MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors

Linghui Luo
Heinz Nixdorf Institute, Paderborn University, Germany
linghui.luo@upb.de

Julian Dolby
IBM Research, USA
dolby@us.ibm.com

Eric Bodden
Heinz Nixdorf Institute, Paderborn University & Fraunhofer IEM, Germany
eric.bodden@upb.de

Abstract

In the past, many static analyses have been created in academia, but only a few of them have found widespread use in industry. Those analyses which are adopted by developers usually have IDE support in the form of plugins, without which developers have no convenient mechanism to use the analysis. Hence, the key to making static analyses more accessible to developers is to integrate the analyses into IDEs and editors. However, integrating static analyses into IDEs is non-trivial: different IDEs have different UI workflows and APIs, expertise in those matters is required to write such plugins, and analysis experts are not typically familiar with doing this. As a result, especially in academia, most analysis tools are headless and only have command-line interfaces. To make static analyses more usable, we propose MagpieBridge—a general approach to integrating static analyses into IDEs and editors. MagpieBridge reduces the $m \times n$ complexity problem of integrating m analyses into n IDEs to $m + n$ complexity because each analysis and type of plugin need be done just once for MagpieBridge itself. We demonstrate our approach by integrating two existing analyses, Ariadne and CogniCrypt, into IDEs; these two analyses illustrate the generality of MagpieBridge, as they are based on different program analysis frameworks—WALA and Soot respectively—for different application areas—machine learning and security—and different programming languages—Python and Java. We show further generality of MagpieBridge by using multiple popular IDEs and editors, such as Eclipse, IntelliJ, PyCharm, Jupyter, Sublime Text and even Emacs and Vim.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases IDE, Tool Support, Static Analysis, Language Server Protocol

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.7

Category Tool Insights Paper

Funding This research was supported by the research training group "Human Centered Systems Security" (NERD.NRW) sponsored by the state of North-Rhine Westphalia in Germany and by the DFG's collaborative research center 1119 CROSSING.

1 Introduction

Many static analyses have been created to find a wide range of issues in code. Given the prominence of security exploits in practice, many analyses focus on security, such as TAJ [59], Andromeda [58], HybriDroid [34], FlowDroid [31], CogniCrypt [48] and DroidSafe [44]. There are also many analyses that address other code quality issues, such as FindBugs [46], SpotBugs [23], PMD [17] for common programming flaws (e.g. unused variables, dead code, empty catch blocks, unnecessary creation of objects, etc.) and TRACKER [57] for resource leaks. Other analyses target code performance, such as J2EE transaction tuning [41]. There

© Linghui Luo and Julian Dolby and Eric Bodden;
licensed under Creative Commons License CC-BY
33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 7; pp. 7:1–7:25
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

are also specialized analyses for specific domains, such as Ariadne [38] for machine learning. These analyses collectively represent a large amount of work, as they embody a variety of advanced analyses for a range of popular programming languages. To make this effort more tractable, many analyses are built on existing program analysis frameworks that provide state-of-the-art implementations of commonly-needed building blocks such as call-graph construction, pointer analysis, data-flow analysis and slicing, which in turn all rest on an underlying abstract internal representation (IR) of the program. Doop [7,33], Soot [21,49], Safe [19], Soufflé [22] and WALA [29] are well-known.

While development of these analyses has been a broad success of programming language research, there has been less adoption of such analyses in tools commonly used by developers, i.e., in interactive development environments (IDEs) such as Eclipse [8], IntelliJ [13], PyCharm [18], Android Studio [1], Spyder [24] and editors such as Visual Studio Code [28], Emacs [10], Atom [3], Sublime Text [26], Monaco [16] and Vim [27]. There have been some positive examples: the J2EE transaction analysis shipped in IBM WebSphere [12], Andromeda was included in IBM Security AppScan [2], both ultimately based on Eclipse technology. Similarly, CogniCrypt comprises an Eclipse plugin that exposes the results of its crypto-misuse analysis directly to the developer within the IDE. Each of these tools involved a substantial engineering effort to integrate a specific analysis for a specific language into a specific tool. Table 1 shows the amount of code in plugins for analyses is a significant fraction of code in the analysis itself. Given that degree of needed effort, the sheer variety of popular tools and potentially-useful analyses makes it impractical to build every combination.

Tool	Analysis (LOC)	Plugin (LOC)	Plugin/Analysis
FindBugs	132,343	16,670	0.13
SpotBugs	121,841	16,266	0.13
PMD	117,551	33,435	0.28
CogniCrypt	11,753	18,766	1.60
DroidSafe	41,313	8,839	0.21
Cheetah	4,747	864	0.18
SPLlift	1,317	3,317	2.52

Table 1 Comparison between the LOC (lines of Java code) for analysis and the LOC for plugin

While the difficulty of integrating such tools into different development environments has lead to poor adoption of these tools and research results in practice, it also makes empirical evaluations of them challenging. Evaluations of static analyses have been mostly restricted to automated experiments where the analyses are run in "headless" mode as command-line tools [31, 50, 53, 62], paying little to no attention to usability aspects on the side of the developer. As many recent studies show [35, 36, 47], however, those aspects are absolutely crucial: if program analysis tools do not yield actionable results, or if they do not report them in a way that developers can understand, then the tools will not be adopted. So to develop and evaluate such tools, researchers need ways to bring tools into IDEs more easily and quickly.

The ideal solution is the magic box shown in Figure 1, which adapts any analysis to any editor,[1] and presents the results computed by the analysis, e.g., security vulnerabilities or other bugs, using common idioms of the specific tool, e.g., problem lists or hovers.

[1] Note: In the following, when we write *editor*, we mean any code editor, which comprises IDEs.

In this work, we present MagpieBridge,[2] a system which uses two mechanisms to realize a large fraction of this ultimate goal:

1. Since many analyses are written using program analysis frameworks, MagpieBridge can focus on supporting the core data structures of these frameworks. For instance, analyses based on data-flow frameworks can be supported if the magic box can render their data-flow results naturally. Furthermore, while there are multiple frameworks, they share many common abstractions such as data flow and call graphs, which allows one to support multiple frameworks with relative ease.
2. More and more editors support the Language Server Protocol (LSP) [15], a protocol by which editors can obtain information from arbitrary “servers”. LSP is designed in terms of idioms common to IDEs, such as problem lists, hovers and the like. Thus, the magic box can take information from a range of analyses and render it in a few common tooling idioms. LSP support in each editor then displays these in the natural idiom of the editor.

Our system MagpieBridge exploits these two mechanisms to implement the magic box for analyses built using WALA or Soot, with more frameworks under development, and for any editor that supports the LSP. In this paper, we present the MagpieBridge workflow, explaining the common APIs we defined for enabling integration. We demonstrate two existing analyses—CogniCrypt and Ariadne, which are based on different frameworks (Soot and WALA), for different application areas (cryptography misuses and machine learning) and for different programming languages (Java and Python) into multiple popular IDEs and editors (Eclipse, Visual Studio Code, PyCharm, IntelliJ, JupyterLab, Monaco, Vim, Atom and Sublime Text) supporting different features (diagnostics, hovers and code lenses) using MagpieBridge. We make MagpieBridge publicly available as `https://github.com/MagpieBridge/MagpieBridge`.

[2] In a Chinese legend, a human and a fairy fall in love, but this love angers the gods, who separate them on opposite sides of the Milky Way. However, on the seventh day of the seventh lunar month each year, thousands of magpies form a bridge, called 鹊桥 in Chinese and Queqiao in pinyin, allowing the lovers to meet.

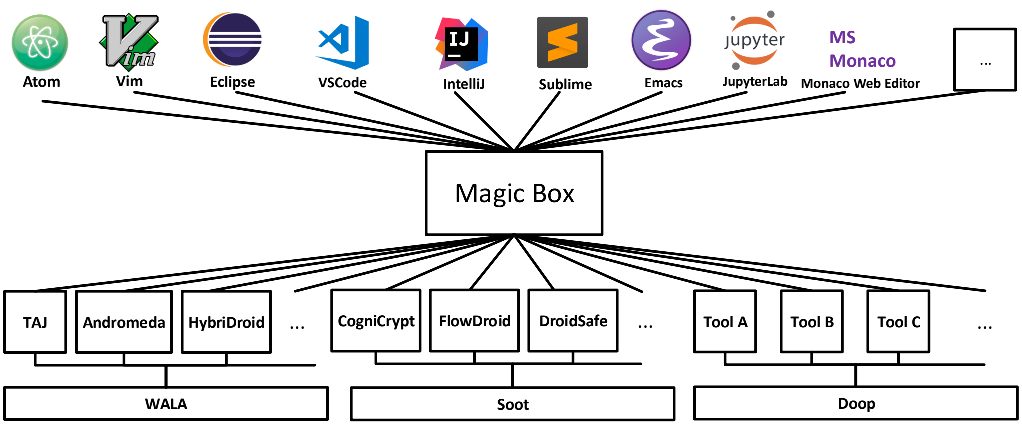


Figure 1 The desired solution: a magic box that connects arbitrary static analyses to arbitrary IDEs and editors

2 Background and Related Work

Existing tools and frameworks

Given the importance of programming tools for IDEs, there have been a variety of efforts to provide them, both commercial and open source. We here survey some significant ones, focusing on those that use WALA [40] or Soot [49,60] and hence are most directly comparable to our work.

There have been a few commercial tools, notably IBM AppScan [2] and RIGS IT Xanitizer [30]. Both products make use of WALA and target JavaScript among other languages. They comprise views to display analysis results as annotations to the source code, and allow for some triaging of the often longish lists of potential vulnerabilities within the IDEs. Among other issues, AppScan finds tainted flows and allows the user to focus on a specific flow through the program, although the user needs to decide what flow is of interest.

There has been a wider variety of open-source tools. WALA has been used in e.g. JOANA [43, 45]. Soot is used in the widely adopted open-source crypto-misuse analyzer Eclipse CogniCrypt [48], and is also part of the research tools Cheetah [36], SPLlift [32] and DroidSafe [44]. All tools named so far integrate with the Eclipse IDE.

JOANA focuses on Java, including Android, and provides a range of advanced analyses based on information flow control.

CogniCrypt is a tool to detect misuses of cryptographic APIs in Java and Android applications. Its current UI integration is relatively basic, offering simple error annotations in the program code and the problems view. CogniCrypt further comprises an XText-based [39] Eclipse plugin that allows developers to edit API-specification files using syntax highlighting and code completion. Those specification files directly determine the definition of the static analysis.

SPLlift is a research tool to analyze Java-based software product lines. Its UI is an extension to FeatureIDE [56], which allows it to show variations in the product line's code base through color coding. Detected programming errors are shown as code annotations and in the problems view. FeatureIDE itself is also an extension to Eclipse.

Cheetah is a research prototype for the just-in-time static taint analysis within IDEs. In Cheetah, the analysis is triggered upon saving a source-code file, but in its case the analysis is automatically prioritized to provide rapid updates to the error messages in those code regions that are in the developer's current scope. From there the analysis works its way outwards, potentially reporting errors in farther parts of the program only after several seconds or even minutes. Due to this mechanism, Cheetah requires the IDE to provide information about which file edit caused the analysis to be triggered, and what the project layout looks like. Cheetah also provides a somewhat richer UI integration than the previously named tools. For instance, when users select an individual taint-flow message in the problems view, it highlights in the code all statements involved in that particular taint, and also shows a list of those statements in a separate view—useful in case those are scattered across multiple source code files.

Analysis based on Doop [7, 33] has been experimentally integrated into the ProGuard optimizer for Android applications [61]. This is a once-off integration rather than a framework for Doop analyses, and it is focused on the build processs rather than the IDE itself. Still, it reflects the special-purpose integrations that show how analysis tends to be used.

Until now, program-analysis frameworks have focused on making it easier to develop analyses, with supportive infrastructure for basics such as scalable call graph, pointer analysis, and data-flow analysis. There have been presentations[3] and tutorials[4] at conferences which have provided both introductions and detailed tutorials for analysis construction; however, until now, there has been little focus on assisting with integrating such analyses into usable tools.

Language Server Protocol (LSP)

The Language Server Protocol (LSP) [15] is a JSON-based RPC protocol originally developed by Microsoft for its Visual Studio Code to support different programming languages. LSP follows a client/server architecture, in which “clients” are typically meant to be code editors, i.e., IDEs such as IntelliJ, Eclipse, etc., or traditional editors such as Visual Studio Code, Vim, Emacs or Sublime Text. Those clients can trigger certain actions in “servers”, e.g. by opening a source-code file. Those servers can be of different flavours, but LSP allows them to contribute certain contents to the editor’s user interface, such as code annotations, list items or hovers. We will give concrete examples, including screenshots, in Section 4. As we show in this work, the LSP’s design lends itself to implement static code analysis tools as servers. In such a design, clients trigger analysis servers through LSP, and those servers communicate back their results through LSP as well, causing analysis results to automatically be shown in the client through the respective editor’s native interfaces.

SASP and SARIF

The Static Analysis Server Protocol (SASP) [25], although similar in name to LSP, is a distinctly different protocol. Started in 2017 by the static code analysis vendor GrammaTech, it describes a standardized communication protocol to facilitate communication between static analysis tools and consumers of their results. Compared to LSP, it supports a richer data-exchange format that is explicitly fine-tuned to static analysis. This is realized through the Static Analysis Results Interchange Format (SARIF) [20, 25] that SASP uses to communicate static-analysis results from servers to clients. Generally, SASP therefore promises a more tight coupled integration compared to LSP static analyses into editors, potentially needing more work on the server. Also, as of now, SASP and SARIF have seen little adoption by tool vendors. Currently, the standard is mostly put forward by GrammaTech, which through SASP offers third-party static analysis tools to allow a triaging of those tools’ results in GrammaTech’s CodeSonar [5]. SARIF exporters currently exist for some few static analysis tools, including CogniCrypt [48], the Clang Static Analyzer [4], Cppcheck [6], and Facebook Infer [11], which makes them amenable for an integration through SASP. However, right now, CodeSonar appears to be the only client ready to consume SARIF results, and it is unclear whether this will change in the near future. It is for this reason that MagpieBridge builds, for now, on top of LSP instead of SASP and SARIF. Furthermore, SASP is currently still in the early stage of its development and there exists no formal specification of the protocol [25], which makes it hard to compare it to LSP in detail and hard to use for our work.

[3] e.g. `https://souffle-lang.github.io/pdf/SoufflePLDITutorial.pdf`
[4] e.g. `http://wala.sourceforge.net/wiki/index.php/Tutorial`

3 Approach

3.1 The MagpieBridge Workflow

MagpieBridge uses the Language Server Protocol to integrate program analyses into editor and IDE clients. MagpieBridge is implemented using the Eclipse LSP4J [9] LSP implementation based on JSON-RPC [14], but MagpieBridge hides LSP4J details and presents an interface in terms of high-level analysis abstractions. The overall workflow is shown in Figure 2.

There are multiple mechanisms by which LSP-based tools can be used, but the most common mechanism is that an IDE or editor is configured to launch any desired tools. Each tool is built as a jar file based on the MagpieServer, with a main method that creates a `MagpieServer` (Listing 1), then adds the desired program analyses (`ServerAnalysis` in Listing 2) with `addAnalysis`, and then launches `MagpieServer` with `launch` so that it receives messages. This is shown with the `addAnalysis` and `launch` edges in Figure 2. With such a jar, MagpieBridge can be used simply by configuring an editor to launch it. Figure 3 shows our Sublime Text setup to launch both Ariadne and CogniCrypt analyses. The user merely obtains jar files of the analyses and sets up Sublime Text to launch each of them for the appropriate languages. That is all the setup that is needed.

Based on LSP4J, there are several mechanisms for sending and receiving messages. Most clients/editors simply launch the server and then expect it to handle messages using standard I/O (e.g. Eclipse, IntelliJ, Emacs and Vim); however some clients expect to talk using a well-known socket (e.g. Spyder), Web-based tools communicate using WebSockets (e.g.

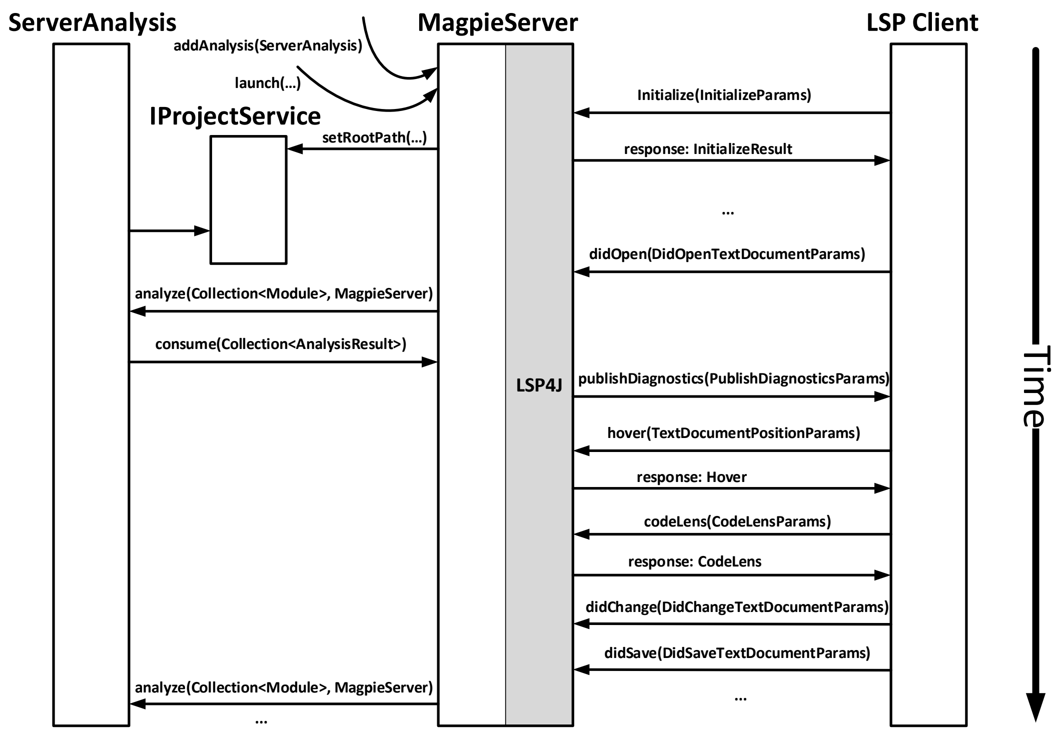


Figure 2 Overall MagpieBridge workflow

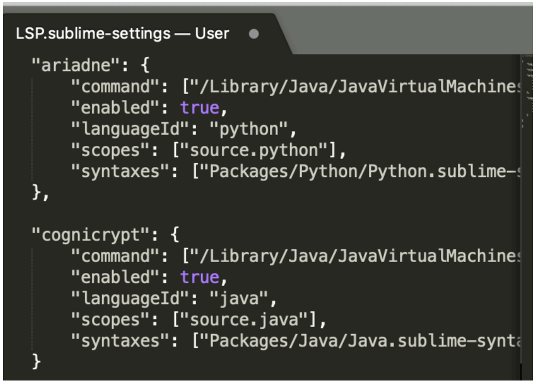


Figure 3 Configuration for Sublime Text to launch MagpieServer

Jupyter and Monaco) and only few tools support both standard I/O and socket (e.g. Visual Studio Code). Our `MagpieServer` supports all these channels out of the box and can be configured to communicate with a client using any of the channels.

Once `MagpieServer` is launched, it interacts with the client tool using standard LSP mechanisms:

- The first step is initialization. The client sends an `initialize` message to the server, which includes information about the project being analyzed, such as its project root path. `MagpieServer` calls `setRootPath` on each `IProjectService` (service that resolves project scope such as source code path and library code path) instance to initialize project path information. MagpieBridge currently understands Eclipse, Maven and Gradle projects. `MagpieServer` also sends the response `InitializeResult` which declares its capabilities back to the client. This is shown in the upper portion of Figure 2
- Subsequently, the client informs `MagpieServer` whenever it works with a file: the `didOpen`, `didChange` and `didSave` messages are sent to the server whenever files are opened, edited and saved respectively. These messages allow MagpieBridge to call the analysis via the `analyze` method whenever anything changes. Each analysis server decides the granularity of when it actually runs analysis and how much analysis it does. This is shown with the `didOpen` and `analyze` edges in Figure 2
- As shown in the rest of Figure 2, analysis uses the `consume` method to report analysis results of type `AnalysisResult` (Listing 4) to `MagpieServer`, which handles them via the appropriate LSP mechanism, specified by the `kind` method (Listing 4), which returns a `Kind` (Listing 5):

  Diagnostic denotes issues found in the code, corresponding to lists of errors and warnings that might be reported by a compiler. Tools typically report them either in a list of results or highlight the results directly in the code. When the program analysis provides such results via `consume`, `MagpieServer` reports them to the client tool with the LSP `publishDiagnostics` API.

  Hover denotes annotations to be displayed for a specific program variable or location. It could be used to report e.g. the type of a variable or the targets of a function call. Tools often show them when the cursor highlights a specific location. When the program analysis provides such results via `consume`, `MagpieServer` keeps them and reports them to the client tool as responses to LSP `hover` API calls by the client tool.

  CodeLens denotes information to be added inline in the source code, analogous to generated comments. Tools typically report them as distinguished lines of text inserted between lines of source code. When the program analysis provides such results via `consume`, `MagpieServer` keeps them and reports them to the client tool as responses to LSP `codeLens` API calls by the client tool.

These analysis results have a `position` method that returns a `Position` (Listing 6) denoting the source location to which the result pertains. The result requires a precise location based on starting and ending line and column numbers, which is required by the LSP protocol. Note that the `Position` of MagpieBridge implements the Java `Comparable` interface; MagpieBridge exploits this to store analysis results in `NavigableMap` structures so that it can find the nearest result if a user hovers in a location near result, e.g. some whitespace immediately after a variable or expression.

```java
public class MagpieServer implements LanguageServer, LanguageClientAware{
    protected LanguageClient lspClient;
    protected Map<String, IProjectService> languageProjectServices;
    protected Map<String, Set<ServerAnalysis>> languageAnalyses;

    public void addProjectService(String language, IProjectService projectService){...}
    public void addAnalysis(String language, ServerAnalysis analysis){...}
    public void doAnalyses(String language){...}
    public void consume(Collection<AnalysisResult>){...}

    protected Consumer<AnalysisResult> createDiagnosticConsumer(){...}
    protected Consumer<AnalysisResult> createHoverConsumer(){...}
    protected Consumer<AnalysisResult> createCodeLensConsumer(){...}
    ...
}
```

Listing 1 The core of the server

```java
public interface ServerAnalysis{
    public String source();
    public void analyze(Collection<Module> files, MagpieServer server);
}
```

Listing 2 Interface for defining analysis on the server

```java
public interface IProjectService {
    public void setRootPath(Path rootPath);
}
```

Listing 3 Interface for defining service which resolves project scope

```java
public interface AnalysisResult {
    public Kind kind();
    public String toString(boolean useMarkdown);
    public Position position();
    public Iterable<Pair<Position,String>> related();
    public DiagnosticSeverity severity();
    public Pair<Position, String> repair();
}
```

Listing 4 Interface for defining analysis result

```java
public enum Kind {
    Diagnostic, Hover, CodeLens
}
```

Listing 5 Enum for defining kinds of analysis results

```java
public interface Position extends Comparable {
    public int getFirstLine();
    public int getLastLine();
    public int getFirstCol();
    public int getLastCol();
    public int getFirstOffset();
    public int getLastOffset();
    public URL getURL();
}
```

Listing 6 Interface for defining position

3.2 The MagpieBridge System

We explain our MagpieBridge system with an overview in Figure 4. MagpieBridge needs to support various analysis tools that were built on top of different frameworks, e.g., TAJ, Andromeda and HybriDroid use WALA, while CogniCrypt, FlowDroid and DroidSafe rely on Soot and many other analyses are based on Doop. These analysis frameworks have different IRs, which MagpieBridge needs to use to generate analysis results. One key requirement for all the frameworks supported by MagpieBridge is very precise source-code

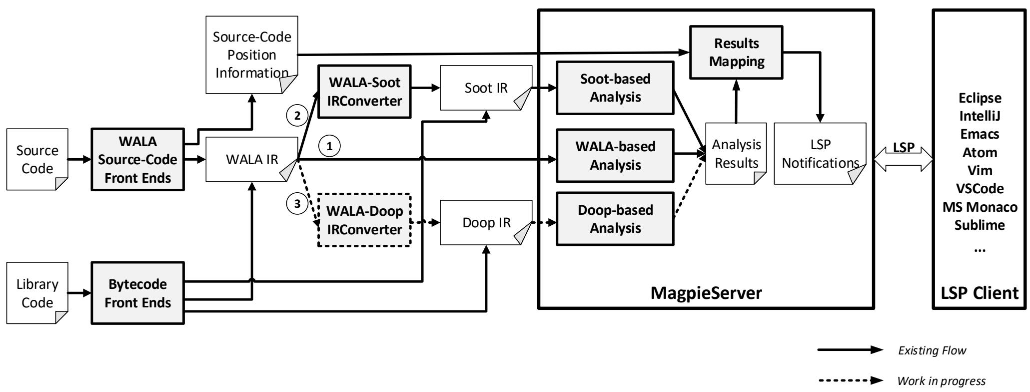


Figure 4 Overview of our MagpieBridge system

mappings, since in LSP all the messages communicate using starting and ending line and column numbers. In the following we explain how MagpieBridge achieves this requirement for WALA-based analyses, Soot-based analyses and Doop-based analyses respectively.

3.2.1 WALA-based Analysis

The simplest code path in MagpieBridge (flow ① in Figure 4) uses WALA source language front ends for creating IR on which to perform analysis. WALA comprises both bytecode and source-code front ends for different languages (Java, Python and JavaScript), and the source-code front end preserves source-code positions very well. This information can be consumed later in the LSP notifications, since it is kept in WALA's IR. WALA's IR is a traditional three-address code in Static Single Assignment (SSA) form, which is translated from WALA's Common Abstract Syntax Tree (CAst).

The approach to source-code front ends for WALA is using existing infrastructure for each supported language: Eclipse JDT for Java, Mozilla Rhino for JavaScript and Jython for Python. Each of these front ends is maintained with respect to its respective language standards, and all the front ends provide precise mappings of source locations for constructs. To provide detailed source mapping for the generated IR, each WALA function body has an instance of `DebuggingInformation` (Listing 7) which allows MagpieBridge to map locations from requests to IR elements at a very fine level.

```
public interface DebuggingInformation {
    Position getCodeBodyPosition();
    Position getCodeNamePosition();
    Position getInstructionPosition(int instructionOffset);
    String[][] getSourceNamesForValues();
    Position getOperandPosition(int instructionOffset, int operand);
    Position getParameterPosition(int param);
}
```

Listing 7 Debugging information interface

Listing 7 details how much source mapping information is available. `getCodeBodyPosition` is the source range of the entire function, and `getCodeNamePosition` is the position of just the name in the body. `getInstructionPosition` is the source position of a given IR instruction. `getOperandPosition` is the source position of a given operand in an IR instruction. `getParameterPosition` is the position of a given parameter declaration in the source.

3.2.2 Soot-based Analysis

Soot comprises a solid Java bytecode front end. The bytecode only has the line number of each statement. This is not sufficient to support features such as hover, fix and codeLens in an editor. For those features, position information about variable, expressions, calls and parameters are necessary. However, they are lost in the bytecode. Soot further comprises source-code front ends. Such front ends, however, require frequent updates due to the frequently changing specification of the Java source language, which has caused Soot's source-code front ends to become outdated. Besides, Soot IR was not designed to keep precise source-code position information, e.g., there is no API for getting the parameter position in a method. Our approach is to take WALA's source-code front end to generate WALA IR and convert it to Soot IR. Soot has multiple IRs, the most commonly used IR is called Jimple [60]. Jimple is also a three-address code and has Java-like syntax, but is simpler, e.g., no nested statements. Opposed to WALA IR, Jimple is not in SSA-form. Both WALA and Soot are implemented in Java and manipulate the IR through Java objects. This makes the conversion between the IRs feasible. In particular, we have implemented the WALA-Soot IRConverter and defined the common APIs (Listing 4) to encode analysis results, as well as the `MagpieServer` (Listing 1) that hosts the analysis. Currently the WALA-Soot IRConverter only converts WALA IR generated by WALA's Java source-code front end. In fact, WALA uses a pre-IR before generating the actual WALA IR in SSA-form, and this non-SSA pre-IR is actually the IR that we convert to Jimple. Since also Jimple is not in SSA, this conversion is more direct. This pre-IR contains 24 different instructions as shown in Figure 5. After studying both IRs, we found out that 15 instructions in WALA IR can be converted to JAssignStmt in Jimple. Most of the times the conversion is one-to-one, only a few cases are one-to-many. The precise source-code position information from WALA IR is encapsulated in the tags (annotations) of the converted Soot IR. In the future, we plan to convert WALA IR from front ends of other languages such as Python and JavaScript to a potentially extended version of the Soot IR.

The flow ② in Figure 4 for integrating Soot-based analysis starts by dividing the analyzed program code into application source code and library code (which can be in binary form). The source code is parsed by one of WALA's source-code front end and it outputs WALA IR, as well as precise source code position information associated in the IR. For a Soot-based analysis, the WALA IR is translated by a WALA-Soot IRConverter into Soot IR

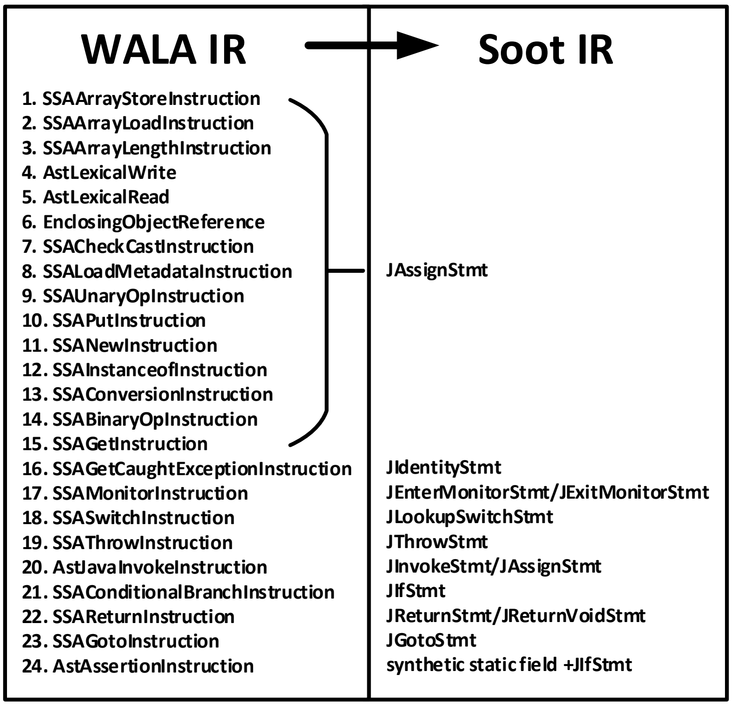


Figure 5 Conversion from WALA IR to Soot IR

```java
public class ExampleAnalysis implements ServerAnalysis{

    @Overide
    public String source(){
        return "Example Analysis"
    }

    @Overide
    public void analyze(Collection<Module> sources, MagpieServer server){
        ExampleTransformer t = getExampleTransformer();
        loadSourceCodeWithWALA(sources);
        JavaProjectService service = (JavaProjectService)
             server.getProjectService("java");
        loadLibraryCodeWithSoot(service.getLibraryPath());
        runSootPacks(t);
        List<AnalysisResult> results = t.getAnalysisResults();
        server.consume(results);
    }
    ...
}

public class Example{

    public static void main(String... args){
        MagpieServer server = new MagpieServer();
        IProjectService service = new JavaProjectService();
        ExampleAnalysis analysis = new ExampleAnalysis();
        String language = "java";
        server.addProjectService(language, service);
        server.addAnalysis(language, analysis);
        server.launch(...);
    }
}
```

Listing 8 The MagpieServer runs a Soot-based analysis

(Jimple). The library code is parsed by Soot's bytecode front end and then complements the program's IR obtained from the source code. The Soot IR in Figure 4 thus consists of two parts: Jimple converted by the WALA-Soot IRConverter, which represents the source-code portion/application code of the program, and Jimple generated by Soot's bytecode front end which represents the library code. Based on the composite Soot IR, Soot further conducts a call graph and optionally also pointer analysis, which can then be followed by arbitrary data-flow analyses.

Listing 8 shows an example of running a Soot-based analysis `ExampleTransformer` (analyses are called transformers in Soot) on the `MagpieServer`. The `ExampleTransformer` accesses the program through the singleton object `Scene` in order to analyze the program. Once the `MagpieServer` receives the source code, the method `loadSourceCodeWithWALA` parses the source code, converts it to Soot IR with the WALA-Soot IRConverter and stores the IR in the `Scene`. The class `JavaProjectService` resolves library path for the current project. `loadLibraryCodeWithSoot` loads the necessary library code from the path and adds the IR into `Scene`. The method `runSootPacks` invokes Soot to build call-graph and run the actual analysis. The analysis results will be then consumed by the server. In this example, only the source files sent to the server are analyzed together with the library code. However, it can be configured to perform a whole-program analysis, since the source code path can also be resolved by `JavaProjectService`.

We explain how the class `JavaProjectService` which implements `IProjectService` resolves the full Java project scope, i.e., source code path and library code path. As specified in LSP, the editors send the project root path (rootURI) to the server in the first request `initialize`. Library and source code path can be resolved by using the build-tool dependency plugins (e.g. caching results of mvn dependency:list) or parsing the configuration (e.g. pom.xml, build.gradle) and source code files located in the root path. Project structure conventions for different kinds of projects are also considered in MagpieBridge. For more customized projects, MagpieBridge also allows the user to specify the library and source code path manually as program arguments.

3.2.3 Doop-based Analysis

Doop uses Datalog to allow for declarative analysis specifications, encoding instructions as Datalog relations as well as instruction source positions. There is code to convert from the WALA Python IR to Datalog, and that captures both the semantics of statements as well as source mapping, and these declarations capture the information needed for analysis tool support. For instance, there is a Datalog relation that captures instruction positions and is generated directly from WALA IR:

```
.decl Instruction_SourcePosition(?insn:Instruction,
  ?startLine:number, ?endLine:number, ?startColumn:number, ?endColumn:number)
```

This code has been used experimentally for analysis using Doop of machine code written in Python. This code path could be used to express analyses in editors using MagpieBridge, and such work is under development.

4 Demonstration

To make MagpieBridge more concrete, we use two illustrative analyses, based on different frameworks—Soot and WALA, respectively—for different languages—Java and Python—in different domains—security and bug finding—both in a range of editors:

CogniCrypt analyzes how cryptographic APIs are used in a program, and reports a variety of vulnerabilities such as encryption protocols being misused or when protocols are used in situations where they should not. The tool then also gives suggestions on how to fix the problem. CogniCrypt comprises a highly efficient demand-driven, inter-procedural data-flow analysis [55] based on Soot, and has its own Eclipse-based plugin. As Table 1 shows, its plugin actually required substantially more code than the analysis itself. The plugin also is limited to Eclipse. We illustrate what it looks like to use CogniCrypt in multiple tools using MagpieBridge. To keep exposition simple, we focus on a case in which a weak encryption mode is used (Electronic Codebook Mode, ECB). In the general case the analysis can also report complex flows through the program. Screenshots in Figure 6, Figure 7, Figure 8 and Figure 9 show the crypto warning reported by CogniCrypt in different editors. As we can see, only the call `Cipher.getInstance` with the insecure parameter is marked in each editor.

Ariadne analyzes how tensor (multi-dimensional array) data structures are used in machine-learning code written in Python, and reports a range of information. It presents basic tensor-shape information for program variables, and finds and fixes certain kinds of program bugs. A key operation is reshaping a tensor: the `reshape` operation takes a tensor and a new shape, and returns a new tensor with the desired shape when that is possible. To simplify complex tensor semantics, a tensor can be reshaped only when its total size is equal to size of the desired new shape. Another operation is performing a convolution, e.g. `conv2d`, which requires the input tensor to have a specific number of dimensions. We illustrate cases of these bugs, and how they are shown in multiple editors (Figure 10, Figure 11, Figure 12, Figure 13, and Figure 14).

We illustrate how the aspects of LSP used by MagpieBridge are rendered in a variety of editors; while there are common notions such as a list of diagnostics, different tools make different choices in how those elements are displayed. We describe in turn several LSP aspects and how analysis information is displayed using them.

4.1 Diagnostics

The most straightforward interface is for an analysis to report a set of issues, but even this simple concept is handled differently in different editors.

- Some editors have a problem view, i.e., a list summarizing all outstanding issues. An example of this interface is Sublime Text, illustrated in Figure 8 where a warning about weak encryption is shown in a list.
- Some editors do not have such a list, but choose to highlight issues directly in the code. An example of this interface is Monaco, illustrated in Figure 7; the same warning about weak encryption is shown inline. To minimize clutter, editors typically make such warnings as hovers, and we show it displayed in Monaco. A somewhat different visualization of the same idea is in Figure 13, in which Atom shows an invalid use of `reshape` in Tensorflow.
- Some editors do both. An example of this interface is Eclipse, illustrated in Figure 6 where a warning about weak encryption is shown both inline and in a list. Again to minimize clutter, the inline message is realized via a hover.

Note that all issues displayed here are computed by the very same analysis in all editors and rendered as the same LSP objects; however, they appear natural in each editor, due to the editor-specific LSP client implementations.

4.2 Code Lenses

Code lenses look like comments, but are inserted into the code by analyses and are used to reflect generally-useful information about the program. An example is shown in Figure 10, in which the shapes of tensors are listed explicitly for various program variables and function arguments.

4.3 Hovers

Hovers are used to reflect generally-useful information about the program, but, unlike code lenses, they are visible only on demand. As such, an analysis can sprinkle them liberally in the program and they will not be distracting since they are only visible when needed. Different tools have different ways of user interaction. In Figure 11, the user hovers over the variable `x_dict` in PyCharm to reveal the shape of tensors that it holds. In Figure 12, the user enters a Vim command with the cursor over the variable `x_dict`.

4.3.1 Repairs

LSP provides the ability to specify fixes for diagnostics; a diagnostic can specify replacement text for the text to which the given diagnostic applies. The method `repair()` in the interface `AnalysisResult` is designed exactly for this purpose (see Listing 4). Figure 14 shows an example of this: the top half shows an error report in Visual Studio Code that a call to `conv2d` is invalid, since such calls require a tensor with four dimensions whereas the provided

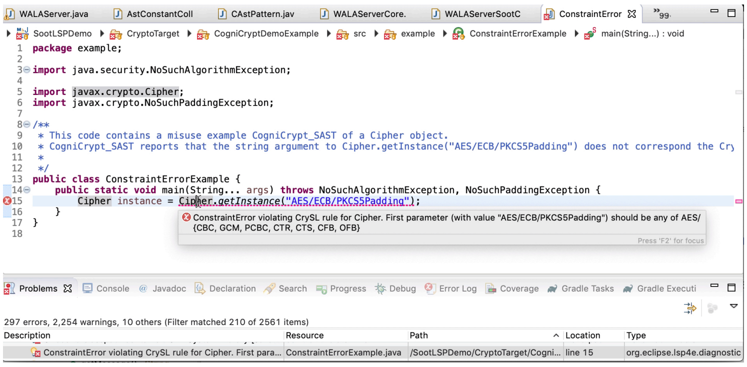


Figure 6 Insecure crypto warning in Eclipse

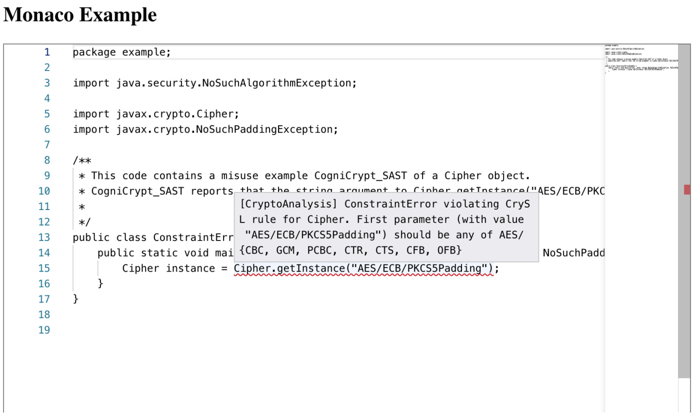


Figure 7 Insecure crypto warning in Monaco

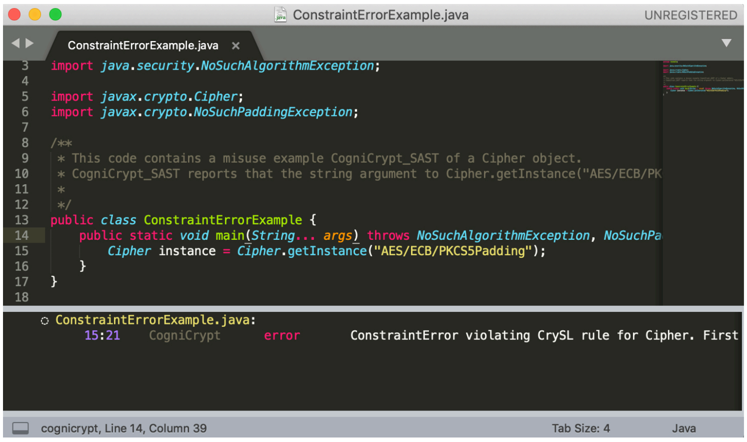


Figure 8 Insecure crypto warning in Sublime Text

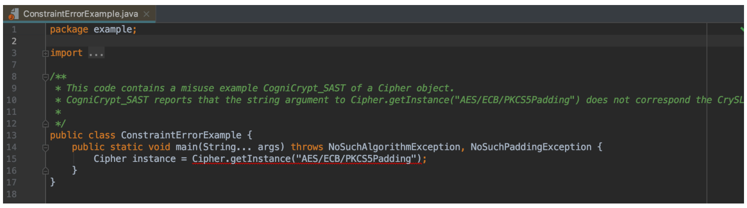
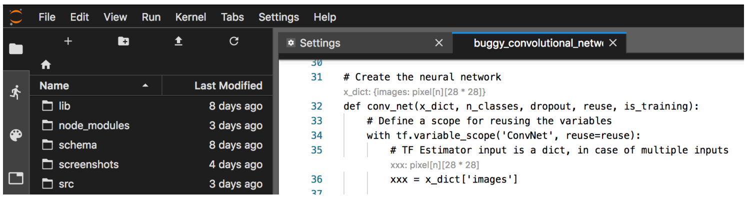
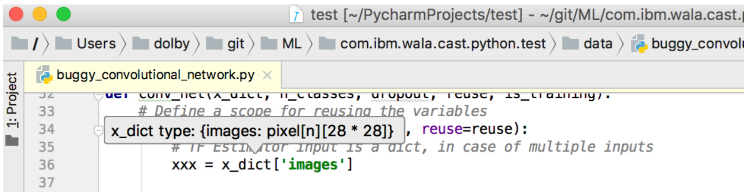


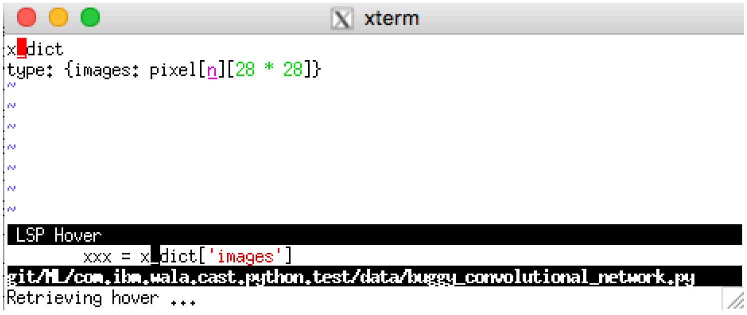
Figure 9 Insecure crypto warning in IntelliJ



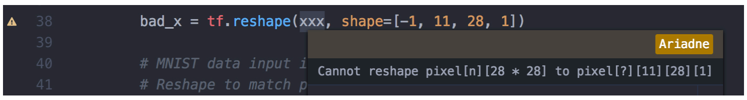
■ **Figure 10** Code lenes showing tensor types in JupyterLab



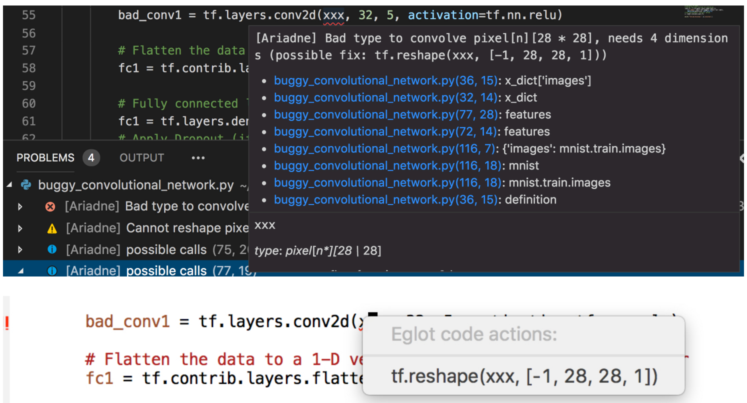
■ **Figure 11** Hover tip showing tensor types in PyCharm



■ **Figure 12** Hover tip showing tensor types in Vim



■ **Figure 13** Diagnostic warning showing an incompatible `reshape` in Atom



■ **Figure 14** Diagnostic error showing fixable incorrect dimensions for `conv2d`. Error shown in Visual Studio Code and quick fix in Emacs.

argument has only 2. However, the analysis determines that a plausible fix is to `reshape` the provided argument to have more dimensions, and the lower part of the figure shows a prompt, in Emacs, suggesting a `reshape` call to insert.

5 Comparison Between MagpieBridge-Based Approach and Plugin-Based Approach

While MagpieBridge enables analyses to run in a larger set of IDEs, the question remains of how the support in any specific IDE using MagpieBridge compares to a custom-built plugin for that same IDE. Because most analysis tools do not have integration with most IDEs, we are going to focus our comparison on one existing combination: the CogniCrypt plugin for Eclipse. Afterwards, we discuss in more general terms the range of functionality exploited by custom plugins that is supported by LSP.

5.1 Comparison Between MagpieBridge-Based CogniCrypt and CogniCrypt Eclipse Plugin

The CogniCrypt Eclipse Plugin [48] consists of two components: code generation, which generates secure implementations for user-defined cryptographic programming tasks, and cryptographic misuse detection, which runs static code analysis in the background and reports insecure usage of cryptographic APIs. MagpieBridge focuses on analysis, and so we do not consider the code-generation component here. For comparison, we integrated the static crypto analysis of CogniCrypt with MagpieBridge into Eclipse IDE.

Figure 15 and Figure 16 are screenshots in which the original CogniCrypt Eclipse Plugin reports insecure crypto warnings. In comparison, Figure 17 shows our CogniCrypt-integration with MagpieBridge. Figure 15 shows two buttons that CogniCrypt adds to the toolbar: "Generate Code For Cryptographic Task" and "Apply CogniCrypt Misuse to Selected Project". By clicking the latter, one triggers the misuse detection using the plugin in its default configuration. The plugin can also be configured to trigger the analysis whenever a Java file is saved. On the other hand, MagpieBridge-based CogniCrypt starts the analysis automatically whenever a Java file is opened or saved. In either case, after the analysis has been run, any detected misuses are indicated in Eclipse in several ways, which the corresponding numbers show in Figure 15 and Figure 17:

1. In the Package Explorer view, the error ticks appear on the affected Java element and their parent elements.
2. In the Problems view, the detected misuses are listed as errors.
3. The editor tab is annotated with an error marker.
4. In the editor's vertical ruler / gutter, an error marker is displayed near the affected line.

As shown in Figure 16, one can hover over an error marker next to the affected line to view the description of the misuse. The appearance of the MagpieBridge-based and plugin-based CogniCrypt is rather similar, with just a few differences:

- MagpieBridge-based CogniCrypt does not change the appearance of the IDE. To work with the MagpieServer which runs the crypto analysis, end-users do not have to do anything different. The analysis runs automatically whenever a Java file is opened or saved by an end-user. In contrast, in the Eclipse Plugin, one can trigger the analysis manually, or (optionally) have it started automatically whenever a file is saved.
- Results are indicated similarly in the CogniCrypt Eclipse Plugin MagpieBridge-based CogniCrypt; however, in MagpieBridge-based CogniCrypt in addition to the error markers, squiggly lines appear under the affected lines.

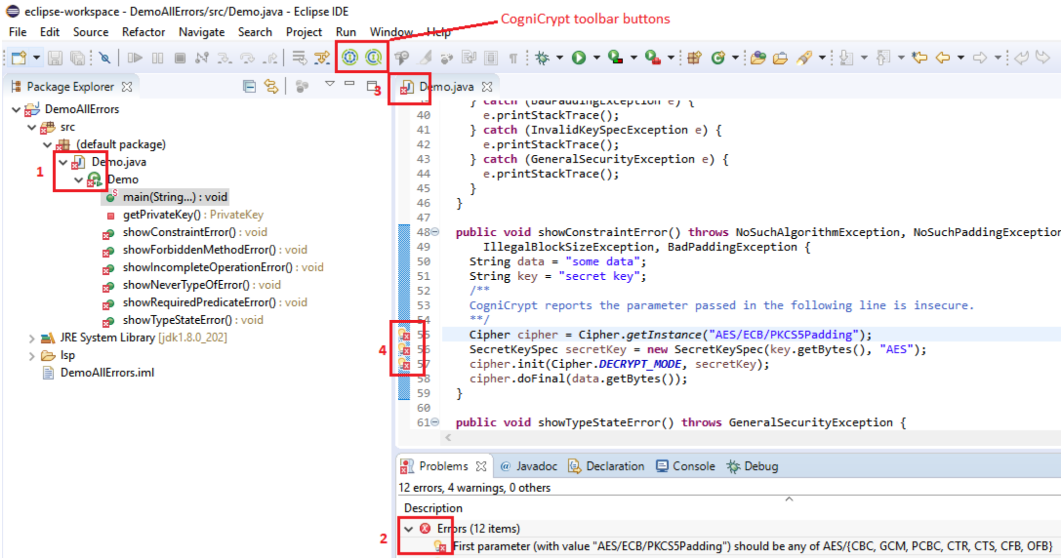


Figure 15 The appearance of CogniCrypt Eclipse Plugin

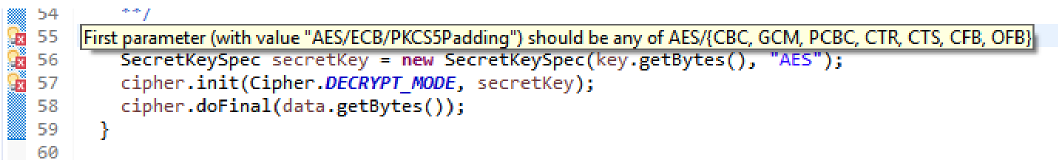


Figure 16 CogniCrypt Eclipse Plugin: insecure crypto warning message shown by hovering

- In MagpieBridge-based CogniCrypt, the hover message also includes a quick fix that can replace the insecure parameter `AES/ECB/PKCS5Padding` with a secure parameter `ASE/CBC/PKCS5Padding` automatically. Since MagpieBridge preserves the precise source code position from the WALA source-code front end, e.g., the exact code range (starting/ending line/column numbers) of each parameter of a method call, we were able to build such quick fix easily with the `codeAction` feature supported by LSP. Such quick fix is not available in the CogniCrypt Eclipse Plugin, although the warning message already indicates what a secure parameter should look like.

Another difference is that, since MagpieBridge does not add buttons to the IDE, it needs to invoke the analysis automatically. When the end-user changes the opened file, the MagpieServer clears the warnings when it receives the `didChange` notification from the IDE. The analysis is then restarted whenever the end-user saves the file, i.e., the MagpieServer receives a `didSave` notification. Once the MagpieServer receives the notification from the Eclipse IDE, it resolves the source code and library code path required for the inter-procedural crypto analysis. This analysis is all asynchronous, so that the analysis always runs in the background and updated error messages are shown once they are available. If they want to, end-users have the ability to connect and disconnect the MagpieServer at runtime, e.g., via "Preferences" in Eclipse IDE.

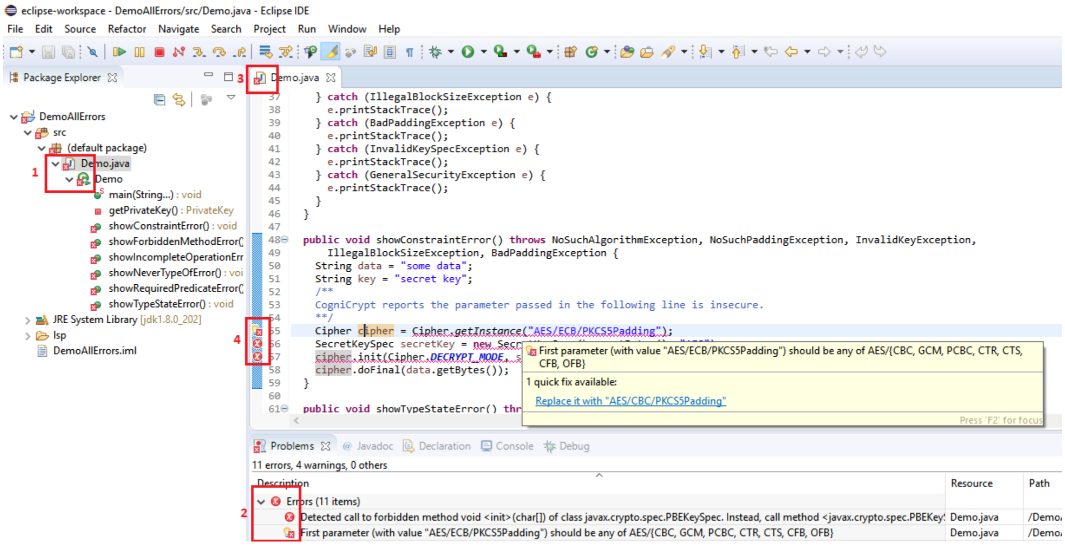


Figure 17 The appearance of MagpieBridge-based CogniCrypt: insecure crypto warning message and quick fix shown by hovering

5.2 Comparison to Other Plugin-Based Approaches

As shown in Figure 18, LSP offers a set of UI features to present the analysis results to end-users that are sufficient to capture the majority of UI features used in a range of existing plugins for a single analysis tool in a specific IDE. Most of the plugin approaches we identified were implemented as Eclipse plugins (Cheetah [37], SpotBugs [23] and ASIDE [63]), but some of them were created for other popular IDEs such as Android Studio (FixDroid [52]), IntelliJ (wIDE [51]) and Visual Studio (GhostFactor [42]). Figure 18 shows the comparison between features that can be supported with LSP to features supported by these existing plugin approaches.

Some plugins do use IDE features that are not explicitly supported by LSP; however,

Feature Comparison

Feature	LSP-based Approach	FixDroid (Android Studio)	wIDE (IntelliJ)	GhostFactor (Visual Studio)	Cheetah (Eclipse)	SpotBugs (Eclipse)	ASIDE (Eclipse)	# Plugins support the feature
Warning Marker	✓	✓		✓	✓	✓	✓	5
Code Highlighting	✓	✓	✓		✓		✓	4
Code Actions (quick fix, code	✓	✓		✓			✓	3
Hover Tips	✓	✓	✓	✓	✓	✓	✓	6
Pop-ups	✓	✓	✓					2
Code Change Detection	✓	✓					✓	2
Customized Icons		✓			✓		✓	3
Customized Views			✓		✓		✓	3
Customized Wizards			✓					1

Figure 18 Feature comparison between LSP-based approach and other plugin-based approaches

there are often analogs in LSP that could be used instead. For instance, Cheetah uses a custom view, essentially a separate window panel in the IDE, to show an example data-flow trace for a bug; in LSP, related information capturing a trace can be attached to problems as illustrated in Figure 14. Other uses of custom views and wizards are mainly for analysis configuration. Simple forms of such analysis configuration could be supported by the message protocol in LSP.

One minor feature unsupported by LSP appeared in the plugins: customized icons (see Figure 19, Figure 20 and Figure 21) are not supported by the LSP-based approach, since that requires changes to the appearance of the IDEs, which LSP intends not to. Although studies have shown customized icons are useful to catch end-users' attention [52, 54, 63], it is not clear if it is more effective than the default error icon supported by each editor.

As we can see in Figure 18, the major features such as hover tips, warning marker and code highlighting, which are supported by a majority of the plugins, can be supported by an LSP-based approach. However, LSP support varies across IDEs, both in what features are handled and how they are shown. In LSP, hover tips are specified as the `hover` request sent from the client to the server, warning marker can be realized by the `publishDiagnostics` notification and `documentHighlight` is the corresponding request for code highlighting. However, the implementation of `documentHighlight` varies from editor to editor, since the specification for this feature in LSP is unclear. Most plugins listed in Figure 18 support code highlighting. This features means changing the background color of affected lines of code as shown in Figure 19, Figure 20 and Figure 21. While Visual Studio Code limits this feature to only highlights all references to a symbol scoped in a file, sublime Text choses an underline for highlighting (see Figure 23). In addition, there is no possibility with LSP to specify the background color used in this feature, all editors have their pre-defined colors.

Some advanced features such as code actions (we have shown quick fix with MagpieBridge-based CogniCrypt), pop-ups and code change detections can also be supported by LSP. There are two interfaces (`showMessage` and `showMessageRequest`) defined in LSP which are implemented as pop-up windows in editors. Figure 24 shows a message sent from a server to the Eclipse IDE that is displayed in a pop-up window. Where more interactions are required, the interface `showMessageRequest` allows to pass actions and wait for an answer from the client. Figure 25 shows a pop-up windows with a message and available actions in Visual Studio Code.

Features that are not supported by LSP for now can be extended to LSP in the future, since LSP is a moving target with ever-growing functionality and support. One just has to keep in mind that, as the LSP is extended, the IDEs/editors that support it, might require extensions as well.

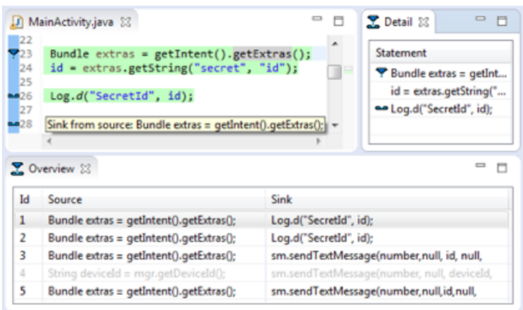


Figure 19 Cheetah: code highlighting, hover tips, customized icon and views

Figure 20 FixDroid: code highlighting, hover tips and customized icon

Figure 21 ASIDE: code highlighting and customized icon

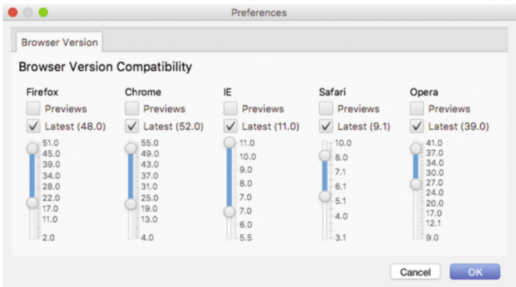


Figure 22 wIDE: customized wizard

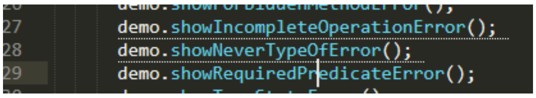


Figure 23 Highlighting in Sublime Text

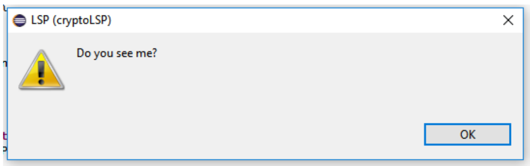


Figure 24 Pop-up in Eclipse

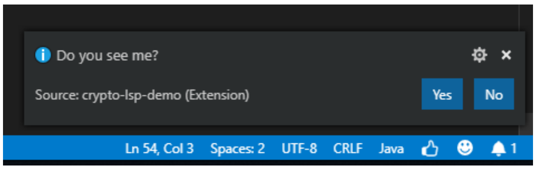


Figure 25 Pop-up with actions in Visual Studio Code

6 Conclusion and Future Work

The difficulty of integrating static tools into different IDEs and editors has caused little adoption of the tools by developers and researchers, and MagpieBridge addresses this problem by providing a general approach to integrating static analyses into IDEs and editors. MagpieBridge uses the increasingly popular Language Server Protocol and supports from rich analysis frameworks, WALA and Soot. We have shown MagpieBridge supporting CogniCrypt, but this is just the beginning; we conclude and presage future work by showing what is, to the best of our knowledge, the first ever IDE integration of the well-known FlowDroid security analysis. Figure 26 shows FlowDroid analyzing the data flow starting from a parameter of the HTTP request, finding a cross-site scripting vulnerability which can be exploited by attackers, and showing a witness trace of it. The expressions in the witness are shown precisely, which is possible since the IRConverter of MagpieBridge is able to run FlowDroid unchanged on the converted IR and recover precise source mappings. As far as we know, this has never been done before with FlowDroid. MagpieBridge then renders this precise trace from FlowDroid in the IDE, also the first time this has been done. While FlowDroid is one of the best-known security analyses, this is just one example of what more can be done with MagpieBridge, and our future work includes handling many more analyses.

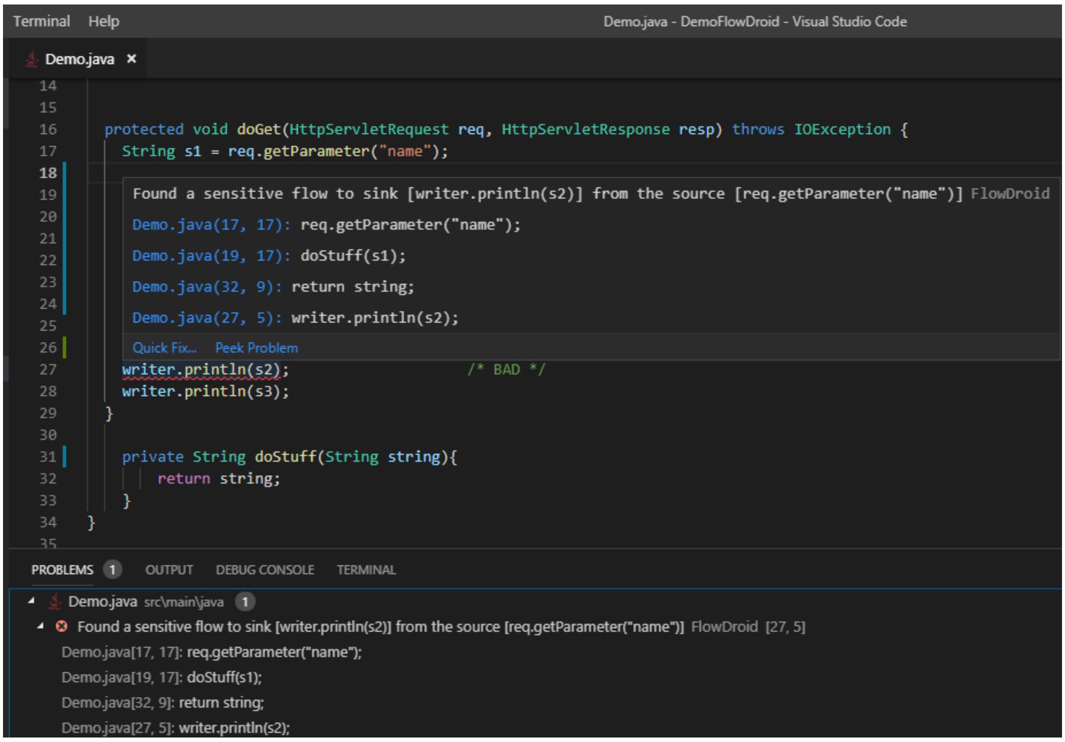


Figure 26 A sensitive data flow reported by FlowDroid in Visual Studio Code

References

1 Android studio. `https://developer.android.com/studio`. Accessed: 2019-01-10.
2 Appscan. `https://www.ibm.com/security/application-security/appscan`. Accessed: 2019-01-10.
3 Atom. `https://atom.io/`. Accessed: 2019-01-10.
4 Clang static analyzer. `https://clang-analyzer.llvm.org/`. Accessed: 2019-01-10.
5 Codesonar. `https://www.grammatech.com/products/codesonar`. Accessed: 2019-01-10.
6 Cppcheck. `http://cppcheck.sourceforge.net/`. Accessed: 2019-01-10.
7 Doop. `http://doop.program-analysis.org/`. Accessed: 2019-01-10.
8 Eclipse. `https://www.eclipse.org/`. Accessed: 2019-01-10.
9 Eclipse lsp4j. `https://projects.eclipse.org/proposals/eclipse-lsp4j`. Accessed: 2019-01-10.

10 Emacs. https://www.gnu.org/software/emacs/. Accessed: 2019-01-10.

11 Facebook infer. https://fbinfer.com/. Accessed: 2019-01-10.

12 Ibm websphere. https://www.ibm.com/cloud/websphere-application-platform. Accessed: 2019-01-10.

13 Intellij. https://www.jetbrains.com/idea/. Accessed: 2019-01-10.

14 Json-rpc. https://www.jsonrpc.org/. Accessed: 2019-01-10.

15 Language server protocol. https://microsoft.github.io/language-server-protocol/. Accessed: 2019-01-10.

16 Monaco. https://microsoft.github.io/monaco-editor/index.html. Accessed: 2019-01-10.

17 Pmd. https://pmd.github.io/. Accessed: 2019-01-10.

18 Pycharm. https://www.jetbrains.com/pycharm/. Accessed: 2019-01-10.

19 Safe. https://github.com/sukyoung/safe. Accessed: 2019-01-10.

20 Sarif specification. https://github.com/oasis-tcs/sarif-spec. Accessed: 2019-01-10.

21 Soot. https://github.com/Sable/soot. Accessed: 2019-01-10.

22 Souffle. https://github.com/oracle/souffle/wiki. Accessed: 2019-01-10.

23 Spotbugs. https://spotbugs.github.io/. Accessed: 2019-01-10.

24 Spyder. https://www.spyder-ide.org/. Accessed: 2019-01-10.

25 Static analysis results: A format and a protocol: Sarif and sasp. http://blogs.grammatech.com/static-analysis-results-a-format-and-a-protocol-sarif-sasp. Accessed: 2019-01-10.

26 Sublime. https://www.sublimetext.com/. Accessed: 2019-01-10.

27 Vim. https://www.vim.org/. Accessed: 2019-01-10.

28 Visual studio code. https://code.visualstudio.com/. Accessed: 2019-01-10.

29 Wala. https://github.com/wala/WALA. Accessed: 2019-01-10.

30 Xanitizer. https://www.rigs-it.com/xanitizer/. Accessed: 2019-01-10.

31 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014. URL: https://doi.org/10.1145/2594291.2594299, doi:10.1145/2594291.2594299.

32 Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spllift: statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 355–364, 2013. URL: http://www.bodden.de/pubs/bmb+13spllift.pdf.

33 Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, pages 1–12, 2009. URL: https://doi.org/10.1145/1572272.1572274, doi:10.1145/1572272.1572274.

34 Hongyi Chen, Ho-fung Leung, Biao Han, and Jinshu Su. Automatic privacy leakage detection for massive android apps via a novel hybrid approach. In *IEEE International Conference on Communications, ICC 2017, Paris, France, May 21-25, 2017*, pages 1–7, 2017. URL: https://doi.org/10.1109/ICC.2017.7996335, doi:10.1109/ICC.2017.7996335.

35 Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. pages 332–343, 2016.

36 Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 307–317, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3092703.3092705, doi:10.1145/3092703.3092705.

37 Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. Cheetah: just-in-time taint analysis for android apps. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 39–42, 2017. URL: `https://doi.org/10.1109/ICSE-C.2017.20`, `doi:10.1109/ICSE-C.2017.20`.

38 Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. Ariadne: Analysis for machine learning programs. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pages 1–10, New York, NY, USA, 2018. ACM. URL: `http://doi.acm.org/10.1145/3211346.3211349`, `doi:10.1145/3211346.3211349`.

39 Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 307–309. ACM, 2010.

40 Stephen Fink and Julian Dolby. Wala–the tj watson libraries for analysis, 2012.

41 Stephen Fink, Julian Dolby, and L Colby. Semi-automatic j2ee transaction configuration. 01 2019.

42 Xi Ge and Emerson R. Murphy-Hill. Manual refactoring changes with automated refactoring validation. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1095–1105, 2014. URL: `https://doi.org/10.1145/2568225.2568280`, `doi:10.1145/2568225.2568280`.

43 Dennis Giffhorn and Gregor Snelting. A new algorithm for low-deterministic security. *International Journal of Information Security*, 14(3):263–287, Jun 2015. URL: `https://doi.org/10.1007/s10207-014-0257-6`, `doi:10.1007/s10207-014-0257-6`.

44 Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

45 Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009. `doi:10.1007/s10207-009-0086-1`.

46 David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 9–14, New York, NY, USA, 2007. ACM. URL: `http://doi.acm.org/10.1145/1251535.1251537`, `doi:10.1145/1251535.1251537`.

47 Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? pages 672–681, 2013.

48 Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: supporting developers in using cryptography. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 931–936. IEEE Press, 2017.

49 Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

50 Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 280–291, 2015. URL: `https://doi.org/10.1109/ICSE.2015.48`, `doi:10.1109/ICSE.2015.48`.

51 Alfonso Murolo, Fabian Stutz, Maria Husmann, and Moira C. Norrie. Improved developer support for the detection of cross-browser incompatibilities. In *Web Engineering - 17th International Conference, ICWE 2017, Rome, Italy, June 5-8, 2017, Pro-*

ceedings, pages 264–281, 2017. URL: `https://doi.org/10.1007/978-3-319-60131-1_15`, `doi:10.1007/978-3-319-60131-1\_15`.

52 Duc-Cuong Nguyen, Dominik Wermke, Yasemin Acar, Michael Backes, Charles Weir, and Sascha Fahl. A stitch in time: Supporting android developers in writing secure code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1065–1077, 2017. URL: `https://doi.org/10.1145/3133956.3133977`, `doi:10.1145/3133956.3133977`.

53 Damien Octeau, Patrick D. McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 543–558, 2013. URL: `https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau`.

54 S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer. The emperor's new security indicators. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 51–65, May 2007. `doi:10.1109/SP.2007.35`.

55 Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL):48:1–48:29, January 2019. URL: `http://www.bodden.de/pubs/sab19context.pdf`, `doi:10.1145/3290361`.

56 Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

57 Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 535–544, New York, NY, USA, 2010. ACM. URL: `http://doi.acm.org/10.1145/1806799.1806876`, `doi:10.1145/1806799.1806876`.

58 Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 210–225, 2013. URL: `https://doi.org/10.1007/978-3-642-37057-1_15`, `doi:10.1007/978-3-642-37057-1\_15`.

59 Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM. URL: `http://doi.acm.org/10.1145/1542476.1542486`, `doi:10.1145/1542476.1542486`.

60 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

61 Christos V. Vrachas. Integration of static analysis results with proguard optimizer for android applications. *Bachelor Thesis*, 2017.

62 Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1329–1341, 2014. URL: `http://doi.acm.org/10.1145/2660267.2660357`, `doi:10.1145/2660267.2660357`.

63 Jing Xie, Bill Chu, Heather Richter Lipford, and John T. Melton. ASIDE: IDE support for web application security. In *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5-9 December 2011*, pages 267–276, 2011. URL: `https://doi.org/10.1145/2076732.2076770`, `doi:10.1145/2076732.2076770`.