UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Lembit Valgma

Usable and Sound Static Analysis through its Integration into Automated and Interactive Workflows

Master's Thesis (30 ECTS)

Supervisor: Vesal Vojdani, PhD

Tartu 2018

Usable and Sound Static Analysis through its Integration into Automated and Interactive Workflows

Abstract:
Static analysis allows software developers to detect and fix many types of errors in code before it is submitted to a production environment. Despite the availability of sophisticated analysis techniques, many preventable bugs still cause security vulnerabilities that allow hackers to steal private information. Studies have shown that even though developers recognize the benefits of static analysis there are many practical usability problems preventing higher adoption rates.

The challenge is even greater with sound analyzers that could potentially verify the total absence of specific types of bugs, but at the cost of rejecting some correct programs. This thesis investigates the current situation of adopting static analyzers in the industry and proposes an approach of integrating an analysis into the IDE and build system. The seamless integration of both interactive and automated analysis may enable developers to adopt sound analysis tools.

A prototype implementation of that static analysis workflow for tainting analysis in IntelliJ and Gradle is presented. The integration proposed works well for tainting analysis used in the prototype, but many challenges remain to generalize this to more complex analyses. The prototype has enabled the exploration of different approaches to usability and is a useful first step in a larger project aimed at building a user-friendly sound static analysis framework.

Keywords: static analysis, taint analysis, usability, IntelliJ

CERCS: P175 Informatics, systems theory

Automatiseeritud ning interaktiivne tööprotsess korrektse staatilise analüüsi kasutajasõbralikkuse parandamiseks

Lühikokkuvõte:

Staatiline analüüs võimaldab tarkvara arendajal tuvastada koodis leiduvaid vigu ning neid parandada enne, kui see jõuab reaalsesse kasutusse. Hoolimata sellest, et tänaseks päevaks on teada mitmeid häid analüüsimeetodeid, põhjustavad ennetatavad tarkvara vead siiski katkestusi kriitiliste rakenduste töös ning võimaldavad kolmandatel isikutel ligipääsu privaatsetele andmetele. Kuigi arendajad on teadlikud staatilise analüüsi kasutamise eelistest, takistavad mitmed asjaolud siiski selliste vahendite laialdasemat kasutuselevõttu. Üheks peamiseks probleemiks on anaüüsi vahendite keerukas ning tüütu kasutatavus.

Veelgi suuremat vastuseisu kohtavad korrektse (*sound*) staatilise anaüüsi vahendid, mis lubaksid potentsiaalselt kontrollida teatud tüüpi vigade puudumist programmis. Nende suureks miinuseks on võimalus vigade (valesti) tuvastamiseks ka osades tegelikult korrektsetes programmides.

Käesolevas magistritöös uuritakse, mis viisil kasutatakse staatilise analüüsi vahendeid ettevõtetes ning pakutakse välja, kuidas oleks mõistlik integreerida analüüsi tarkvara arenduskeskkonda (IDE) ning tarkvara ehitust automatiseerivasse töövahendisse (*build tool*). Interaktiivse analüüsi ja automatiseeritud analüüsi tugev integreeritus võib olla oluline komponent, mis paneks arendajad neid töövahendeid kasutama.

Töö tulemusena valmis ka näidislahendus, mis integreerib lekke analüüsi (*taint analysis*) IntelliJ ja Gradle töövahenditesse. Välja pakutud lahendus on sobilik lekke analüüsi jaoks, aga selle üldistamine keerulisemate analüüsimeetodite jaoks jääb lahtiseks probleemiks. Näidislahenduse arendus andis võimaluse uurida erinevaid lähenemisi kasutatavusele ning on kasulikuks esimeseks sammuks suurema lõppeesmärgi poole, milleks on kasutajasõbraliku korrektse staatilise analüüsivahendi loomine.

Võtmesõnad:

staatiline analüüs, lekke analüüs, kasutatavus, IntelliJ

CERCS: P175 Informaatika, süsteemiteooria

Contents

1 Introduction

Static program analysis is a technique for reasoning about the behavior of computer programs without having to run them. First introduced in the seventies, it has been an active academic research area ever since. The main application of static analysis is optimizing code that compilers generate, but it can also be used for code verification and error finding. *Sound* static analysis makes the exciting promise of verifying that the program contains no errors of the kind that the analyzer can find (e.g., null reference, buffer overflow, and sensitive information leakage).

Since all non-trivial questions about computer programs are undecidable, actual analysis methods must make some approximations about the code. To achieve soundness these approximations are done conservatively, thus ensuring that when the tool claims to have found no errors in the program, the program, in fact, does not contain any. It also means that some of the errors that are reported by the analyzer can actually be due to the imprecision of the analysis. Such cases are known as false positives, and they are one of the key obstacles to using static analysis tools.

While many great and advanced static analysis techniques have been invented, the adoption of analysis tools by software developers has not been as great. In addition to false positives, static analysis tools also face the problems with the understandability of error messages, the speed of the analysis, integration into the development process and lack of collaboration support. Poor adoption rates mean that many preventable bugs still make it to deployed code and cause crashes or security vulnerabilities. This results in much more financial damages than preventing them would have cost. Therefore, usability is an important consideration when developing a static analysis tool that has been somewhat neglected.

This thesis contributes to a larger project whose ultimate goal is to create a sound static analysis tool integrated into IntelliJ IDEA. The final tool should provide advanced analysis techniques with simple to use interface that would allow developers who are not static analysis specialists to discover and fix bugs in their code. It should address the issues raised by usability studies and be well integrated into the development process. The envisioned ideal workflow is as follows.

1. The analyzer flags the problematic locations, either on-the-fly or at compile time, in the user's development interface.

2. The user interacts with the tool that guides him to understand and fix the problem. The fix can also be in the form of annotations that will help the tool verify the program and encode environmental assumptions under which the program is correct.

3. These annotations serve as a correctness *witness*. The witness is generated when the user has dealt with all warnings. It is a verification artifact that can be committed

to the software repository together with the code.

4. The witness is then checked by a certified witness checker, which can verify the code offline without any user interaction. This verification is done each time code is submitted to the repository.

The direct aim of this thesis is to integrate a sound analysis tool into an IDE and emulate the above workflow. This should lay the groundwork for understanding the IntelliJ plugin development process and allow us to identify the kind of features that are important for the analysis tool to include; in particular, the implementation should demonstrate how to avoid the most common usability pitfalls faced by static analysis tools.

1.1 Thesis outline and contribution

The thesis analyzes the most important features for useful and usable static analysis tool, proposes how to integrate static analyzer into IDE and build system and presents a prototype implementation of the static analysis workflow for tainting analysis in IntelliJ IDEA and Gradle. The key contribution of this thesis is the conclusion that a usable sound analysis framework needs to integrate both into the build system as well as the user's development environment. The integrated design of an automated component together with a user-friendly frontend will allow fixes and user annotations to be conveniently added to the codebase. The high false positive rate of sound analyzers may be more tolerable if the verification effort is recorded in a meaningful way, such as adding source code annotations, allowing the automated component to confirm the verification.

The remainder of the thesis is organized as follows.

Section 2 gives a literature review for static analyzer usability and adoption. It concludes with the extraction of important features for prototype development.

Section 3 describes the sample analysis chosen for the prototype, tainting analysis. It introduces SQL injection vulnerability that can be prevented using tainting analysis. The section concludes with the overview of actual analysis framework (The Checker Framework) used in the prototype.

Section 4 discusses how to integrate static analysis into IntelliJ IDEA and describes the prototype application.

Section 5 summarizes the thesis and discusses future work.

2 Usability of current static analysis tools

Although many static analysis tools are available and developers mostly recognize the benefits static analysis can provide, the consistent usage of these tools is not very frequent [13]. In this section, we review some of studies that address the questions of why developers don't use static analysis and what could be done to improve these tools. Based on reports from Google and Facebook about the adoption of static analysis into their development process, we note that the trend within industry is to abandon soundness. We therefore turn to research that attempts to improve usability without making that sacrifice. Finally, we synthesize important properties for the development of the prototype.

2.1 Why developers don't use static analysis tools

Johnson et al. [19] conducted interviews with professional software developers to investigate issues facing usage of static analyzers. The interview consisted of three parts. The first part, Questions and Short Response, included questions related to general usage, understanding and opinion of static analysis tools. The second part, Interactive Interview, included observations of a developer actually using popular static analysis tools, such as FindBugs [2] within Eclipse IDE [1]. In the final part of the interview, the developer was asked how the tools should be improved.

The most common complaint, which almost everybody raised, was **result understandability**. Users found that tools do not give enough information to assess what the problem is, why it is a problem and what should be done differently. They felt that error messages should be more descriptive and possibly include some examples of correct solutions. Most preferred to have automatic *quick fixes*, reasoning that if the analyzer can detect the mistake, it should also be able to fix it.

Almost three-quarters of respondents described having problems with reviewing the **tool output**. Mainly the possibly large number of warnings, which could contain a high percentage of false positives. Participants felt that the organization of the output and its presentation play a big role in the usability as well.

Even more participants said that **customization** of the tool is important. Configuring the tool to only perform a certain type of analysis or analyze only part of the project would reduce the number of warnings that the developer has to review at one time. Participants also wanted the option to temporarily suppress certain types of warnings, making the output easier to comprehend.

About half of the participants raised concern about lack of team **collaborative** features in the tools. This issue depends heavily on the type of company developer is working in and can be extremely important to large corporations adopting static analysis tools.

The final issue that was considered important by almost all participants was **workflow integration**. Three quarters thought that the way current tools integrate static analysis was lacking. This can be a complex problem to solve since development tools may vary. People using IDEs preferred to have analysis integrated within the IDE. Not all developers use IDE, though, and everybody marked that opening another tool for analysis was distracting and "painful".

2.2 What developers want and need from static analysis

Christakis and Bird [16] performed a similar study among Microsoft developers. Instead of an interview, they sent out a survey questionnaire. This allowed them to collect many more answers, they got 375 responses.

Results were quite similar to the Johnson et al. [19] survey. Most frequently marked issues preventing analysis adoption were wrong default rules, bad warning messages, and too many false positives. Complete results of problems are listed in Figure 1.

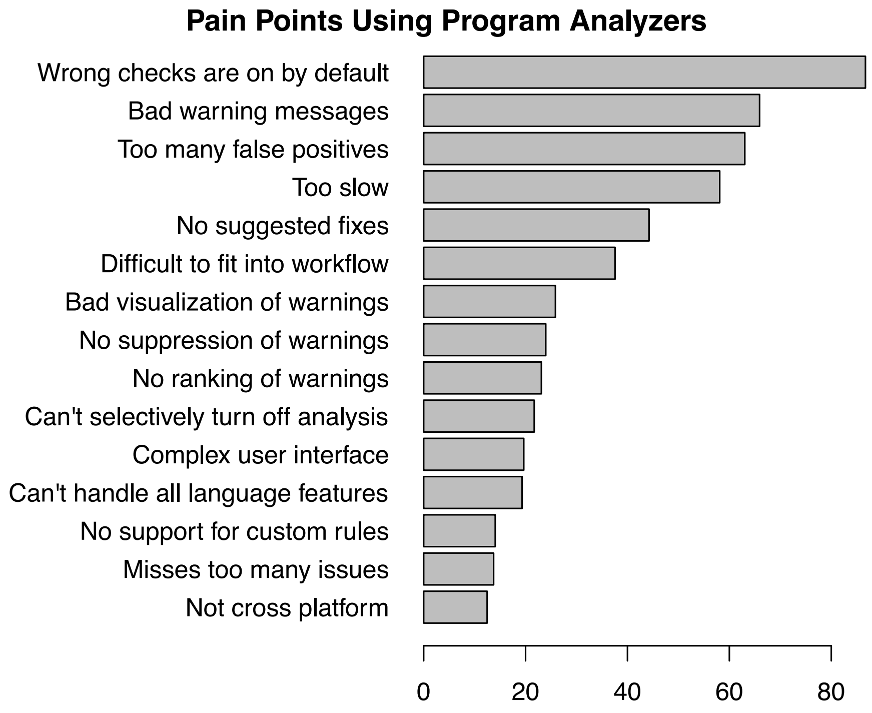


Figure 1. Pain points reported by developers when using program analyzers [16].

The most important code issues that developers would like detected by analysis tools were security and best practices. The lowest priority issues were power consumption and portability. Surprisingly, reliability was relatively low in the list, although reliability-related incidents were the most common actually occurring incidents in the company. The survey authors speculate that this implies developers don't actually trust code analyzers to detect more intricate reliability-related issues.

Analysis scope configuration was also marked as an important feature. Participants wanted to have the option to analyze only specific part of the project and also be able to analyze only the changed part of the code (when submitting new code to a repository).

Participants pointed out that a large number of false positives rapidly decreases trust in the analyzer (more than 15-20% was not acceptable) and that they accept some false negatives if it means reducing false positives.[1] Developers were, however, ready to add annotations and assumptions to their code if that would reduce the false positives number. Most were also willing to have analysis take more time if it leads to higher quality results. Additionally, they proposed a two-stage approach, with one stage running in real time/compile time, providing quick results in the development environment, and another stage running overnight, finding more intricate issues.

The preferred location for analysis reports was in the IDE. Developers want the tool to be seamlessly integrated into their workflow.

2.3 Current approaches

In this section we consider some of the examples from the industry where adoption of static analysis into their processes have been reasonably successful.

2.3.1 Google

Sadowski et al. [25] give an overview of Google's static analysis efforts. Google holds almost all of its codebase in one monolithic repository consisting of over two billion lines of code. Ownership of that code is divided between internal teams, and although everybody can propose changes to any part of the code, the owner has to approve all changes to his code. Before approving, all code goes through a testing and code review process. Code review is done through a centralized platform called Tricorder [26].

Google has had a relatively long history of attempting to adopt static analysis into its processes. The main tool they tried to integrate was FindBugs [2], but they eventually disbanded the effort due to a large number of false positives and difficulties integrating it into their development workflows. They found that developers tended to ignore all the warnings, mostly due to the presence of false positives.

Currently, Google has adopted a two-stage analysis process. They have integrated some of the more robust analyses into the compiler, emitting errors that prevent a successful build. That way the developer is forced to deal with issues before submitting code for review. To reduce developer complaints, before including an analysis into

[1] Just to be clear: this means that undetected security vulnerabilities are fine as long as one does not have to scroll through so many false alarms.

the compiler, the static analysis team first analyses the whole codebase itself and fixes all found errors. The analysis included in the compiler must also satisfy strict criteria: easily understandable and fixable, produce no effective false positives and concern only correctness of code (as opposed to best practices and style issues). This means that only relatively simple analyses can be included at this stage.

Code review is the second stage where Google includes static analysis. There more advanced analysis techniques can be applied, and the criteria for including an analysis are somewhat relaxed. Most importantly, it may produce up to 10% false positives. The analyses can also be customized for each project so that developers can include only relevant analysis types.

Google provides a framework to developers to add new analysis, which is based on the analysis of the abstract syntax tree of the Java source code. These analyses can and often do include *quick fixes*. Other developers can include the new analysis and provide feedback on the rate of false positives. Analyses producing more than 10% false positives are temporarily disabled until the submitter can fix it. Google has found that this kind of crowd-based analysis creation process produces good results.

Most of the analyses deployed in Google are of a relatively simple type. They have not used more sophisticated analysis techniques mainly due to the challenges of implementing them on a two-billion-line codebase. The financial cost of such implementation has been considered too high.

The most important insight from Google is that careful developer workflow integration is key for static analysis tool adoption. Building trust by not reporting false positives is also very important. More complicated analyses can be done during code review.

2.3.2 Facebook

Calcagno et al. [15] describe the static analysis process in Facebook. Compared to Google it has a smaller codebase, its main products being the Facebook website and its mobile application. There is still millions of lines of code to analyze. Facebook's development process is somewhat similar to Google. The programmer makes a change on the codebase, a *diff*, and then submits it to code review. During that phase, other developers can comment and suggest fixes. After being approved, the *diff* is applied internally and then used in production. Static analysis is performed in the code review process. The analyzer inserts comments on the lines of code where it detects a possible bug.

Facebook uses the static analyzer INFER [14] which supports compositional analysis. The analysis does not have to run on the whole project at once but can analyze each functional unit in isolation and compose the result. This makes it very suitable for analyzing the *diffs*, which can be done relatively quickly since the unchanged parts of the code need not be reanalyzed. Facebook aims for 10 minute feedback time. The compositional analysis also means that Facebook can utilize more complicated analysis compared to Google. They run the full codebase analysis each night and then new *diffs* can be analyzed quickly.

The authors mention the difficulty of getting developers to react to analyzer results and, similar to Google, suggest starting small with analyses that produce almost no false positives. That way developer trust is built and they will be ready to accept additional analysis. They specify four features that are extremely important for a static analyzer to have: full automation and integration, scalability, precision, and fast reporting.

2.4 Attempts to preserve soundness

While the industry seems to have taken a pragmatic (unsound) approach by accepting false negatives in order to reduce false positives, there are attempts to make sound analysis and verification more usable. Here we mention two that might be possible future directions for our analysis: *angelic verification* [17] and *verification modulo versions* [21].

2.4.1 Angelic verification

Software systems are rarely separate self-contained units. Their behavior usually depends on the environment that they interact with. This is particularly the case for open programs, such as libraries, plugins, and device drivers. These are expected to be used by a host system, but the host system can be configured in many different ways, so the ideal is to analyze only the module of interest based on how the environment may legitimately use it.

A precise static analysis, then, needs to model the environment. This can be done manually, but requires much effort and is prone to error. In order to preserve soundness, analysis techniques often make conservative assumptions about the environment. It means they analyze the worst case scenario (*demonic verification*) and provide guarantees under any conceivable use (and abuse) of the module. This typically leads to many false positives because valid environmental assumptions are ignored.

The *angelic verifications* approach instead provides the user a way to specify which environmental assumptions that may or may not hold, and only reports an error when no acceptable specification exists. For example, it would be reasonable to assume that calls to two different library methods will not return aliased pointers (pointers to the same memory location). In the demonic approach, the analyzer assumes that all unknown pointers may alias, so it will fail to prove many valid safety properties about the program if that assumption is valid. The angelic verifier, in contrast, would try to find the set of least restrictive assumptions under which verification succeeds, so rather than producing a conservative warning, it would suggest that the program is correct if the aliasing assumption is true.

2.4.2 Verification modulo versions

Static analyzers usually try to analyze the entire piece of code it is directed to (e.g., the project). In a standard software development process, the code is added regularly (*diffs*) to an existing repository, and the developer is more interested in whether any new problems were added than the correctness of the entire codebase. There might be valid reasons the old code includes some parts that analyzer thinks are errors, and the warnings from those issues are simply noise and distraction for the developer who added the new piece of code. Thus the goal is to allow the suppression of the previous warnings and only display new ones. Some existing analyzers allow that by relatively simple syntactic matching (e.g. if there was an error on line 176 in the original code, then the error found in the modified code on line 176 is not shown). This approach is not sound because the meaning of a statement at a given code location may depend on changes elsewhere in the code.

Logozzo et al. [21] propose a sound approach. First, they try to extract environmental assumptions under which the original program is correct (analyzer would report no errors). These assumptions can be either sufficient (every program execution ends in a good state) or necessary (it holds whenever the program reaches a good state). Next, they insert the extracted assumptions into the modified code and attempt to verify the modified code under the assumptions made by the original.

The conclusion one may draw from such differential verification depends on whether one infers sufficient or necessary conditions. If the initial assumptions are sufficient for the correctness of the original, and an error is found in the modified code, a regression was introduced into the program by the new code. That is, the modified code requires stricter assumptions for its correctness than the original code. If the assumptions are necessary for the correctness of the original code, and these assumptions allow us to prove the absence of errors in the modified code, then the modified code is proven correct relative to the original. That is, the modified code is correct under assumptions the original code already required for its correctness.

Their proposed framework is relatively general and thus allows the use of many existing analyzers. However, the theoretical solution requires mapping of old and new program points which is in practice unfeasible. Thus, they suggest using matching of call conditions (method names are assumed to be the same in modified code). They also implemented their approach using necessary assumptions for the static analyzer `cccheck` [18] and analyzed a couple of projects. In general, they found approximately 50% fewer warnings with their approach. In some of the cases, almost all warnings were removed (meaning all were false positives).

2.5 Requirements for prototype

We conclude from the literature that the most important features for a static analyzer to be adopted by developers are trust in its usefulness and integration into standard development workflows. As this thesis aims to address usability issues for *sound* static analysis, the conclusion of the industry surveys about false positives eroding user trust is sobering. There is, at least, a potential willingness to annotate programs. Developers may be less reluctant to deal with false positives if doing so was not perceived as a complete waste of time. It is therefore critical that the verification effort generates some form of proof artifacts, such as meaningful code annotations, which can be used in subsequent verification efforts of other developers.

For workflow integration, the most desired feedback location for running the analysis was the IDE, which aligns well with the workflow envisioned in the introduction. Having the analyzer in the IDE ensures that bugs are discovered early in the process when they are easier to fix. Displaying analysis results conveniently also allows us to best address error message understandability, so the user clearly understands what is wrong. An important feature that Google found to be helpful for adopting analyzers is *quick fixes*. Having a tool to suggest or apply automatic fix made developers to use analysis tools more. This also helps the user understand the problem in the first place.

It is also important to integrate analysis with the build systems. Almost any bigger project uses some kind of build system (Gradle, Maven etc.) to manage dependencies. This means our application should also support execution via the chosen build system. This would allow the analysis to be run automatically on continuous integration servers with each commit to the repository.

3 Tainting analysis

Managing information flow is a core part of programming. Tainting analysis is a concept that divides variables into trusted (untainted) and not trusted (tainted) categories. No tainted variable should be used where untainted is required. It can be viewed as a simplistic approximation of non-interference in the more general concept of secure information flow [24], which requires high security (input) values do not affect low security (output) values. Tainting analysis can be performed in several ways. Data flow analysis builds a constraint system for variables based on the program's control flow graph (CFG) and checks if all constraints are satisfied. Type checking assigns each variable as tainted or untainted and checks for type correctness.

Tainting analysis can help detect SQL injection and secure information leakage vulnerabilities among others. For SQL injection analysis, the tainted user input must not reach untainted SQL query executions. For information leakage analysis the tainted secure information must not reach untainted system output, such as a non-authenticated user or webpage.

3.1 SQL injection

SQL injection [7] is a security vulnerability in database-driven applications where the user is able to modify the query that the database runs. A simple example of it would be a form that asks for the user's name, queries the database with that name and replies with data about that name [20]. The relevant backend code in Java might look like the code shown in Figure 2. The user is asked for their name and then the result is simply concatenated to the SQL query (line 5).

```
1 ...
2 conn = DriverManager.getConnection(url + dbName, user, passwd);
3 String user = request.getParameter("user");  // tainted source
4 Statement st = conn.createStatement();
5 String query = "SELECT * FROM  User where userId='" + user + "'";
6 ResultSet res = st.executeQuery(query);  // problematic execution
7 ...
```

Figure 2. Code example that is vulnerable to SQL injection [20].

Clearly, the developer's intention was to only display information about a single person. Since user input is not validated, however, the user could provide malicious input like `"xx' or '1'='1"`, which when concatenated to the original query, would make the query condition always true and return all the rows from the database. Wikipedia [12] lists tens of high profile SQL injection cases until very recently and OWASP [8] lists injection as the topmost security vulnerability.

3.1.1 Prevention

Preventing this type of attack can be done through input validation. The user-provided input should not include quotes that are not escaped nor SQL keywords. Doing the validation manually can be error-prone and thus the recommended solution is to use the parameterized query API [7]. The proper way to fix the vulnerability in our example is shown in Figure 3. The user-provided parameters are marked in the prepared query by `"?"` and added afterward using safe methods.

```
1  ...
2  conn = DriverManager.getConnection(url + dbName, user, passwd);
3  String user = request.getParameter("user");  // tainted source
4  String query = "SELECT * FROM  User WHERE userId=?";
5  PreparedStatement  ps = conn.prepareStatement(query);
6  ps.setString(1, user);  // perform validation
7  ResultSet res = ps.executeQuery();  // safe execution
8  ...
```

Figure 3. Code example where SQL injection vulnerability has been removed [20].

If the user provide now the same problematic input `"xx' or '1'='1"`, the `setString` method on line 6 would transform it to `"xx\' or \'1\'=\'1"`, which would probably result in query returning no rows.

3.1.2 Using tainting analysis for detection

Detecting SQL injection vulnerability is a relatively straightforward application of tainting analysis. All the user-provided data should be treated as tainted sources. All database query executions should be marked as untainted sinks. The validation methods should change tainted data to untainted (for the prototype implementation, the correctness of the validation functions remain the responsibility of the developer). In such a setup, tainting analysis finds all direct user input flows to query execution that has not been validated. In the example in Figure 2 tainting analysis gives a warning on row 6 (if `Statement.executeQuery()` has been marked as an untainted sink).

3.2 The Checker Framework

The Checker Framework [23, 10] enhances Java's type system by adding a pluggable type system in a backward-compatible way. This allows developers to detect and prevent errors in their Java programs. The framework includes new Java syntax for expressing type qualifiers in the form of annotations, which was adopted into Java 8 [6], provides a mechanism for writing type-checking rules and supports flow-sensitive local type qualifier inference. It includes several implemented type checkers (e.g., nullness, interning, tainting and many more).

Type qualifiers extend a type with a particular optional attribute. For example `@Untainted` is a type qualifier which denotes that the value is untainted (trusted). This can be added to the type `String` resulting in the type `@Untainted String`. The type checker now notifies us if we try to assign some `@Tainted` value to an untainted variable.

Using the Checker Framework is relatively simple [11]. The annotations library should be included and source code annotated using relevant type qualifiers. When compiling, the corresponding checker should be included in the Java compiler call using the `-processor` argument (e.g., `-processor TaintingChecker`). Then the specified type checker verifies the compiled code. If the developer does not use the `-processor` argument, the code is compiled as regular Java code by ignoring the annotations. The user can suppress warnings either by adding annotations to a specific location in the source code or by specifying compile parameters. In addition to using predefined types and checkers, the developer can define their own and leverage the existing type checking framework.

3.2.1 Comparison to other tools

Since the Checker Framework is type-checking tool, it differs from many of the bug detectors mentioned (like FindBugs). The main differences are:

- Type checking is sound, meaning it finds all errors of a specific type, so it can verify the absence of errors. A bug detector tries to find as many errors it can, but cannot usually guarantee that there aren't any left.
- Type checkers requires annotations for type qualifiers. Some bug detectors are fully automatic and require no user input.
- Type checkers may use more sophisticated and complete analysis. A bug detector typically does a more lightweight analysis, coupled with heuristics to suppress false positives

Studies have shown [23] that type checking can find more bugs than many other bug finding tools.

3.2.2 Tainting Checker

The tainting checker has been predefined in the framework. It allows one to use the following annotations:

- `@Untainted` indicates a type that includes only untainted values.

- `@Tainted` indicates a type that may include tainted or untainted values. It is the default qualifier.
- `@PolyTainted` is a qualifier that is polymorphic over tainting. This enables context-sensitive analysis; i.e., a method that is polymorphic can return a tainted or untainted value depending on its input parameter.

Static tainting analysis needs to make sure that information from tainted sources does not reach untainted sinks. Thus, the checker does not ascribe specific semantics to the tainting types. Since it is allowed to assign untainted values to a tainted variable, `@Untainted` is subclass of `Tainted`. Then, tainted values flowing to untainted variables will violate the subtyping rules.

The Checker Framework performs flow-sensitive intraprocedural qualifier inference. For example, this allows the method in Figure 4, which should return an untainted value, to be successfully type checked because the value assigned to the variable `result` will be inferred to be untainted, even though the variable itself could also store tainted values.

```java
@Untainted
public String inference(@Untainted String bar) {
  @Tainted String result = bar;
  return result;
}
```

Figure 4. Example of type qualifier inference.

Inference allows developers to only annotate the important parts of the code, like untainted sinks and validation methods. Local variables mostly do not need special annotations and can be viewed by default as potentially tainted. Inference may need additional annotations when working with aggregated data structures, such as a map from strings to arrays of untainted integer values.

When analyzing code for injection vulnerabilities, the user input should be marked as tainted (untrusted) and input parameters for code execution methods (like `Statement.execute`) should be marked as untainted. Input validation functions have to use tainting warning suppression annotations (`@SuppressWarnings("tainting")`).

4 Plugin development

This section presents the prototype that implements the suggested static analysis workflow. We first consider the scope of the problem and requirements on the prototype. Then, we turn to application usage, code design, and also a short description of the plugin development process in IntelliJ IDEA.

4.1 Problem scope

The aim of the prototype was to implement an example a static analysis tool in IntelliJ IDEA while addressing the most important problems facing the adoption of such tools. We chose to implement tainting analysis with the focus on SQL injection, described more thoroughly in the previous section. It was chosen because it is a relatively simple and easily understandable analysis type that allows checking of some of the most widespread and important security vulnerabilities. The prototype is limited to the Java programming language and uses the Checker Framework as backend analysis framework.

Although the focus is on SQL injection, we also support general tainting analysis to detect any flow from a tainted source to an untainted sink.

4.2 Solution choices

The inspiration for the prototype was the ASIDE analyzer [28] that implemented interactive tainting analysis in Eclipse [1]. It analyzed code for both input validation and access control and prompted the user to specifically annotate lines that perform those tasks. Based on those assumptions, the tool could verify the rest. The analyzer has not become very popular and the project seems inactive. One of the problems with IDE integration is that the tool is constrained inside Eclipse IDE. The annotations were separate from the code and had no meaning outside the IDE, so the analysis could only be performed inside IDE. We aim to address that problem by including annotations in the code and supporting verification via build tools.

4.2.1 Requirements

Based on the analysis in the previous chapter the following requirements for the prototype were set:

1. Perform sound analysis of SQL injection vulnerability in Java source code.
2. Provide user with clear and understandable information about the detected vulnerabilities.
3. Provide automated help to create fixes for the detected problems.

4. Be interactive, integrated into IntelliJ IDEA.
5. Analysis should result in very few false positives.
6. Good compatibility with various build tools and option to turn analysis off.

4.2.2 Tool selection

It was decided to integrate the prototype in the IDE in the form of a plugin for IntelliJ IDEA [4], a choice based mostly on author and supervisor personal preference, but also on the fact that it has been gaining a lot of popularity in recent years. Some studies show it to be already most popular Java IDE [27].

The Checker Framework [10] was chosen as the analysis tool because it allows enhancing the Java type system with additional types by using annotations. Type checking is done during the compile process by using the Checker Framework checker as an annotation processor for the *javac* compiler. This allows for relatively large usage configuration flexibility, the annotated code can be compiled without checking for enhanced types by simply not providing annotation processor. The *javac* compiler plugin solution also means that the Checker Framework can be easily integrated into various build scripts and other external tools [11], which is an important feature when considering tool usability.

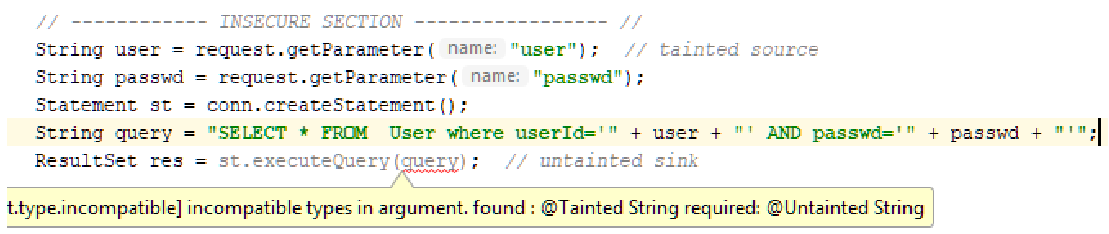
4.3 Solution workflow description

Setup. The user should first download the Checker Framework and set up the environmental variables for his system. Then, add the framework support to his build process according to the manual [11]. The process was developed and tested using Gradle, but both Ant and Maven should behave similarly. It is important to note that only the tainting checker should be declared as an annotator processor. We also provide a modified Gradle script in our repository.

After this, the project should be imported to IntelliJ and Gradle takes care of the library dependencies. If no build tool is used, `checker.jar` library should be added to the relevant modules (to allow using annotations).

Annotating source code. The main logic in the analysis is that unvalidated user input should not reach database query execution methods. This means marking input as tainted and query execution parameters as untainted. By default, all variables and methods are treated as tainted, except for `java.sql.Statement` methods, whose input parameters are already annotated as untainted. The user should also annotate any other methods that he considers as untainted sinks. This can be done either directly in the source code or within the `main.astubs` file in the project directory. In the latter case, only method signatures need to be annotated.

Running the analysis. The analysis can be started by selecting `Run Tainting Analysis` from the toolbar menu. It compiles the project adding the tainting checker as an annotation processor to the Java compiler. This also means that any other settings set up for the build process are respected. The user sees the found problems in the Message window, but more importantly, issues are highlighted in the editor as shown in Figure 5.

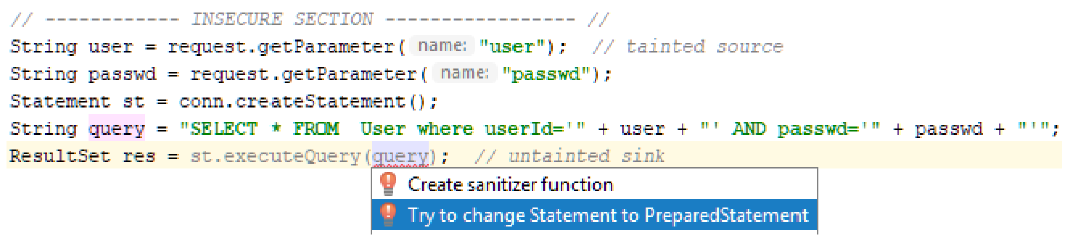


```
// ------------ INSECURE SECTION ----------------- //
String user = request.getParameter( name: "user");  // tainted source
String passwd = request.getParameter( name: "passwd");
Statement st = conn.createStatement();
String query = "SELECT * FROM  User where userId='" + user + "' AND passwd='" + passwd + "'";
ResultSet res = st.executeQuery(query);  // untainted sink
```

```
t.type.incompatible] incompatible types in argument. found : @Tainted String required: @Untainted String
```

Figure 5. Found errors are highlighted in the editor.

Fixing the errors. The plugin provides assistance to the user for fixing found errors. A specific fix is targeted for SQL injection vulnerability, which is shown in Figures 2 and 3. By invoking intention actions on the highlighted expression, the user is prompted to select a quick fix, as can be seen in Figure 6.



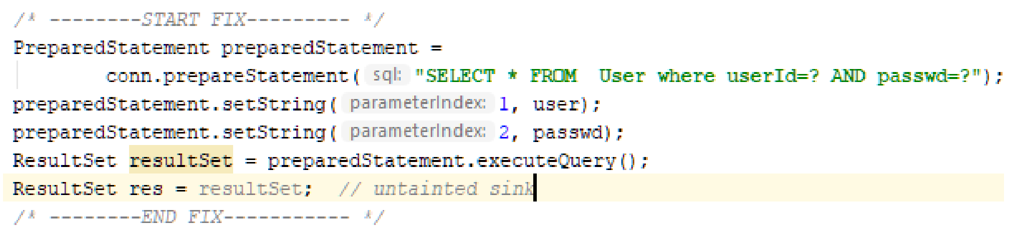
```
// ------------ INSECURE SECTION ----------------- //
String user = request.getParameter( name: "user");  // tainted source
String passwd = request.getParameter( name: "passwd");
Statement st = conn.createStatement();
String query = "SELECT * FROM  User where userId='" + user + "' AND passwd='" + passwd + "'";
ResultSet res = st.executeQuery(query);  // untainted sink
```

Create sanitizer function
Try to change Statement to PreparedStatement

Figure 6. Quick fix intention selection

The automatic fix tries to find the concatenated query and transform it into a `PreparedStatement`. The result of this transformation is shown in Figure 7.

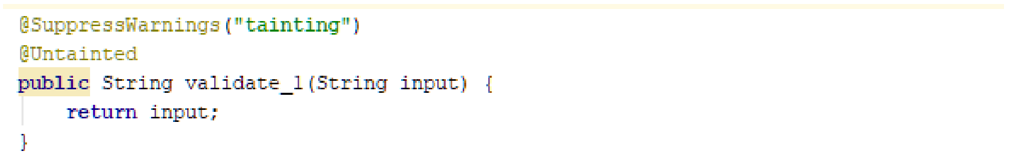
The general use case. The automatic fix offered for the general use case is always similar: the tainted input should be validated. The problematic expression should be wrapped inside a validation function that takes in the tainted value, outputs an untainted value and suppresses warnings. The correctness of the validation function is the user's responsibility, as it can be highly application specific. The IDE supports this process by



```
/* --------START FIX--------- */
PreparedStatement preparedStatement =
        conn.prepareStatement( sql: "SELECT * FROM  User where userId=? AND passwd=?");
preparedStatement.setString( parameterIndex: 1, user);
preparedStatement.setString( parameterIndex: 2, passwd);
ResultSet resultSet = preparedStatement.executeQuery();
ResultSet res = resultSet;  // untainted sink
/* --------END FIX----------- */
```

Figure 7. Unsafe string concatenation automatically converted to `PreparedStatement`.

automatically creating a new method with the required annotated type signature, but the user needs to implement it. The empty annotated template is shown in Figure 8.



```
@SuppressWarnings("tainting")
@Untainted
public String validate_1(String input) {
    return input;
}
```

Figure 8. Annotated validation function template.

Once the user has fixed all highlighted errors, he should run the analysis again.

4.4 IntelliJ plugin development

IntelliJ IDEA Community Edition [3] is the free and open source version of IntelliJ Ultimate IDE, which can be used to develop plugins. It provides the developer access to many powerful tools used in the IDE and a relatively simple process to integrate new functionality into IntelliJ. There is an effort by the company to provide documentation [5] and it does have the necessary information to get started; however, many of the topics are missing or not fully covered. In order to find how to do something not specified in the guide, one usually need to read the source code, which is at least well commented itself. Finding the correct class to achieve the desired task can be a considerable challenge, and much of the author's time was consumed by searching the code. Below we introduce some of the tools and concepts that are important for plugin development.

4.4.1 Components and actions

Plugin components are the main concepts of plugin integration. A component is basically a class that gets loaded when the user starts the plugin and can include a separate interface. The developer can define application-, project- or module-level components—the level defines for which scope a new component is created. There is also a concept of a service; it is a component that is loaded on demand. The developer can create completely new components or extend the functionality of existing ones (through extensions). Extensions allow for a simple and integrated way to leverage IntelliJ's own functionality (e.g., annotator, inspection and intention action).

Another important plugin concept is an action. This allows one to define what happens when a menu item or a toolbar button is selected. The developer can define new actions or extend existing ones.

All of the component, action and plugin information needs to be declared in the `plugin.xml` file. The interface and implementation class names are also in that file. The developer is allowed to use either Java or Kotlin to implement the classes.

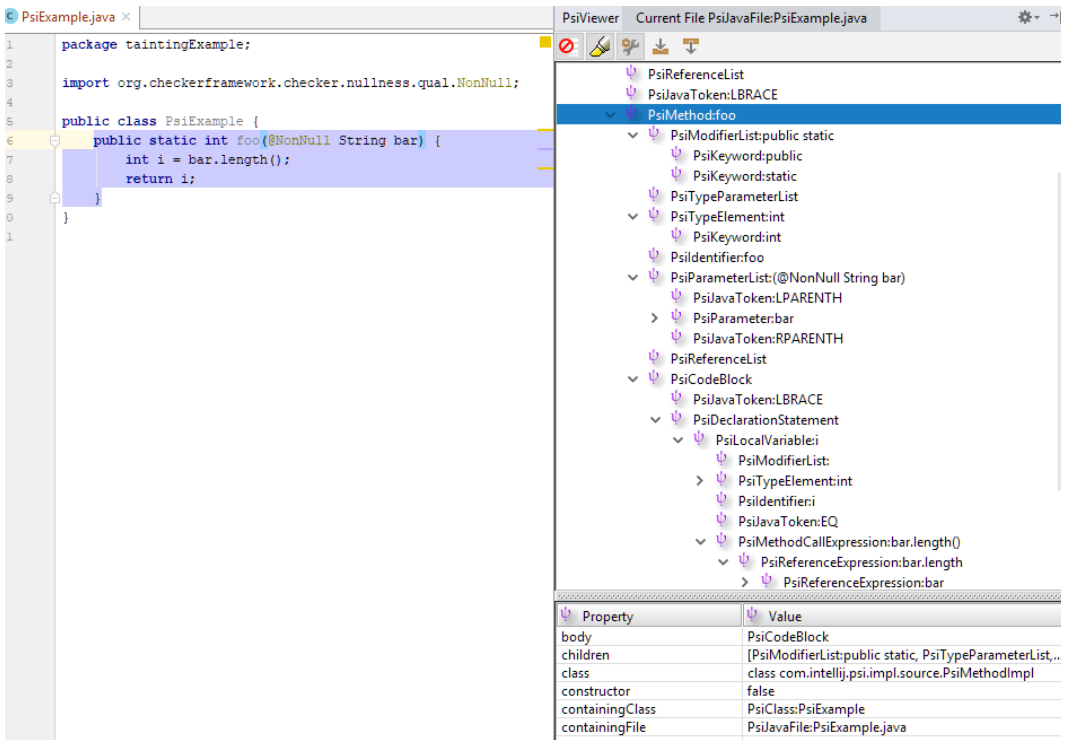
4.4.2 PSI

The Program Structure Interface, commonly referred to as just PSI, is the layer in the IntelliJ Platform that is responsible for parsing files and creating the syntactic and semantic code model that powers so many of the platform's features. For each code file, IntelliJ builds a PSI element tree with PSI file being the root. The PSI tree is similar to the abstract syntax tree (AST); however, it contains more information related to context, location, and references of the element. Navigating and manipulating the PSI are the main ways to perform code analysis and transformations. The PSI tree can be navigated as a regular tree, but there are many helper functions that provide shorter solutions. A great tool to explore and discover PSI trees is the *PsiViewer* plugin [9], which is shown in Figure 9.

4.4.3 Threading

IntelliJ is an advanced IDE with many analyses and processes supporting the user. To accomplish that without freezing the UI, it uses background processes—many running in different threads. Due to that, modifying the PSI must always be done within the UI thread, using a write action (which locks the modifiable part). Reading the PSI from non-UI threads also requires using a separate read action. The developer should also be careful not to allow a background process to read action to take too long or it might affect the user experience.

Another effect of highly threaded processes is that sometimes it might be difficult to ensure the order of actions, especially when new threads are spawned by the called methods. IntelliJ sometimes offers callbacks to remedy such situations. A good example of this in the current plugin is the quick fix intention action, where we have to be sure that the project building process has been completed before we can collect the output warnings.


PsiExample.java

```
package taintingExample;

import org.checkerframework.checker.nullness.qual.NonNull;

public class PsiExample {
    public static int foo(@NonNull String bar) {
        int i = bar.length();
        return i;
    }
}
```

PsiViewer Current File PsiJavaFile:PsiExample.java

```
PsiReferenceList
PsiJavaToken:LBRACE
PsiMethod:foo
  PsiModifierList:public static
    PsiKeyword:public
    PsiKeyword:static
  PsiTypeParameterList
  PsiTypeElement:int
    PsiKeyword:int
  PsiIdentifier:foo
  PsiParameterList:(@NonNull String bar)
    PsiJavaToken:LPARENTH
    PsiParameter:bar
    PsiJavaToken:RPARENTH
  PsiReferenceList
  PsiCodeBlock
    PsiJavaToken:LBRACE
    PsiDeclarationStatement
      PsiLocalVariable:i
        PsiModifierList:
        PsiTypeElement:int
        PsiIdentifier:i
        PsiJavaToken:EQ
        PsiMethodCallExpression:bar.length()
          PsiReferenceExpression:bar.length
            PsiReferenceExpression:bar
```

Property	Value
body	PsiCodeBlock
children	[PsiModifierList:public static, PsiTypeParameterList,..
class	class com.intellij.psi.impl.source.PsiMethodImpl
constructor	false
containingClass	PsiClass:PsiExample
containingFile	PsiJavaFile:PsiExample.java

Figure 9. PsiViewer example.

4.4.4 Previous Checker Framework plugin

There have been previous efforts to integrate the Checker Framework into IntelliJ. In 2014, Daniil Ovchinnikov wrote a plugin that adds the Checker Framework support to IntelliJ [22]. It reported the errors found by the Checker Framework as inspection results and did not offer any quick fixes. It did support all the different checkers at that time.

The plugin worked only for IntelliJ 13.X and the author has not added support to new IntelliJ versions (nor has he any plans to do so). He also has no documentation about the plugin. His goal was to analyze code on-the-fly, but that turned out to be too slow due to the complexity of the Checker Framework. Eventually, he abandoned the project but stated that the most reasonable solution would be to run the Checker Framework on the project and collect the output it provided. Since our goal was not on-the-fly analysis, and compile time has been seen as the most reasonable place to run a static analysis, we decided to follow his advice and did not try to enhance the old plugin but write a new one. Initially, we only need to support one checker, which also makes the additional complexity unnecessary. Simply collecting the Checker Framework output also makes it compatible with various build tools and platforms, which was an important design choice for us.

4.4.5 Integrating static analyzer

IntelliJ offers two good ways to integrate new (static) analyses into the IDE: annotator and inspection. The annotator analyses the currently opened file in the editor and can highlight desired sections. It also enables the adding of error messages and quick fixes. Annotator is run automatically when a file is opened and upon file changes. Inspections partially duplicate the annotator functionality, but they also support batch analysis of code, the possibility to turn inspection off and the configuring of inspection options. Running an inspection also can open a separate window where the result is shown. Additionally, some of the analysis tool integrations (e.g., FindBugs) have opted to create their own tool windows inside IntelliJ to configure options and display results. This is the most flexible option but also requires the most effort to develop.

The choice between annotator, inspection and separate tool window should be done based on the required features, while also considering that the annotator offers the best integration into the IDE and the tool window the least. Since the prototype has very few configurable options and the goals is to achieve maximal integration (as using separate perspective for analysis was marked as big negative in surveys), we opted to use an annotator.

4.5 Developed plugin

The developed plugin [2] includes an action for starting the analysis, a service that collects and parses the Checker Framework output, an annotator that highlights the problems and intention actions that apply the quick fix. The general process flow is shown in Figure 10.

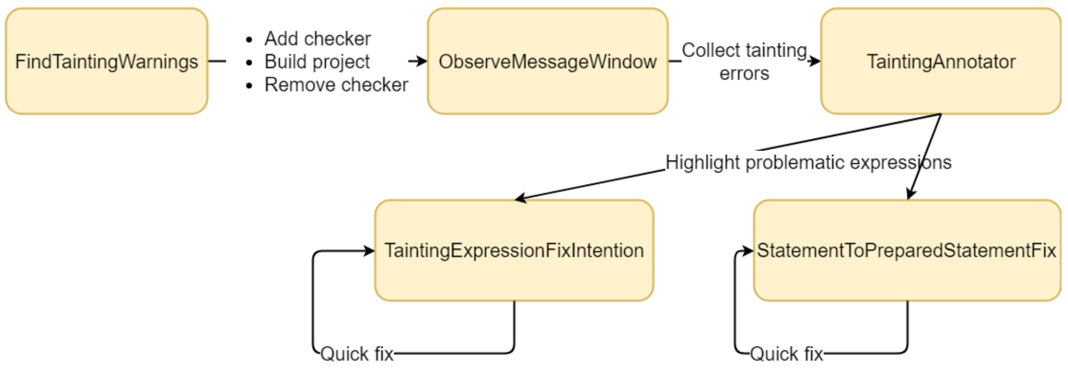


Figure 10. Prototype general process flow design.

[2] https://bitbucket.org/valgma/taintingplugin/src/master/

4.5.1 `FindTaintingWarnings` action

The `update` method is called every few seconds in the IDE. Its main function should be to define when an action is available and should not contain code that requires many resources. We have defined an action to be available when the project is opened.

The `actionPerformed` method defines what happens when an action is executed. For our action we need to add the Checker Framework annotator processor to the build configuration, then compile the project and afterward call our defined service that collects the Checker Framework warnings and errors. Afterward, we need to remove the annotator processor again to allow normal build process without the Checker Framework.

4.5.2 `ObserveMessageWindow` service

Since the analysis is started by an action, the component can be started on demand and is thus declared as a service. It is very straightforward to change it to a regular component in the future if there should be a need.

The service provides `updateWarnings` method to collect the Checker Framework warnings from the output and a getter to access the current warning list. Since there can be more warnings beside tainting, each warning is validated before being included in the list. The benefit of using IntelliJ output messages is that they give the exact location of the error, which allows the plugin to pinpoint the problematic expression.

4.5.3 `TaintingAnnotator`

The annotator extension point allows the highlighting of areas in the open editor window. The `annotate` method is run for each PSI element and can be used to create new annotations and quick fixes. Our annotator needs to check if the current element is listed as problematic by the service. If it is, a new error annotation is created with a possible quick fix.

4.5.4 `TaintingExpressionFixIntention`

Intention actions are invoked in the editor with `Ctrl + Enter`. Since our intention is registered in the annotator, it is only available after the annotator has marked the expression as problematic.

A new validation method stub is created and the expression is wrapped inside. The stub accepts tainted input parameters provides untainted output value (same type as input) and also suppresses tainting warnings. It should be noted that any PSI modification should be triggered from inside write action, but since we are using `BaseIntentionAction` the `invoke` method is automatically wrapped inside a write action.

4.5.5 StatementToPreparedStatementFix

This class realizes the fix described in Figure 3.

The task for this intention is to detect if problematic expression is inside `Statement.executeQuery()` call and the query string is created by concatenation (e.g. `"SELECT * FROM User where userId='" + user + "'"`). Other ways of building the query, like `StringBuilder`, are not supported.

If the above conditions are met, the query extracts the `Connection` parameter and the concatenated variables inside the query string. Then a new `PreparedStatement` is built where variables are replaced by `"?"` (e.g. `"SELECT * FROM User where userId=?"`) and then variables are added with `setString` method. Finally, the original statement is replaced.

4.6 Testing

Due to the nature of the plugin testing was mostly done empirically during development. The plugin was tested with IntelliJ 2018 on Ubuntu 16.04, OSX 10.13 and Windows 10. Windows has some output formatting problems with the Checker Framework, but it does not affect the plugin. One of the problems during development and testing was the fact that when starting the plugin via the Gradle task, opening new projects often seemed to require specifying the project JDK as IntelliJ couldn't properly detect the existing one. It is probably due to Gradle not managing to solve the dependency properly. That behavior was also inconsistent for Ubuntu and Windows, where Windows seemed to require specification each time but Ubuntu only sometimes.

IntelliJ offers automatic plugin testing system where the user needs to provide input and expected output files. The testing system then executes the plugin and checks if the provided output is equal to the expected. This system can be useful in the future but is too heavy-handed for the present needs of the plugin.

4.7 Reflection

Although IntelliJ advertises plugin development, the documentation is very inconsistent and thus the development process can be challenging. Their examples usually provide solutions focused on a single function and including some other IDE functionality into the process can be complicated and difficult to figure out. There is, however, plenty of predefined functionality, so implementing some rather sophisticated feature (like parsing error messages or applying quick fixes) can be quite straightforward. For program analysis, PSI is an extremely valuable tool.

One of the trickiest parts about such tool development is coming up with the quick fixes since error patterns can be very varied. Currently, we provide a specific quick fix for unsafe statement execution. But there can be many other libraries or frameworks where such errors can raise. Writing separate quick fixes for each could be very labor intensive.

The speed of the analysis and annotations did not present a problem for the prototype; however, the examples used were very limited.

4.8 Future development

The current plugin was a prototype to lay the groundwork for a larger comprehensive static analysis tool development and also to understand the IntelliJ IDEA plugin development process. That is also the reason why current plugin will not be published as a separate plugin in IntelliJ repositories. We plan to add more sophisticated control flow analysis (e.g. access control, differential privacy), concurrency analysis and possibly some other sound techniques. The Checker Framework can probably be leveraged more, but adding other existing analyzers or writing our own will be considered.

Error reporting might also change. Once there will be different kinds of errors, it might be easier to comprehend if errors are reported under inspections. It allows for easier overview and grouping. The current solution uses the annotator approach because it is more integrated and simpler to use (if errors are of one type).

User customization of the analysis properties will probably be an important topic to consider. Ideally, we would want all programs to be correct and verified by a sound analysis; however, soundness is based on assumptions and is seldom absolute. Thus, users should be allowed to specify what they require from the analysis. But it should not be too easy to turn off or ignore the errors, which makes it a complicated question. Company-wide policies could be one solution, industry-wide standards for specific fields even more ambitious.

5 Conclusion

The current thesis was the first step towards a fully integrated interactive sound static analysis tool. We reviewed the current situation in the industry and looked at some of the approaches that try to improve the usability of sound analysis and verification. We extracted some of the important features such a tool should have and developed a prototype for tainting analysis. The prototype was implemented as an IntelliJ IDEA plugin using the Checker Framework as analysis backend.

The prototype gives well-integrated feedback about tainting analysis results and provides an interactive problem-fixing process. It offers a non-trivial quick fix for SQL injection problems. During the development, several problems were identified, including the lack of documentation for IntelliJ and difficulties in coming up with general quick fixes. This means that time and effort required for the final tool should be carefully planned; however, no fundamental weaknesses were discovered that would not allow the integration of a comprehensive static analysis tool into IntelliJ.

The main conclusion of the thesis is that a usable sound analysis framework needs to integrate both into the build system as well as the user's development environment.

The prototype was not evaluated on any real-world codebase, thus it is unclear how the solution would be able to find more sophisticated tainting violations. The next step would be analyzing some known vulnerable codebase. Further development should add more types of analyses from the Checker Framework and also write our own. Also integrating *angelic verification* and *verification modulo versions* approaches into the tool.

References

[1] Eclipse Foundation. https://www.eclipse.org/. Accessed 2018-08-01.

[2] FindBugs - Find Bugs in Java Programs. http://findbugs.sourceforge.net/. Accessed 2018-08-01.

[3] Github - IntelliJ IDEA Community Edition. https://github.com/JetBrains/intellij-community. Accessed 2018-08-01.

[4] IntelliJ IDEA. https://www.jetbrains.com/idea/. Accessed 2018-08-01.

[5] IntelliJ Platform SDK DevGuide. http://www.jetbrains.org/intellij/sdk/docs/basics.html. Accessed 2018-08-01.

[6] Java - Type Annotations and Pluggable Type Systems. https://docs.oracle.com/javase/tutorial/java/annotations/type_annotations.html. Accessed 2018-08-01.

[7] OWASP Top 10-2017 A1-Injection. https://www.owasp.org/index.php/Top_10-2017_A1-Injection. Accessed 2018-08-01.

[8] OWASP Top 10 Application Security Risks - 2017. https://www.owasp.org/index.php/Top_10-2017_Top_10. Accessed 2018-08-01.

[9] PSI Viewer for IntelliJ IDEA plugin development. https://github.com/cmf/psiviewer/. Accessed 2018-08-01.

[10] The Checker Framework. https://checkerframework.org/. Accessed 2018-08-01.

[11] The Checker Framework Manual. https://checkerframework.org/manual. Accessed 2018-08-01.

[12] Wikipedia - SQL injection. https://en.wikipedia.org/wiki/SQL_injection. Accessed 2018-08-01.

[13] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

[14] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium*, pages 3–11. Springer, 2015.

[16] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 332–343. IEEE, 2016.

[17] Ankush Das, Shuvendu K Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *International Conference on Computer Aided Verification*, pages 324–342. Springer, 2015.

[18] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *International Conference on Formal Verification of Object-Oriented Software*, pages 10–30. Springer, 2010.

[19] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[20] Rama Krishnan. SQL Injection in Java Application. `https://www.javacodegeeks.com/2012/11/sql-injection-in-java-application.html`. Accessed 2018-08-01.

[21] Francesco Logozzo, Shuvendu K Lahiri, Manuel Fähndrich, and Sam Blackshear. Verification modulo versions: Towards usable verification. In *ACM SIGPLAN Notices*, volume 49, pages 294–304. ACM, 2014.

[22] Daniil Ovchinnikov. Checker Framework integration for Intellij Platform. `https://github.com/dovchinnikov/intellij-checker-framework`. Accessed 2018-08-01.

[23] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.

[24] Andrei Sabelfeld. Language-based information security. In *Foundations of Computer Security*, page 99, 2003.

[25] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018.

[26] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 598–608. IEEE Press, 2015.

[27] Oleg Shelajev. RebelLabs Developer Productivity Report 2017: Why do you use the Java tools you use? https://zeroturnaround.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/. Accessed 2018-08-01.

[28] Jun Zhu, Jing Xie, Heather Richter Lipford, and Bill Chu. Supporting secure programming in web applications through interactive static analysis. *Journal of advanced research*, 5(4):449–462, 2014.

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public