies that the `password` information cannot be changed by the class `CookieManager`.

```
CookieManager noset UserModel {password}
```

In addition to the `noflow` constraint, we could use the `noset` declaration above to prevent a problem even more critical than the one introduced in Section 2. Again, we are not imposing further constraints on the specified class (`CookieManager`). It could, for example, write to sensitive information like `location`.

In contrast to `noflow`, Salvum supports the `flow` construct. It specifies that only a certain set of information can flow to certain parts of the code. No other information is allowed to flow to specified parts of the code. For example, the following constraint specifies that only user `login` can flow to the specified class.

```
UserModel {login} flow CookieManager
```

Similarly, in contrast to `noset`, we provide the `set` construct, which specifies that only a certain set of information can be altered by certain parts of the code. 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. This code contribution cannot alter other information (like the user e-mail address). Together, the `flow` and `set` constructs can be used to enforce the *Principle of Least Privilege*:[23] a code contribution should use the least amount of privilege necessary to correctly complete its execution.

Although the previous simple examples illustrate constraints that refer to a single sensitive information and a single part of the code expressed as a class, our language allows us to specify more than one specific sensitive information, possibly declared in different classes. It also allows us to represent parts of the code expressed as a number of classes or code contributions. Such contributions can be specified in three distinct ways: intentionally, extensionally, and parametrically.

Intentionally. We specify code contributions by stating properties they satisfy. For example, we can say that the contributions of interest correspond to the set of commits in project history that contains a particular message keyword. The example below specifies the only information (`login`, `nickname`, or `countrCode`) that can flow to commits in a given pull request, assuming that all commits in this pull request have `#921` (the pull request identifier) in their messages.[30]

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

Here we assume that only the commits from a given pull request have `#921` in their message, but we could write more complex expressions to refer exactly to the commits of interest. Alternatively, we could specify code contributions by referring to a specific project issue[31] or task, or even a feature, as long as we can identify their commit messages by some property, such as using `#921`.

To illustrate another kind of property that can be explored to define contributions, consider the following use of the `set` construct to specify the information that can be altered by a code contribution.

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

Here we use the commit `author` information 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 changes public user information. The illustrated constraint verifies whether the code Bob committed alters information other than those stored in `login` and `nickname`.

Extensionally. Salvum also supports the specification of program elements or contributions by simply listing them. The example below specifies that only a specific set of public information can flow to a specific set of two methods (abbreviated as `Log`) of the `Logger` class; other information cannot flow.

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

This constraint might help to assure that only public information flows to public outputs, such as `Log`methods.

Parametrically. We can also specify code contributions in an abstract way by referring to a parameter that is provided when invoking Salvum's interpreter. For example, `Contribution` in the example below is a parameter, to be instantiated when invoking the interpreter. Such parameter does not represent a fixed specified set of commits.

This type of specification is useful when we want to check constraints for arbitrary sets of commits, as we did to detect the problem introduced by the code contribution of Listing 1 in Section 2. In that case, we simply checked the constraint for a subset of Gitblit commits that we had previously built, as checking for all commits would be too expensive.

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

We first identified fields that could store sensitive information, such as `password` and `locality` in `UserModel`. The `noflow` construct specifies that this set of sensitive information cannot flow to lines changed or added by the code contribution represented by `Contribution`. By properly instantiating `Contribution` with the hashes of commits we had previously built, we identified the violation illustrated in Section 2.

3.3 | Salvum grammar

Part of the grammar for Salvum is shown in Figure 2. 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 (eg, a `Contribution`), parts of a program, such as a method call, or a sequence of fields that stores sensitive information.

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. We can also specify constraints considering only commit hashes. For example, in case we want to enforce a constraint for a given project version. Finally, our constraints define boundaries for code contributions or classes. Salvum sees these elements as software modules.[32] In Section 3.4, we explain it in more detail as well as Salvum's implementation.

3.4 | 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 a set of IFC Analysis.[16] This way, we can use these analyses to detect information flows that might violate the specified constraints.

Our tool has the following implementation steps: *Preprocessing*, *Generating System Dependence Graphs (SDG)*, *Labeling*, *Analyzing*, and *Results processing*. Figure 3 illustrates them as well as their inputs and outputs.

3.4.1 | First step - Preprocessing

As shown in Figure 3, 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 depends on the constraints.

The first one regards the specification of code contributions as Operations listing. In this case, we defined an algorithm to find occurrences in the source code of calls to methods that are specified by the constraint. In this context, our algorithm

```
<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>
```

FIGURE 2 Salvum simplified grammar

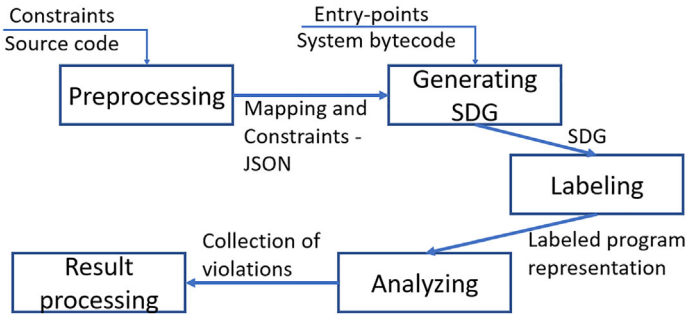

FIGURE 3 Five steps of our tool implementation [Colour figure can be viewed at wileyonlinelibrary.com]

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 different security levels[16] and to mitigate the *time-consuming and error-prone* problem, which we explained in Section 2.

The second preprocessing procedure concerns the specification of explicit and implicit code contributions. Therefore, we need to deal with program history in Version Control Systems.[34] For now, Salvum supports Git.[35] In this context, the Git `diff` command generates a `diff` file, which contains the differences between existing code and code contribution. We defined 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 1 is `[RootPage:5]`. This mapping is also useful for automatically labeling source code to execute an IFC analysis and detect illegal flows.

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 do this task, we use the `checkout` Git command either automatically or manually depending on the project and constraint. Additionally, we built a parser to validate the syntax of the constraints and to generate a corresponding JSON[36] containing the specification of sensitive information and code contributions.

3.4.2 | Second step - Generating SDG

In this step, we create the necessary SDG[37] using JOANA.[9] In these dependence graphs, graph nodes represent program statements or expressions. Moreover, there are two kinds of edges: data dependence and control dependence.[16] 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`.

We provide as inputs the set of entry methods and the compiled system versions. 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. Depending on the constraint, we might need to generate many SDGs in this step. For instance, recalling the fired developer example: 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 SDG.

3.4.3 | Third step - Labeling

Our tool automatically annotates information and code contributions so that we can detect flows between them in either direction depending on the specified constraints. We use JOANA annotation mechanism to label a set of program parts with security levels (eg, *High* and *Low*).

Based on the constraints written in Salvum and the first step output mapping explained before, our implementation automatically labels program parts that represent sensitive information for `noflow` construct, as *High*. In contrast,

program parts identified in the mapping as part of code contributions are automatically labeled as *Low*. To deal with positive constructs (`flow` and `set`), we use the inverse labeling strategy of `noflow` and `noset` constructs since they are dual.[38]

For instance, consider again the constraint for Listing 1: `UserModel {password, locality} noflow Contribution`

It 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. In this context, our algorithm determines the code contribution by analyzing the Git `diff`between the commit we aim to check and its predecessor. 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.

3.4.4 | Fourth step - Analyzing

In this step, Salvum automatically runs JOANA[9] IFC 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 violation warnings, which we provide as an input to the last step: *Results processing*.

3.4.5 | Fifth step - Results processing

Salvum creates a report to inform violations occurrences. For instance, consider Listings 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

This way, we know that there is a warning of 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`. We manually check this violation warning to determine that this code contribution introduces a violation.

4 | EVALUATION

To evaluate our policy language presented in Section 3, we perform an empirical assessment focusing on finding violations by writing Salvum policies and constraints for selected software projects and checking whether they can help developers to reduce the effort of manual code review.

In Section 4.1, we discuss the Goal-Question-Metric (GQM) design[39] that drives our evaluation in Section 4.1. Additionally, in Section 4.2, we perform the first evaluation considering highly active and well-supported software projects. In Section 4.3, we explain our second evaluation with benchmark 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. Also, we discuss the threats to validity in Section 4.4.

4.1 | Goal, questions, and metrics

We use the GQM[39] design to better drive the evaluation process. Our goal is to check whether we can detect privacy and security violations concerning real software projects. Besides that, we aim to determine if Salvum can reduce the effort to find these violations. In particular, we investigate two major and two minor research questions:

- **Q1**- Can Salvum detect code contribution violations to sensitive information?
- **Q1.1**- Do developers solve violations before code contribution merging on GitHub?

- **Q1.2**- Are unmerged code contributions on GitHub related to violations of sensitive information?
- **Q2**- Can Salvum reduce the effort to find violations of specified constraints in a set of code contributions?

To answer our research questions, we define a set of simple metrics. Regarding **Q1**, we use the number of violation warnings **(NVW)** and Number of violations **(NV)** metrics. The former regards the NVW 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 NV before merging code contributions **(NvC)** metric to answer **Q1.1**. Additionally, to answer **Q1.2**, we manually check unmerged pull requests to know if they were not merged due to violations in our context. Thus, we use the number of unmerged code contributions containing violations **(NrC)**metric.

To answer **Q2**, we consider two metrics: Source lines of code to analyze **(SA)** and 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, 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 10 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.

4.2 | Assessing highly active and well-supported software projects

In this section, we explain nine highly active software projects. Furthermore, we define a set of constraints for them. At last, we describe the results we obtained. Our goal here is to answer **Q1**, **Q1.1**, **Q1.2**, and **Q2**. To do that, we use the *backward* approach (Section 3). In this case, it is more interesting than the *forward* approach because the existing project history has more commits, and consequently merged code contributions. Nonetheless, we plan to perform an additional evaluation considering our *forward* approach in future work.

4.2.1 | Selected software projects

For this assessment, we consider the following open-source Java software projects hosted on GitHub: Gitblit,[9] Open Refine,[10]Voldemort,[11]ScribeJava,[12]Solo,[13]HikariCP,[14]Apache Kafka,[15]Teammates,[16]and Crawler4J.[17]These projects have at least 100 stars and 50 forks as well as many pull requests. Table 1 summarizes some of these projects' characteristics.

In this context, we obtained a list of the first 300 Java projects with the criteria mentioned above. Then, we visited each GitHub project repository webpage to choose those that would be useful for our research. Initially, we discarded the projects that are Android applications. Furthermore, we put away projects that we could not find a set of 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 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 developers manually review the changes that other developers submit via pull requests. Therefore, we might find a few critical violations regarding the repository history. However, these 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 4.4.

[9]http://gitblit.com
[10]http://openrefine.org
[11]https://github.com/voldemort/voldemort
[12]https://github.com/scribejava/scribejava
[13]https://github.com/b3log/solo
[14]https://github.com/brettwooldridge/HikariCP
[15]https://github.com/apache/kafka
[16]https://github.com/TEAMMATES/teammates
[17]https://github.com/yasserg/crawler4j

TABLE 1 Characteristics of selected software projects

Project	KSLOC	Commits	Contributors	Total Pull R equests	GitHub stars	GitHub forks
Gitblit	123	3.000	100	400	1.400	500
Open Refine	75	2.700	60	250	5.000	900
Voldemort	225	4.300	60	350	2.000	500
ScribeJava	5	800	90	350	4.300	1.500
Solo	41	2.000	15	190	3.900	1.700
HikariCP	17	2.600	80	250	6.000	900
Apache Kafka	113	4.700	400	4.500	7400	4.400
Teammates	145	16.600	350	3.100	500	1.200
Crawler4J	8	300	25	100	2.600	1.500

4.2.2 | 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. Then, we define policies and constraints for each project with the goal to enforce them for the selected projects. An experienced developer on a particular project could play this role. For simplicity, we explain only those policies and constraints that helped us to detect the Gitblit problem we explained in Section 2. However, we provide a complete list of constraints we wrote for each project in our online Appendix.[17]

Policy P1: User password, locality, access information, and permission cannot flow to code contributions

```
1        ChangePasswordPage {password, confirmPassword},
2        UserModel {password, locality},
3        AccessPermission {perm},
4        RepositoryModel {accessRestriction, authControl}
5         noflow Contribution
```

Listing 5: Constraint C1

```
1        ChangePasswordPage {password, confirmPassword},
2        UserModel {password, locality},
3        AccessPermission {perm},
4        RepositoryModel {accessRestriction, authControl}
5         noflow WrittingOps
6          where
7           WritingOps = {Logger.info(), Logger.error(),
8           Logger.warn(), System.out.println(), setCookie()}
```

Listing 6: Constraint C1'

In **P1** we specify a set of 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 set of 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 set of sensitive information and code contributions. Notice that we use the `Contribution`identifier,

which implicitly defines the contributions to be analyzed. In our assessment, the `Contribution` identifier represents 49 Gitblit 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.

4.2.3 | Answering Q1

In particular, Table 2 illustrates the number of different software versions we analyzed, and the NV we have found for each selected software project.

For the cases that 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 violation, 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 harmful. For example, 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. Thus, we do not report a violation.

This scenario is typical when analyzing initial commits. In summary, the procedure to obtain the results of this evaluation is as follows: 1) We write constraints for a software project; 2) We run our tool to detect these constraints; 3) We manually analyze the set of violation warnings; 4) We decide whether the warnings are real violation; 5) We report the violations 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 contributors 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 4.4. The remaining projects either handle a few sensitive information or present fewer contributors or commits. Thus, we consider 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 observe that many code contributions only fix user interface presentation, typos, and configuration files. These kinds of commits do not add or change lines of Java code, which avoids introducing violations.

In Table 2, the `NVW` metric results show that we found six violations warnings for the selected projects of which five were confirmed as a violation by 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. Only the path of a sensitive file was exposed and not the file content itself.

Regarding Gitblit, we found two violations of the constraints we explained in this section. In particular, we illustrate the output Salvum provides for constraint **C1** below. We use this violation as motivating example in Section 2. This output

TABLE 2 Total number of warnings and violations

Project	Versions	NVW	NV
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

indicates that there is an illegal flow of the information initially stored in `password` to line 284 in `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. Afterward, we submitted a pull request to fix it. We solve it by implementing a new way of generating a hash for the cookie mechanism for Listing 1. 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 opened an issue to change the hash function after we warn its weakness.

Based on `NVW` and `NV` results shown in Table 2, we answer **Q1** stating that Salvum can detect proper violations of sensitive information and code contributions in the context of software projects that are highly active and well-supported.

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, one author of this work read developer discussions of at most 25 merged pull requests containing at least two messages attached considering the nine projects. We choose 25 because while performing this analysis, we noticed the reasons for 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 3 depicts the number of pull request we manually analyzed for each project.

Following References 40 and 41, 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 requests with at least two messages whereas Crawler4J has only 10. Thereby, we analyzed a total of 198 pull requests and their messages.

TABLE 3 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	13	14	8	35	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	64	61	76	198	6

The main reason for discussions throughout these 198 pull requests in GitHub is to: (i) include or run tests, (ii) obey development guidelines like code indentation, (iii) change 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 3 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. Nonetheless, we analyzed the code of these six pull requests, and none of them fixed violations.

In this way, these manually analyzed pull requests ratify our previous findings with Salvum. Only a small amount of code contributions is 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.

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 there is no evidence on GitHub that developers solved these violations for our sample.

Answering Q1.2. To answer **Q1.2**, one author of this work manually analyzed the source code of at most five unmerged pull requests that have already been closed. Hence, developers will not merge it in the future as they currently are. Manually reviewing code contributions is a time-consuming task. However, we plan to extend this analysis in future work.

The corresponding results might bring evidence that we missed violations because the code contributions that would introduce them were 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 it without demanding changes to the submitter.

We search for violations of the constraints we defined for each project. Thus, while manually reviewing, we have in mind mainly where and how the code contribution deals with the set of sensitive information. The procedure to collect data to answer **Q1.2** for each project is as follows. First, we search for the five most commented pull requests that are closed and unmerged. Second, we read the last comments to check why developers rejected the pull request. Third, we search for occurrences of sensitive information on the resulting `diff`. Lastly, we decide whether those occurrences are violations.

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 amount of Java code lines that we read was reduced. Table 4 summarizes our manual analysis results. Despite the fact that we could not find a connection between privacy and security pull requests and Lientz et al[40] categorization, we use it to keep the pattern equal to Table 3.

As we discussed before, security-related pull requests are rare for our sample. This way, only two ScribeJava pull requests approach privacy and security properties. They implement changes that print access tokens on the console.

TABLE 4 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

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.

> 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 little amount 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 pull requests rejection were: (i) the 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.

4.2.4 | Answering Q2

Moreover, to assess whether Salvum reduces the effort (regarding project versions and line of code to review) to manually review these violations that Salvum detects, we consider **Q2**. Table 5 illustrates the results for `PA` (Project versions to analyze) and `SA` (Source lines of code to analyze) metrics. For instance, we needed to perform a manual code review for only three versions of the Gitblit project. Salvum did not detect violations regarding the other 46 versions. Therefore, in the worst case, a manual code reviewer would need to consider the 49 versions. This scenario could happen for cases that we need to analyze all the contributions submitted by a particular developer. Table 5 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 23 053. 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 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. In summary, we needed to consider 0.53% of code contribution lines for Gitblit, 0.08% for ScribeJava, and 4.45% for Crawler4J.

TABLE 5 Comparison of the number of revisions with and without Salvum

Project	PA Without Salvum	PA With Salvum	Percentage of Total	SA Without Salvum	SA With Salvum	Percentage of Total
Gitblit	49	3	6,12%	6.012	32	0,53%
Open Refine	10	0	0%	578	0	0%
Voldemort	52	0	0%	5.040	0	0%
ScribeJava	27	1	3,7%	4.496	4	0,08%
Solo	10	0	0%	952	0	0%
HikariCP	10	0	0%	216	0	0%
Apache Kafka	43	0	0%	3.402	0	0%
Teammates	10	0	0%	1.863	0	0%
Crawler4J	10	1	10%	494	22	4,45%
Total	221	5	-	23.053	58	-

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

TABLE 6 Total number of warnings and violations

We manually reviewed all the 221 project versions and their 23 053 lines. Indeed, we could not find additional violations. 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 core code, and consequently, no violation is introduced. On the other hand, our tool has the advantage to precisely show from and to the sensitive information flows. In this context, even an experienced reviewer would have to put lots of effort to find interprocedural dependencies. For instance, the task of identifying violations of sensitive information that flows throughout many methods of different classes.

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

At last, we acknowledge that we need further observational study to better understand how reviews are done focusing on security and privacy. Moreover, a controlled experiment would be useful to provide better estimates of effort savings. For instance, we could estimate the additional effort for developers to write constraints in Salvum and to use our tool. We plan to do it in future work.

4.3 | Assessing benchmark software projects

For this assessment, we consider five benchmark software projects used in previous study.[12,21] We selected the projects that are still available either in SourceForge[42] or GitHub. From a total of 36 projects, we discarded the unavailable ones and also those that we could not correctly build or identify a set of sensitive information.

Different from the nine projects in Section 4.2, the current ones are benchmarks. This way, other researchers have previously introduced a NV throughout their source code. Additionally, we define a set of constraints for them in a similar way of Section 4.2. Our goal here is to answer **Q1** considering the latest version of each project.

We select the following open-source and Java written software projects: Blojsom,[18] Personalblog,[19] GridSphere,[20] SnipSnap,[21] and Lutece.[22] Table 6 illustrates the NVW for each selected software project. These warnings represent potential violations to the policies and constraints written for each project. Blojsom and Personalblog are not pull-based development. Thus, there is no pull request to analyze. Additionally, GridSphere and SnipSnap were migrated from SVN[43] to Git.[35] Therefore, they do not have GitHub pull requests. In turn, Lutece is currently developed on GitHub, and it has pull requests. Although, this project does not have an organized code review process, we include it in our assessment.

Salvum warned 86 violations for these projects of which we confirmed 80. This way, we looked carefully to verify whether it is a violation because we could not report them since contributors do not develop these projects anymore. We did not confirm six reported warnings because they were a Java reflection usage, and JOANA does not support it.[33] Thus,

[18] https://sourceforge.net/projects/blojsom/
[19] https://sourceforge.net/projects/personalblog/
[20] https://github.com/brandt/GridSphere
[21] https://github.com/thinkberg/snipsnap
[22] https://github.com/lutece-platform/lutece-core

it generates a false-positive because JOANA conservatively establishes that there is a flow for every reflection usage.[16] Listing 7 shows the constraint C2 as an example of constraints we defined for Blojsom.

```
1        AtomAPIServlet {password},
2        XMLRPCServlet {password},
3        BloggerAPIHandler {password},
4        WeblogAPIHandler {password}
5        noflow Writeops
6        where Writeops = {Logger.error(),Logger.info()}
```

Listing 7: Constraint C2

Below, we show the output our tool provided for C2 in Listing 7. 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.[20] The majority of problems we found in Blojson is similar or related to leaking sensitive information through writing operations. Thus, we do not discuss them here because it would be repetitive.

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

Other tools have also found many privacy and security issues for Blojsom.[12,21] However, this project is not developed for a long time. Therefore, developers did not fix its problems. The other projects present similar problems. Thus, we omit in this work for brevity. Our results here are different from the study in Section 4.2 because we consider benchmark software projects. Thus, since there is no code review process, it is natural that many violations are introduced by code contributions.

Based on `NVW` and `NV` results shown in Table 6, we answer **Q1** stating that Salvum can detect proper violations of sensitive information in the context of benchmark software projects.

4.4 | Threats to validity

In this section, we discuss the threats to validity of our work. We separate the threats as: Construct, Conclusion, External, and Internal validity.[44]

Construct validity: In Section 4.2, we evaluate effort using approximations: 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 violations. Therefore, we 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. This assessment is an important future work in our context.

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.

Conclusion validity: 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 detected violations and tag the developer that introduced it. Thus, we could not select a 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.

External validity: 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 with different programming languages or proprietary software. However, we still could find real violations, and we could reduce the effort to manually review code contributions. 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[45-47] or Bayesian-based.[48] However, implementing policy languages using these alternative techniques could open an interesting avenue of research. We consider these alternatives as future work. Additionally, we cannot generalize our results for projects that allow untrustworthy developers to write Salvum constraints. This way, they could accept vulnerable code contributions.

In Section 4, we define a set of policies and constraints for our selected software projects. We cannot assure, necessarily, that we could enforce these policies and constraints for other systems. A number of them are specifically related to the domain and source code we want to analyze. However, we could reuse some policies across different projects. For example, Policy P1 in Section 4.2.2 states *User password, locality, access information, and permission cannot flow to code contributions*. We could consider P1 for other projects as long as they have user passwords, locality, access information, and permission. Certainly, we would still have to define constraints for P1, as we showed in Listing 6. Therefore, this difficulty to reuse policies and constraints is both a threat and a Salvum drawback.

Internal validity: 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 selected set of pull requests.

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 how 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. 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. 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 threat by establishing their Salvum constraints. In Section 4.3, only one author of this work manually decided whether a violation warning is a real problem. Ideally, at least two researchers should make these decisions. This way, we would reduce bias because there would be a discussion before determining a violation occurrence. In contrast, this issue does not happen in Section 4.2 because we report the violations on GitHub, directly to the project developers. Therefore, these developers decided whether we found a real violation.

Regarding performance for industry adoption, 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 execution. This delay happens even using our powerful hardware (Intel Xeon E7 with 64 gigabytes of RAM). It occurs mainly because the SDG[37] 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.

In Section 4.2, 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.

5 | RELATED WORK

In this section, we present related work. First, we discuss manual code review and some existing IFC analysis tools. Lastly, we approach related language-based information flow.

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)[27] provides a number of these guides. Therefore, 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 CWE.[19] 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 understand 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.

IFC analysis tools. TAJ[12] 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 it is also a WALA[49] 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 Section 3, and to for other technical domains than Java Web projects.

Another related static analysis tool is the Flowdroid,[11] 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,[50] 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.

Similar to 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[13] 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 Section 2. 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[46] 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 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.[48]

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 work because we would have to ultimately implement a new dynamic IFC analysis so that it could work for our scenarios.

Moreover, ANDROMEDA[21] statically detects flows wherein information returned by *Source* reaches a *Sink* without being adequately endorsed by a ϵdowngrader,ϵ that is, the information has not been validated. ANDROMEDA is capable of detecting typical information flow-related vulnerabilities,[27] 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[4] since the latter guarantees noninterference.[16,51] 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[52] 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 Section 2, we would have to add a WAFL generator for each framework our selected software projects use (Section 4.2.1). For example, Gitblit uses the Apache Wicket[53] 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.

Language-based information flow. There are languages designed to support IFC. Jif[10,54] is a security-typed programming language that extends Java with support for IFC. 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 (Section 3). Additionally, Jif tangles constraint code with program code, which might hinder code maintenance and understanding since according to the principle of Separation of Concerns[32] one should be able to implement and reason about each concern independently.

The Checker Framework[24] extends Java's type system to let software developers detect and prevent errors in their Java programs. It provides some checkers for different purposes, 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.[32]

Joe-E[26] 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.[23] However, to use Joe-E to solve the problems presented in Section 2, 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[55] 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.[15] 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 his exact location (eg, Informatics Center, UFPE, Recife). On the other hand, people that are not friends with Bob can only see part of this location information (eg, 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 Section 2, 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[14] 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 4.2.2. 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.[14]

Finally, Apel et al[56] investigate some shortcomings of object-oriented modifiers, such as `public` or `private`, that limits expressiveness of feature-oriented languages.[57] 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,[58] and `public` to make the variable globally available. However, this solution does not support Salvum's flexibility to define different kinds of policies.

6 | 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.

We evaluate our approach in two 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 benchmark software projects. In this way, our results indicate that we can find a higher NV for these kinds of projects.

As future work, we intend to extend this study in several directions. In this work, we only enforce constraints for code contributions. However, we could also specify policies and constraints for features.[59] We believe that this extension could identify harmful feature interactions or configurations. Also, 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. 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.[23]Nonetheless, when this support becomes more robust, we plan to use Salvum to investigate Android applications. Additionally, 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 improvement would probably decrease the time to analyze multiple system versions. Moreover, 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 potential bias in our results. Also, we could perform additional analysis to better understand the reasons why developers introduce violations. To achieve that, we intend to collect a significant amount of code review comments for each software project. Then, we could study the collected data and draw interesting conclusions.

Regarding Salvum specification, we could improve it in different ways. First, we could add a declassification feature. It allows the security level (eg, *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 to operations that save it in a database unless it is encrypted before. Inspired by AspectJ wildcards,[60] we could allow developers to 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. Besides that, developers should have the possibility to declare variables that could hold definitions used in more than one location. This improvement would reduce constraint code duplication. Furthermore, Salvum could support more general constraints. For instance, developers could write this constraint for MVC (Model-View-Controller) projects: `Model.Userpassword noflow View.Page.textArea`. It 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. 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

[23]https://preview.tinyurl.com/yauwyvsj

automatically identify the classes that implement `Logger` so that we could also detect violations for them without explicitly declaring their methods.

ACKNOWLEDGEMENTS

We thank colleagues of the SPG[24] for their feedback. We also acknowledge financial support from the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) under grant number 141590/2013-0. Finally, we thank the anonymous reviewers for their comments on our work.

ORCID

Rodrigo Andrade https://orcid.org/0000-0001-9051-5347

REFERENCES

1. Denning DE, Denning JP. Certification of programs for secure information flow. *Commun ACM*. 1977;20:504–513.
2. Saghafi S, Fisler K, Krishnamurthi S. Features and object capabilities: reconciling two visions of modularity. Paper presented at: Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development; 2012.
3. Meneely A, Srinivasan H, Musa A, Tejeda AR, Mokary M, Spates B. When a patch goes bad: exploring the properties of vulnerability-contributing commits. Paper presented at: Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement; 2013.
4. Snelting G. Understanding probabilistic software leaks. *Sci Comput Progr*. 2015;97, Part 1:122-126.
5. Livshits VB, Lam MS. *Finding Security Vulnerabilities in Java Applications with Static Analysis*. Baltimore, MD: USENIX Security; 2005.
6. Chess B, McGraw G. Static analysis for security. *IEEE Sec Privacy*. 2004;6:76-79.
7. McGraw G. Automated code review tools for security. *Computer*. 2008;41:108-111.
8. Tan DJ, Chua T-W, Thing VL. Securing android: a survey, taxonomy, and challenges. *ACM Comput Surv*. 2015;47:58:1-58:45.
9. JOANA - Java object-sensitive ANAlysis http://joana.ipd.kit.edu.
10. Myers A. C., Nystrom N., Zheng L., Zdancewic S.. Jif: Java information flow http://www.cs.cornell.edu/jif.
11. Arzt Steven, Rasthofer Siegfried, Fritz Christian, Bodden Eric, Bartel Alexandre, Klein Jacques, Le Traon Yves, Octeau Damien, McDaniel Patrick. FlowDroid. *ACM SIGPLAN Notices*. 2014;49(6):259–269.
12. Tripp O, Pistoia M, Fink JS, Sridharan M, Weisman O. TaJ: effective taint analysis of web applications. *ACM Sigplan Notices*. 2009;44:87-97.
13. Fuchs AP, Chaudhuri A, Foster JS. SCanDroid: automated security certification of android applications. Paper presented at: Proceedings of the : IEEE Symposium on Security and Privacy; 2010.
14. Johnson A, Waye L, Moore S, Chong S. Exploring and enforcing security guarantees via program dependence graphs. Paper presented at: Proceedings of the Conference on Programming Language Design and Implementation; 2015:291-302.
15. Yang J, Yessenov K, Solar-Lezama A. A language for automatically enforcing privacy policies. Paper presented at: Proceedings of the Symposium on Principles of Programming Languages; 2012.
16. Hammer C, Snelting G. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int J Inf Sec*. 2009;8:399-422.
17. Online Appendix. https://sites.google.com/site/spe2019appendix/.
18. Andrade R. Privacy and security constraints for code contributions (PhD thesis). Federal University of Pernambuco; 2018.
19. CWE - Common Weakness Enumeration https://cwe.mitre.org/.
20. OWASP Code review guide. https://www.owasp.org/images/2/2e/OWASP_Code_Review_Guide-V1_1.pdf.
21. Tripp O, Pistoia M, Cousot P, Cousot R, Guarnieri S. Andromeda: accurate and scalable security analysis of web applications. *Fundamental Approaches to Software Engineering*. Rome, Italy: Springer-Verlag; 2013.210–255.
22. Enck W, Octeau D, McDaniel PD, Chaudhuri S. A study of android application security. Paper presented at: Proceedings of the USENIX Security Symposium; 2011.
23. Saltzer JH. Protection and the control of information sharing in multics. *Commun ACM*. 1974;17:388-402.
24. Ernst MD., Just R, Millstein S., et al. Collaborative verification of information flow for a high-assurance app store. Paper presented at: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security; 2014:1092-1104.
25. SLF4J -Simple Logging Facade for Java. http://www.slf4j.org.
26. Mettler A, Wagner D, Close T. Joe-E: a security-oriented subset of Java. Paper presented at: Proceedings of the Network and Distributed System Security Symposium; 2010.
27. OWASP Open Web Application Security Project. https://owasp.org/.
28. Kästner C, Apel S, Kuhlemann M. Granularity in software product lines. Paper presented at: Proceedings of the International Conference on Software Engineering; 2008:311-320.
29. Hardt D. The OAuth 2.0 authorization framework; 2012.
30. Using pull requests. https://help.github.com/articles/using-pull-requests/.
31. Issues in Github. https://guides.github.com/features/issues/.

[24] http://www.cin.ufpe.br/spg

32. Parnas D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*. 1972;15(12):1053–1058.
33. Graf J, Hecker M, Mohr M. Using JOANA for information flow control in Java programs - a practical guide. Paper presented at: Proceedings of the Working Conference on Programming Languages; 2013.
34. Tichy WF. RCS - a system for version control. *Softw Pract Exp*. 1985;15:637-654.
35. Git https://git-scm.com.
36. JSON https://www.json.org/.
37. Horwitz S, Reps T, Binkley D. Interprocedural Slicing Using Dependence Graphs. *ACM Trans Progr Lang Syst*. 1990;12:26-60.
38. Biba K. J. *Integrity Considerations for Secure Computer Systems ESD-TR-76-372*. Bedford, USA: USAF Electronic Systems Division; 1977:76–372.
39. Basili V., Caldiera G., Rombach H. D.. The goal question metric approach. In: *Encyclopedia of Software Engineering*. Wiley, Hoboken, NJ 1994 (pp. 528–532).
40. Lientz PB, Swanson EB. *Software Maintenance Management*. Boston, USA: Addison-Wesley Longman Publishing Company Inc; 1980.
41. Gousios G, Storey MA, Bacchelli A. Work practices and challenges in pull-based development: the contributor's perspective. Paper presented at: Proceedings of the International Conference on Software Engineering; 2016:285-296.
42. SourceForge https://sourceforge.net/.
43. Apache Subversion https://subversion.apache.org.
44. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslen A. *Experimentation in Software Engineering*. New York, NY: Springer; 2012.
45. Rastogi V, Chen Y, Enck W. AppsPlayground: automatic security analysis of smartphone applications. Paper presented at: Proceedings of the Conference on Data and Application Security and Privacy; 2013:209-220.
46. Enck W, Gilbert P, Chun B, et al. TaintDroid: an information-flow tracking system for real-time privacy monitoring on smartphones. Paper presented at: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation; 2010.
47. Yan K, Yin H. DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. Paper presented at: Proceedings of theUSENIX Security Symposium; 2012:569-584.
48. Tripp O, Rubin J. A Bayesian approach to privacy enforcement in smartphones. Paper presented at: Proceedings of theUSENIX Security Symposium; 2014:175-190.
49. *WALA: The T.J. Watson Libraries for Analysis*. http://wala.sourceforge.net/.
50. Arzt S., Rasthofer S., Bodden E. *SuSi: A Tool for the Fully Automated Classification and Categorization of Android Sources and Sinks. TUD-CS-2013-0114*. Darmstadt, Germany: University of Darmstadt; 2013.
51. Graf J, Hecker M, Mohr M. Security policies and security models. Paper presented at: Proceedings of the IEEE Symposium on Security and Privacy; 1982.
52. Sridharan M, Artzi S, Pistoia M, Guarnierie S, Tripp O, Berg R. F4F: taint analysis of framework-based web applications. *ACM SIGPLAN Notices*. 2011;46:1053-1068.
53. Apache Wicket http://wicket.apache.org.
54. Myers AC. JFlow: practical mostly-static information flow control. Paper presented at: Proceedings of the ACM Symposium on Principles of Programming Languages; 1999.
55. Miller M. S., Samuel M., Laurie B., Awad I., Stay M.. Caja: safe active content in sanitized JavaScript. https://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.
56. Apel S, Kolesnikov SS, Liebig J, Kästner C, Kuhlemann M, Leich T. Access control in feature-oriented programming. *Sci Comput Progr*. 2010;77:174-187.
57. Prehofer C. Feature-oriented programming: a fresh look at objects. Paper presented at: Proceedings of the European Conference on Object-Oriented Programming; 1997.
58. Apel S, Kastner C, Lengauer C. FEATURE HOUSE: language-independent, automated software composition. Paper presented at: Proceedings of the International Conference on Software Engineering; 2009.
59. Kang K. C., Cohen S. G., Hess J. A., Novak W. E., Peterson A. S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study. SEI-90-TR-21*. Pittsburgh, USA: Software Engineering Institute, Carnegie Mellon University; 1990.
60. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. Paper presented at: Proceedings of the European Conference on Object-Oriented Programming; 2001:327-354.

How to cite this article: Andrade R, Borba P. Privacy and security constraints for code contributions. *Softw Pract Exper*. 2020;50:1905–1929. https://doi.org/10.1002/spe.2872