AXA: Cross-Language Analysis through Integration of Single-Language Analyses

Tobias Roth
ATHENE
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
roth@cs.tu-darmstadt.de

Julius Näumann
ATHENE
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
julius.naeumann@tu-darmstadt.de

Dominik Helm
University of Duisburg-Essen
ATHENE
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
helm@cs.tu-darmstadt.de

Sven Keidel
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany

Mira Mezini
ATHENE
hessian.AI
Department of Computer Science
Technische Universität Darmstadt
Darmstadt, Germany
mezini@cs.tu-darmstadt.de

ABSTRACT

Modern software is often implemented in multiple interacting programming languages. When performing static analysis of such software, it is desirable to reuse existing single-language analyses to allow access to the results of decades of implementation effort.

However, there are major challenges for this approach. In this paper, we analyze them and present AXA, an architecture that addresses them and enables cross-language analysis by integrating single-language analyses.

To evaluate AXA, we implemented a cross-language points-to analysis for Java applications that interact with native code via Java Native Interface (JNI) and with JavaScript code via Java's `ScriptEngine`. The evaluation shows that AXA enables significant reuse of existing static analyses. It also shows that AXA supports complex interactions and significantly increased recall of reused analyses without compromising precision.

KEYWORDS

Static Analysis, Multi-language

ACM Reference Format:
Tobias Roth, Julius Näumann, Dominik Helm, Sven Keidel, and Mira Mezini. 2024. AXA: Cross-Language Analysis through Integration of Single-Language Analyses. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3691620.3696193

1 INTRODUCTION

Modern software utilizes multiple programming languages [17], leveraging their respective features and strengths for enhanced functionality and efficiency. Scripting languages such as JavaScript are commonly used for frontend, object-oriented languages like C# or Java for backend, and low-level languages such as C, C++, or WebAssembly for performance-critical computations. Code written in multiple languages is composed through *cross-language interactions* like calling foreign-language functions and passing values.

Although multi-language software is widespread, state-of-the-art static analysis frameworks analyze a single language only [7, 8, 25, 29]. Even though some frameworks use summaries for native methods in the Java Class Library, they disregard other cross-language interactions, leading to unsound analysis results [7, 25, 29], or imprecisely over-approximate possible effects of this interaction [5], rendering the analysis results useless.

Cross-language analyses analyze multi-language programs across language boundaries. We identified the following four different approaches for developing cross-language analyses that either:

(1) use handcrafted summaries for foreign function calls,
(2) implement all analyses into a single framework,
(3) reify summaries, or
(4) translate all languages into a common representation.

As we elaborate in Section 5, they all have limitations: (1) handcrafted summaries [7, 10, 12] are time-consuming and prone to becoming outdated; (2) cross-language analyses within a single framework [3, 15, 21, 30] require significant effort to implement for unsupported languages; (3) summary extraction and reification [2, 4, 19, 27] struggles with implicit boundaries and cyclic dependencies; and (4) translating code to a common representation [6, 14, 22, 26, 28] loses high-level information, reducing analysis precision. These problems can be addressed through the integration of existing single-language analyses implemented in different frameworks.

In this paper, we present such an approach: the *AXA* architecture, an **A**rchitecture for **Cross**-language **A**nalysis. By reusing existing analyses, our approach does not require hand-crafted summaries and avoids duplicating implementation effort. Moreover, our approach analyzes each language by language-specialized analyses and, as a result, does not need to reify summaries. Lastly, our approach does not lose precision through translation to an intermediate representation because it analyzes each language in a suitable representation.

AXA comprises the following distinct components: First, AXA augments existing single-language analyses through *language detectors* designed to identify cross-language interactions like foreign function calls and foreign memory accesses. Second, *translators* convert analysis results between different single-language analyses. Third, a central *coordinator* orchestrates the interleaved execution of single-language analyses and propagates results between them. Crucially, the individual analyses are executed in an interleaved manner, exchanging intermediate results as soon as they become available, thus computing a global fixed-point solution. Finally, *connectors* wrap the reused single-language analyses, providing a unified interface for the coordinator to start and resume them. The coordinator is analysis- and framework-independent and reusable across different cross-language analyses. Connectors are framework-specific, but need to be implemented only once for each integrated analysis framework. Translators and language detectors are framework- and analysis-specific. Yet, our evaluation shows that the effort of implementing translators and language detectors is negligible compared to the complexity of state-of-the-art static analysis frameworks.

We evaluate AXA by developing a cross-language points-to analysis for Java interacting with embedded JavaScript code and native code compiled to LLVM bitcode. We implement this cross-language analysis by integrating an existing Java analysis from the OPAL framework [7], a JavaScript analysis from the TAJS framework [8], and an LLVM analysis from the SVF framework [27]. Each of these analyses represents the respective state of the art and their performance and precision has been fine-tuned over many years.

Our evaluation shows that AXA allows the reuse of more than 98% of lines of code of the existing analyses. Furthermore, we validate that our analysis can handle all relevant Java-JavaScript and Java-Native cross-language interaction patterns. For this purpose, we use a hand-crafted benchmark for the cross-language interactions we identified. Finally, we show that our cross-language analysis significantly improves the recall over the integrated single-language analyses without compromising precision.

In summary, our contributions are:

- AXA, an **A**rchitecture for **cross**-language **A**nalysis that integrates existing single-language analyses modularly. AXA allows building upon existing, carefully fine-tuned, state-of-the-art analyses implemented in different frameworks. AXA works for multi-language programs with implicit language boundaries and dynamically generated code and does not sacrifice precision by translating multi-language programs to a common representation.
- A cross-language pointer and call-graph analysis based on AXA for Java with embedded JavaScript and native LLVM code, integrating existing single-language analyses from OPAL, TAJS, and SVF.
- A systematic evaluation that shows that cross-language analyses in AXA require negligible implementation effort compared to state-of-the-art static analysis frameworks, can support all relevant Java-JavaScript and Java-Native interaction patterns, and improve recall over single-language analyses.

```
1  // Java
2  class JavaScriptCalculator {
3    int add(int x, int y) { return calc(x,"+",y); }
4    int calc(int x, String operator, int y){
5      var sem = new ScriptEngineManager();
6      var se = sem.getEngineByName("JavaScript");
7      se.put("x", x); se.put("y", y);
8                // JavaScript
9      se.eval("var result = x "+operator+" y;");
10     int r = se.get("result");
11     return r;
12   }
13 }
```

Listing 1: Example of Java-JavaScript Interaction

2 CROSS-LANGUAGE ANALYSIS CHALLENGES

This section discusses challenges that arise when implementing cross-language analyses by integrating single-language analyses. In total, we identified three challenges that we illustrate at the examples of Java-JavaScript and Java-Native cross-language interactions. Challenge 1 and 3 describe general problems and also apply to other cross-language analysis approaches.

Challenge 1: Detecting Cross-language Interaction
Cross-language analyses first must detect cross-language interactions. Depending on the languages, this is more or less difficult. Listing 1 illustrates a Java-JavaScript cross-language interaction, where Lines 5-6 setup a `ScriptEngine` JavaScript interpreter, Line 7 sets the value of JavaScript variables `x` and `y`, Line 9 evaluates a piece of JavaScript code, and Line 10 retrieves the value of the JavaScript variable `result`. The detection of this cross-language interaction is difficult because the calls to the JavaScript interpreter are not easily distinguishable from regular intra-language calls and could even be located in different Java methods. Furthermore, the JavaScript code itself is dynamically generated while the Java program is running (Line 9). Reliably detecting such cross-language interaction requires precise Java points-to and string analysis information.

Challenge 2: Translating Between Analysis Lattices
Analyses for different languages represent their information with different lattices. For example, OPAL's Java points-to analysis represents pointer information with sets of allocation-sites of objects $\mathcal{P}(\text{JavaAllocSite})$ and TAJS' JavaScript analysis operates on a reduced product of approximations of each JavaScript type $\text{Undef} \times \text{Null} \times \text{Bool} \times \text{Num} \times \text{String} \times \mathcal{P}(\text{JavaScriptAllocSite})$ [8]. Hence, a cross-language analysis needs to translate from the lattice of one analysis to the lattice of another analysis. This translation can be difficult, because one lattice may contain more or less information than the other lattice. For example, OPAL's Java lattice does not

```
1  // Java
2  class Person {
3    String name;
4    void native setName(String name);
5    public static void main(String args[]) {
6      var p = new Person();
7      p.setName(args[0]);
8      var n = p.name;
9    }
10 }
11
12 // C
13 JNIEXPORT void JNICALL Java_Person_setName(
14   JNIEnv *env, jobject obj, jstring newValue
15 ) {
16   jclass personClass = env->GetObjectClass(obj);
17   jfieldID fieldId = env->GetFieldID(personClass,
         "name", "Ljava/lang/String;");
18   env->SetObjectField(obj, fieldId, newValue);
19 }
```

Listing 2: Example of Java-Native Interaction

contain numeric information, in contrast to TAJS' JavaScript lattice. Furthermore, the type of information in different lattices may be incompatible. For instance, OPAL's points-to sets of Java allocation sites do not conform to TAJS' points-to sets of JavaScript allocation sites.

Challenge 3: Cross-Language Fixed-point
To compute the final analysis result, a cross-language analysis needs to run single-language analyses interleaved until no new information can be found and a cross-language fixed-point is reached. For example, Listing 2 illustrates a Java-Native interaction, where the native method `setName` sets field `Person.name` in Line 18. To compute the points-to set for variable `n`, we first need to propagate Java pointer information to determine the points-to set for the arguments of `p.setName`, then propagate native pointer information to determine the points-to set of `newValue`, and finally propagate Java pointer information again. However, computing a cross-language fixed-point is difficult if single-language analyses are not designed to be executed interleaved with other analyses.

In summary, cross-language interactions can be complex and pose numerous challenges to cross-language static analysis. In Figure 3, we explain how AXA tackles these challenges.

3 ARCHITECTURE

This section presents AXA, our **A**rchitecture for **cross**-language **A**nalysis. AXA integrates several single-language analyses. For instance, to analyze multi-language software using Java, JavaScript, and native (C++) code, AXA would integrate existing analyses for Java ($analysis_{\mathcal{J}}$), JavaScript ($analysis_{\mathcal{JS}}$), and native code ($analysis_{N}$) (cf. Figure 1). AXA introduces specialized components to handle each of the challenges discussed in Section 2. We describe these components (cf. Section 3.1) before elaborating on how they interact with each other (cf. Section 3.2). In Section 3.3, we describe requirements to analysis frameworks for integration into AXA.

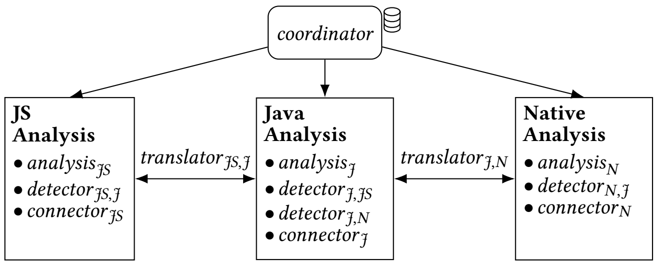

Figure 1: Architecture overview, by the example of three languages, Java (J), JavaScript (JS), and Native (N). Arrows indicate interactions between components.

3.1 Components

Existing single-language analyses are agnostic of cross-language interactions (**Challenge 1**). To address this issue, AXA extends each single-language analysis to be integrated with one *detector* for each language it interacts with. For example, $analysis_{\mathcal{J}}$ is extended with $detector_{\mathcal{J},\mathcal{JS}}$ for JavaScript and $detector_{\mathcal{J},N}$ for native code.

To cope with different internal representations of the results of the integrated single-language analyses (**Challenge 2**), AXA introduces *translators*. For instance, $translator_{\mathcal{JS},\mathcal{J}}$ translates between representations of $analysis_{\mathcal{JS}}$ and $analysis_{\mathcal{J}}$.

To resolve mutual dependencies across language boundaries (**Challenge 3**), AXA introduces a central *coordinator* that executes the integrated analyses in an interleaved manner rather than in a fixed sequence. To allow the coordinator to interact with the integrated analyses, AXA wraps each integrated analysis into a connector (e.g., $connector_{\mathcal{J}}$ for $analysis_{\mathcal{J}}$) that implements a uniform interface.

Overall, AXA is similar to a blackboard architecture [23], with the integrated single-language analyses being the experts and the coordinator and its central store being the blackboard over which the experts indirectly share results to collaboratively solve a multi-language analysis problem.

We now describe each of these components in detail.

3.1.1 Language Detectors. In AXA, a language detector (a) identifies cross-language interactions with a particular other language, (b) gathers information about these interactions, and (c) queries analysis results of the foreign-language analysis.

For example, $detector_{\mathcal{J},\mathcal{JS}}$ in Figure 1 may identify the assignment of the result of a JavaScript call to a Java variable, gather the called function's name and parameter values, and query the results of $analysis_{\mathcal{JS}}$ for the assigned result value.

Recall from Section 2 that cross-language interfaces may differ in complexity. Language interactions can be explicit or implicit. A Java-to-native call can be identified and resolved at the callsite alone. In contrast, resolving native-to-Java interactions requires collecting information over multiple JNI callsites (cf. Lines 16-18 in Listing 2), including analysis of string parameters, similar to resolving reflective invocations, a problem known to be hard to

```
1  function detect(code):
2    foreach statement ∈ code:
3      if stmt contains native method call:
4        methodCall = stmt.getNativeMethodCall()
5        xl = new CrossLanguageInfo()
6        xl.language = native
7        xl.methodName = methodCall.name
8        actualParams = methodCall.getParams
9        xl.paramsPTS = query points-to sets of
             actualParams
10       add xl to coordinator state
11       if stmt.isAssignment:
12         query points-to set of xl from coordinator
13         translate points-to set with translator_{J,N}
14         add translated points-to set to points-to
               set of stmt.lhs
```

Listing 3: Java Native Detector (Pseudocode)

```
1  function detect(code):
2    m = map (allocation-site -> CrossLanguageInfo)
3    foreach stmt ∈ code:
4      if stmt contains ScriptEngine interaction:
5        se = query points-to set of stmt.receiver
6        xl = m(se)
7        if stmt is put:
8          xl.puts += (put-variable-name -> query
               points-to set of put-value)
9        if stmt is eval:
10         xl.eval = stmt
11       if stmt is get:
12         query points-to set for get-variable-name
               in xl from coordinator
13         translate points-to set with translator_{JS,J}
14         add translated points-to set to points-to
               set of stmt.lhs
15   add m to coordinator state
```

Listing 4: JavaScript Detector (Pseudocode)

resolve for static analyses [13]. Detectors must be able to detect these interactions and resolve their parameters.

Listing 3 shows pseudocode for a simplified detector that extends a Java points-to analysis to handle native method calls ($\mathit{detector}_{\mathcal{J},N}$). The detector scans every statement (Line 2), checking whether it contains a native method call (Line 6). If this is the case, it (a) collects the method name (Line 7) and the actual parameters (Line 8), (b) queries and collects the points-to sets for the actual parameters (Line 9), and (c) queries the result of the native call from the coordinator (Line 12). Then, the detector uses $\mathit{translator}_{\mathcal{J},N}$ to translate the result of the native call (Line 13) and adds the translated points-to set to to the Java points-to sets (Line 14).

Listing 4 shows pseudocode for a detector that extends a Java points-to analysis to handle interactions with JavaScript via Java's `ScriptEngine`. As illustrated in Section 2, an interaction of Java with JavaScript via the `ScriptEngine` may span multiple calls to `ScriptEngine` methods like `put` (for setting JavaScript to Java values), `get` (for retrieving JavaScript values), and `eval` (for evaluating JavaScript code). The detector scans the code to gather information about these `ScriptEngine` calls (JS code, JS values set using `put`) over multiple statements (cf. Line 3). Each `ScriptEngine` instance is identified by its allocation site (Line 5). When the detector encounters the `ScriptEngine`'s `put` method, it queries the points-to set of the parameter (Line 8). For the `eval` method, it collects the evaluated JavaScript code (Line 10). When encountering a call to `get`, the detector queries the coordinator for points-to information of the value retrieved via `get` (Line 12). This points-to information is computed by the JavaScript analysis. Subsequently, the detector uses the JavaScript-Java translator $\mathit{translator}_{\mathcal{JS},\mathcal{J}}$ to translate the queried information to the representation expected by $\mathit{analysis}_{\mathcal{J}}$ (Line 13) and adds the result to the analysis state, i.e., it extends the respective points-to set (Line 14). Finally, the map with information of all `ScriptEngine` instances is added to the internal state of the central coordinator (Line 15). This allows the central coordinator to trigger the JavaScript analysis for these interactions.

3.1.2 *Translators.*

A translator transforms analysis results of one analysis lattice into lattice elements of another analysis, and vice versa.

The information and sensitivity can differ between analyses, e.g., pointer information compared to a reduced product over abstractions of types, or flow sensitive vs. flow insensitive results. The result of the translation is either equivalent to the original result or a sound over-approximation of it (recall that lattice representations of different analyses may be incompatible with each other).

Listing 5 shows pseudocode of the translation function of a JavaScript-Java translator for points-to information. Both analyses compute points-to sets for objects, i.e., JavaScript resp. Java objects. These points-to sets are translated element-wise, i.e., the sets are unpacked and the containing analysis-specific points-to elements are translated to the representation of the respective other analysis (Line 9) before being packed again to a points-to set.

3.1.3 *Coordinator and Connectors.*

The central coordinator of AXA orchestrates the interleaved execution of integrated analyses as dictated by dependencies between them. It does so via the `Connector` interface shown in Listing 6. Each single-language analysis integrated into AXA is wrapped by a connector that implements the `Connector` interface in an analysis-specific manner using analysis-specific methods for initialization, configuration, and execution.

```
interface Connector:
  start: ∅ → Result × 𝒫(Dependency)
  resume: ∅ → Result × 𝒫(Dependency)
```

Listing 6: Connector Interface (Pseudocode)

The coordinator starts the first analysis as configured by AXA's user by calling the respective `start` method. The `start` method sets up the corresponding single-language analysis with an initial state; subsequently, it executes the analysis together with the analysis-specific language detectors.

```
function js2javaValue(jsObject):
  jsPointsToSet = jsObject.pointsToSet
  javaPointsToSet = js2JPointsToSet(jsPointsToSet)
  return javaPointsToSet

function js2JPointsToSet(jsPointsToSet):
  result = EmptySet
  foreach jsElement ∈ jsPointstoSet:
    jElement = translatePtsToJS2J(jsElement)
    result.add(jElement)
  return result

function translatePointsToSetJS2J(jsElement):
  information = jsElement.getInformation
  jElement = createNewJElement(information)
  return jElement
```

Listing 5: JavaScript-to-Java Translator (Pseudocode)

Whenever the analysis needs results for a cross-language interaction, the detector for the respective foreign language queries the most up-to-date results from the coordinator (cf. Line 12 in both Listing 3 and Listing 4). Due to potential mutually-recursive language interactions, these results may not be final. Hence, `start` returns an intermediate result and the list of queried results on which this results depends. Accordingly, the return type of the `start` method in the `Connector` interface is a pair of a `Result` value (final or intermediate) and a set of dependencies. Intermediate results are potentially unsound under-approximations to be further refined until sound. The coordinator stores the result and provides it to other analyses upon demand; it also registers the dependencies on other analyses' results and invokes `resume` whenever a (new) value for a registered dependency becomes available.

Method `resume` returns an updated result and any remaining or new dependencies to the coordinator. The coordinator updates the result in its internal state and registers the dependencies. Interleaved execution can go back and forth until a global fixed-point is reached and the computed results become final.

3.2 Component Interaction

We illustrate how AXA's components interact at the example of analyzing method `calc` from Listing 1. The steps for analyzing this method are shown in Figure 2. We assume the integration of existing single-language analyses for Java ($analysis_{\mathcal{J}}$) and JavaScript ($analysis_{\mathcal{JS}}$) into AXA. They are respectively extended by detectors ($detector_{\mathcal{J},\mathcal{JS}}$ and $detector_{\mathcal{JS},\mathcal{J}}$) and wrapped into connectors ($connector_{\mathcal{J}}$ and $connector_{\mathcal{JS}}$), together composing **Java Analysis** and **JS Analysis** respectively (cf. Section 3.1). ($translator_{\mathcal{JS},\mathcal{J}}$) translates between the analyses' representations in both directions.

First, the coordinator starts **Java Analysis** through $connector_{\mathcal{J}}$ (J1). $detector_{\mathcal{J},\mathcal{JS}}$ detects the cross-language interaction of Java with JavaScript through the `ScriptEngine` in Line 6 of Listing 1 (SE_6). It proceeds to collect information about the interaction, such as the values set via the `put` method (Line 7) and the code string passed

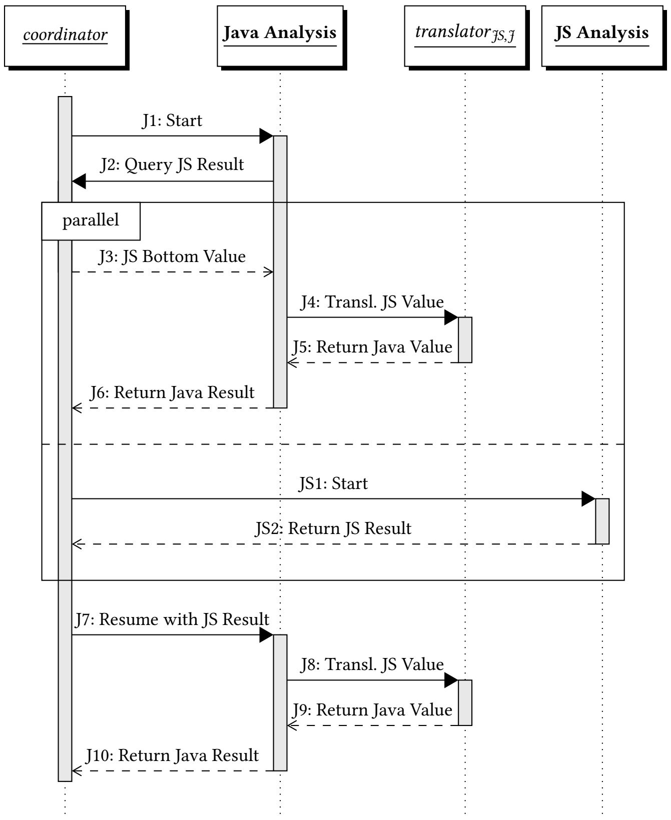

Figure 2: Component Interaction

to the `ScriptEngine`'s `eval` method (Line 9). $detector_{\mathcal{J},\mathcal{JS}}$ also encounters that `get`, which returns a JavaScript value, is invoked on SE_6 (Line 10). It thus queries the coordinator for analysis results of the interaction of SE_6 to determine the value returned (J2).

However, at this point $analysis_{\mathcal{JS}}$ has not yet been started, so no results for SE_6 are available. Hence, the coordinator returns an empty points-to set (i.e., the respective lattice's bottom value) for the JavaScript value `result`, represented in the lattice of $analysis_{\mathcal{JS}}$ (J3). To convert this value to the lattice expected by $analysis_{\mathcal{J}}$ (J4), $detector_{\mathcal{J},\mathcal{JS}}$ uses the JavaScript-Java translator $translator_{\mathcal{JS},\mathcal{J}}$. It then continues with the analysis (J5). Using the translated empty points-to set, $analysis_{\mathcal{J}}$ can complete its computation; it sets the points-to set of the Java variable `r` to empty (cf. Line 10) and returns a result to the coordinator (J6). This result is not final as it is based upon the bottom value from (J3). Thus, $analysis_{\mathcal{J}}$, alongside its result, returns to the coordinator a dependency on that value.

Concurrently, the coordinator has started execution of **JS Analysis** via $connector_{\mathcal{JS}}$ (JS1). For simplicity, we assume that the points-to set of the JavaScript values `x` and `y` can be computed directly, i.e., the figure does not illustrate the analysis of the accesses from JavaScript to Java occurring in Line 9. Thus, $analysis_{\mathcal{JS}}$ completes and returns its result to the coordinator (JS2). The coordinator now resumes **Java Analysis** via $connector_{\mathcal{J}}$ and makes the newly computed JavaScript analysis result available to **Java Analysis** (J7).

$detector_{J,JS}$ invokes $translator_{JS,J}$ to convert this JavaScript points-to set to the lattice expected by $analysis_J$ (J8, J9). $analysis_J$ uses this updated result to update the points-to set of Java variable `r` (cf. Line 10) and returns the final result to the coordinator (J10).

3.3 Analysis Integration Requirements

After we have elaborated how AXA enables to integrate analyses, we will now describe what is required to integrate an existing analysis framework into AXA. First, AXA requires that an integrated analysis:

- has its own fixed-point solver and
- starts with the bottom value of its respective lattice (i.e., is an *optimistic* analysis)

Second, to integrate a static analysis framework into AXA, one:

- adapts the lattice of the static analysis framework,
- adapts the fixed-point solver of the static analysis framework,
- extends the analyses with detectors,
- wraps each integrated analysis with a connector, and
- writes translators to translate between analysis representations

Third, additional minor changes may be required depending on the framework. This may, e.g., include enabling extensibility through making private fields or methods public.

Not all of these steps are necessary for every framework or language pair, e.g., if native method calls are already recognized by a Java analysis, no further detector for this cross-language interaction may be necessary. The integration process will be described in further detail using a case study in Section 4.1.

4 EVALUATION

Our evaluation[1] aims to answer the following research questions:

RQ1 What effort is needed to integrate existing analyses, and to what extent can they be reused?

RQ2 Can the implementation handle all identified kinds of cross-language interaction?

RQ3 Can AXA multi-language analyses improve recall over the integrated single-language analyses without compromising precision?

4.1 Reusability of Single-Language Analyses

To showcase the required effort to integrate existing static analysis frameworks into AXA (**RQ1**), we implemented AXA as an extension of the OPAL static analysis framework[2] [7] and used it to implement a cross-language points-to analysis for analyzing Java programs that embed JavaScript code via the Java `ScriptEngine` and call native methods via the Java Native Interface. We chose a points-to analysis as our use case, because it is a prerequisite for many other analyses, e.g., advanced call-graph analyses, and is a complex analysis of its own that is not trivially compositional.

OPAL's blackboard architecture enables executing multiple decoupled analysis modules in an interleaved manner [7, 9]. Analyses exchange their results via a central data store called the *blackboard* until a global fixed-point is reached. OPAL's blackboard provides all features needed to implement AXA's coordinator. Our cross-language points-to analysis reuses respective single-language analyses from OPAL, TAJS[3] [8], and SVF[4] [27], three state-of-the-art static analysis frameworks for Java, JavaScript, and LLVM bitcode respectively. The analysis information differs between the analyses: pointer, call-graph, and string information for Java and LLVM, and a reduced product over abstractions of each JavaScript type. Furthermore, the Java and LLVM analyses are flow-insensitive, whereas the JavaScript analysis is flow-sensitive. In the following, we elaborate on what we did to integrate these analyses; Table 1 quantifies the effort in terms of LOCs (lines of code).

Java Analysis. To analyze Java code, we reused the points-to analysis of OPAL. Since the latter already uses the blackboard architecture also used to implement AXA, we did not need to implement a connector or adapt its fixed-point algorithm. OPAL already identifies native method calls in its call graphs, thus, no detector or lattice extension is required for native method calls. To analyze Java-to-JavaScript interactions, we extended OPAL's points-to analysis with $detector_{J,JS}$ for JavaScript code that is executed via the Java `ScriptEngine`. $detector_{J,JS}$ handles the `ScriptEngine` methods `eval`, `put`, and `get` and was implemented as an OPAL points-to analysis module, comprising 836 LOCs. Furthermore, we extended OPAL's data store with a new lattice for interactions with JavaScript via Java's ScriptEngine (60 LOCs). The lattice elements aggregate all relevant information such as the allocation site of the particular `ScriptEngine`, the JavaScript code that the engine evaluates via `eval`, and the JavaScript values set via `put`. Overall, we added 896 LOCs to the OPAL points-to analysis, corresponding to 0.6% of the original OPAL code (155 101 LOCs), i.e., 99.4% of the code was reused. Most of the additional code handles the inherent complexity of analyzing the `ScriptEngine` interface, which is necessary for any cross-language interaction of this interface, i.e., not overhead introduced by AXA.

JavaScript Analysis. To analyze JavaScript code, we integrated the TAJS framework. TAJS has a fixed-point solver operating with a worklist algorithm and an intermediate representation on which the fixed-point solver runs the transfer functions implemented in the `NodeTransfer` class. This class plays the visitor role in a visitor pattern, visiting the elements of the intermediate representation, e.g., nodes for variable declarations, method calls, or property/value reads and writes.

We implemented the Java detector, $detector_{JS,J}$, in a class that inherits from TAJS' `NodeTransfer`, extending the behavior of five visitor methods that are responsible for handling method calls and variable/property reads and writes. $detector_{JS,J}$ comprises 166 LOCs, including code to access AXA's state via the connector $connector_{JS}$. $detector_{JS,J}$ needs to access AXA's state in order to query the actual points-to sets of Java values that are accessed in JavaScript (cf. Figure 3). To make the analysis use $detector_{JS,J}$, with the original and extended visitor methods instead of `NodeTransfer`, we modified one line (counted as 2 changed LOCs, 1 deletion and 1 addition) that sets the transfer functions in TAJS. We added new node types to TAJS' internal representation to represent Java objects and types

[1] Artifact available at https://doi.org/10.5281/zenodo.13364690
[2] https://opal-project.de
[3] https://www.brics.dk/TAJS/
[4] https://svf-tools.github.io/SVF/

Analysis	Detector	Lattice	Solver	Connector	Translator (to Java)	Total (Framework)
Java	836 (JS), 0 (Native)	60 (JS)	0	0	-	896 (0.6% of OPAL)
JavaScript	166 + 2	90+14	2	452	137	863 (1.4% of TAJS)
Native	328	107	16	1 025	16	1 492 (1.9% of SVF)

Table 1: Extensions and Changes of Single-Language Analyses for Integration into AXA

(90 LOCs). To support these new node types, we had to extend the existing visitor classes (14 LOCs). Finally, we removed 2 LOCs to prevent TAJS' fixed-point solver from clearing the initial state. This allows $connector_{\mathcal{JS}}$ to set TAJS' initial state before starting the analysis. This initial state contains, e.g., JavaScript values set by `ScriptEngine`'s `put` method. Finding the place where the fixed-point solver clears the initial state was straightforward, because TAJS' fixed-point solver is a standard work-list algorithm.

To recap, for integrating TAJS into AXA, we added or changed a total of 274 LOCs. Together with $connector_{\mathcal{JS}}$ that wraps TAJS to interact with AXA's coordinator (452 LOCs), and $translator_{\mathcal{JS},\mathcal{J}}$ that translates between the points-to representations of TAJS and OPAL (137 LOCs), this totals to 863 LOCs changed in or added to AXA and TAJS to integrate the points-to analyses of OPAL and TAJS. These additions and changes (less than 1 000 LOCs) are negligible compared to TAJS overall code size (60 321 LOCs), just 1.4%.

Native Analysis. We use SVF's Andersen-style points-to analysis in $Solver_{SVF}$ to implement a fixed-point analysis on points-to sets. We extend SVF's points-to-set lattice by adding additional node types to SVF's Pointer Assignment Graph (PAG) to represent Java allocation sites. The solver, outlined in Listing 7, executes $detector_{N,\mathcal{J}}$ and Andersen until a fixed-point is reached.

```
Andersen()
while (any points-to set has changed):
  detector$_{N,J}$()
  Andersen()
```

Listing 7: SVF Fixed-Point Solver (Pseudocode)

We implemented $detector_{N,\mathcal{J}}$ (328 LOCs) to detect and resolve Java method calls and field accesses. SVF's points-to analysis was used to recover class, field, and method names from JNI interactions as shown in Listing 2. Our changes to the SVF lattice comprise 107 LOCs. Implementing $Solver_{SVF}$ (16 LOCs) was trivial, while $detector_{N,\mathcal{J}}$ required the most effort. $connector_{N}$ (1 025 LOCs) communicates points-to sets for call-site parameters and field writes to $analysis_{\mathcal{J}}$ and receives points-to sets for method return values and field reads. $connector_{N}$ consists mostly of standard JNI code that makes SVF (implemented in C++) accessible to OPAL. $translator_{\mathcal{J},N}$ (16 LOCs) finally manages a mapping between Java allocation sites and SVF PAG nodes to translates between the points-to sets of OPAL and SVF. This totals 1 492 changed or added LOCs, 1.9 % of SVF's 76 801 LOCs.

Summary. The summary of the changed and added LOCs in Table 1 clearly demonstrates that AXA enables reusing existing work to a significant extent. While the static analysis frameworks we integrated were not built for being reused within AXA, the required changes to integrate them were minor compared to the complexity of the reused analyses, allowing AXA to tap into analyses developed, tested, and improved over years by experts in the respective languages and analyses. We were able to repurpose SVF and TAJS, two frameworks with largely different implementations and purposes, for cross-language analysis with AXA.

The design of TAJS, with its worklist algorithm and transfer functions following the visitor pattern, made it easier to extend compared to SVF. This design allowed incorporating cross-language values into TAJS' lattice during the main solver loop. We were able to preset the analysis state before execution and extend the analysis results during solver iterations by adding new visitors and extending existing ones for the added lattice elements. Conversely, SVF's points-to analysis is not inherently extensible. In order to integrate cross-language values into SVF's points-to analysis, we had to develop a custom fixed-point solver where SVF's points-to analysis is executed at every step, with cross-language values added to the initial state of the succeeding points-to analysis step.

Our case study shows the importance of extensible software design. Support for cross-language analyses has so far not been a primary objective in the design of static analysis frameworks. However, adapting static analyses for an integration into AXA increases their relevance and allows them to be reused for analyses across language borders. We encourage developers of static analyses to consider support for cross-language analyses in their designs. To simplify integration into AXA, the analyses should provide: 1) the ability to preset the analysis state, 2) the capability to extend the analysis lattice, and 3) the flexibility to extend analysis behavior to recognize cross-language interactions during solver execution.

▪ *Changes required to integrate existing static analyses into AXA are minor compared to the complexity of these analyses. Reusing existing analyses in AXA allows access to years of research and development, thus saving major effort to reimplement analyses with the same soundness and precision level as existing state of the art analyses.*

4.2 Handling Cross-Language Interactions

Answering **RQ2**, whether our implementation can handle all identified kinds of cross-language interaction, requires a ground truth of multi-language code. To the best of our knowledge, no ground-truth benchmark for cross-language analysis of Java/JavaScript multi-language programs exists. Regarding Java/Native, the microbenchmark of Lee et al. [16] consists of Android applications with JNI calls, but the reused analyses from OPAL are not suited for Android applications. Related work by Li et al. [18] also uses a hand-crafted test suite to validate their cross-language analysis, but targets multi-language programs in Java and Python.

Thus, we created AXA-Benchmark, the first cross-language-analysis benchmark featuring non-Android-Java programs that execute JavaScript and interact with native code. The benchmark was created with the following methodology. One author searched for literature initially in Google Scholar using keywords related to cross-language analysis, e.g., "cross-language static analysis" and "multi language static analysis". We studied the resulting peer-reviewed papers and their references and categorized them in Section 5. From this study, we derived the benchmark categories. The individual test cases were derived from both studying real-world code and based on the authors multi-year experience with static analyses. Test cases were implemented by two authors that checked and supplemented each other's work. To the best of our knowledge, we considered all relevant cases of cross-language interaction patterns. The small number of test cases for the JNI Java-to-native interface is due to the simplicity of this interface.

4.2.1 *Benchmark.*

AXA-Benchmark[5] contains a total of 56 test cases, grouped into five categories of cross-language interaction patterns that we identified:

- **Unidirectional Execution:** Java code executes either native or JavaScript code. Native methods are called explicitly, JavaScript code is executed via the `eval` method of Java's `ScriptEngine`.
- **Interleaved Execution:** Java code executes JavaScript or native code, which in turn calls static or virtual Java methods. In native code, Java virtual methods are called on objects instantiated using JNI or method parameters passed from Java. In JavaScript, Java virtual methods are called on Java objects instantiated in JavaScript code or passed via the `ScriptEngine`'s `put` method.
- **Mutual Recursion:** Advanced test cases involving recursion: A Java method `f` calls a native method or executes JavaScript code, which in turn invokes `f` recursively. Such recursion creates cyclic dependencies between the analyses (cf. Section 2).
- **Unidirectional State Access:** Java code that accesses or manipulates the state of native or JavaScript code or vice-versa. Native code accesses Java's state via field writes. For JavaScript, state access is implemented through the Java `ScriptEngine`'s `put` and `get` methods and through reads and writes to Java fields, both instance and static, from JavaScript code, respectively.
- **Bidirectional State Access:** Advanced test cases that combine the aforementioned state accesses for accesses from Java to native or JavaScript code and vice versa. Cases include independent unidirectional state accesses, as well as state accesses that pass the same references in both directions, potentially creating cyclic dependencies (cf. Section 2).

Each test case is implemented as an executable Java class. That class either creates a `ScriptEngine` instance to interact with locally embedded JavaScript code through the methods `put`, `eval`, and `get` or invokes native methods. The native methods are available in executable LLVM files.

[5]https://github.com/stg-tud/AXABenchmark

```
1   @PointsToSet(line = 10,
2     mustAllocSites = {@AllocSite(line=8)}
3   )
4   void javaFunction(){
5     sem = new ScriptEngineManager();
6     se = sem.getEngineByName("JavaScript");
7     se.put("jThis", this);
8     this.myfield = new Object();
9     se.eval("var x = jThis.myfield;");
10    Object o = se.get("x");
11  }
12  Object myfield;
```

Listing 8: AXA-Benchmark Example Test Case

Category	Passed/Test	
	JavaScript	Native
Unidirectional Execution	15 / 16	2 / 2
Interleaved Execution	5 / 5	7 / 7
Mutual Recursion	2 / 4	1 / 1
Unidirectional State Access	6 / 6	4 / 4
Bidirectional State Access	7 / 9	2 / 2
Sum	35 / 40	16 / 16

Table 2: Benchmark Results

Furthermore, each category includes intra- and inter-procedural test cases, i.e., executing Java `ScriptEngine`'s `put`, `eval` and `get` methods located in a single Java method or across multiple methods.

We manually annotated the benchmark with expected points-to sets of references affected by cross-language interactions. The annotations specify expected allocation sites for method parameters and local variables (fields are validated by assignment to local variables). Listing 8 shows a simplified example of a test case with the corresponding annotation. The annotation states that the Java variable `o` (defined in Line 10) points to the Java object instantiated in Line 8. To pass a test case, an implementation has to match the annotated points-to set exactly, thereby verifying its ability to correctly handle the cross-language interaction being tested.

4.2.2 *Results.*

Table 2 shows AXA's results on the benchmark: AXA can handle all cross-language interaction patterns, passing 35/40 of Java/JavaScript test cases and 16/16 of Java/native test cases. The higher number of JavaScript test cases is a result of the more complex interface.

Two test cases in the Mutual Recursion category fail due to a cross-language interaction inside a JavaScript function. Our implementation of $detector_{\mathcal{JS},\mathcal{J}}$ only detects cross-language interactions in the global scope of JavaScript, which is a deliberate limitation to constrain the scope of our implementation. Two test cases in the Bidirectional State Access category fail due to a limitation of our implementation where $translator_{\mathcal{JS},\mathcal{J}}$ assigns the Java type `java.lang.Object` to JavaScript objects. This leads to problems in case they are passed back to JavaScript but can be solved by

	Java/JS		Java/Native	
Points-To Analysis	**OPAL**	**AXA**	**OPAL**	**AXA**
False Negatives	169	50	34	3
False Positives	1	1	0	0
Precision	99.7 %	99.8 %	100.0 %	100.0 %
Recall	66.1 %	90.6 %	54.7 %	96.1 %

Table 3: Precision And Recall of Points-To-Sets

introducing a mapping between Java and JavaScript points-to set elements in $connector_{JS}$.

In summary, AXA passed most test-cases of all identified kinds of cross-language interaction. This shows that our approach is applicable to relevant, dissimilar cross-language interaction patterns. Failed test cases revealed opportunities for improving the implementation through further engineering effort, but do not affect AXA's applicability.

▪ *AXA can handle most test cases of all identified kinds of cross-language interaction.*

4.3 Precision and Recall

To answer **RQ3**, whether AXA can analyses can improve recall, we compared the recall and precision of our cross-language points-to analysis against OPAL's single-language points-to analysis. For **RQ2**, we manually annotated parameters and local variables impacted by cross-language interactions, representing only a subset of all reference variables in AXA-Benchmark's codebase. For a broader assessment of recall and precision, we implemented an automated instrumentation which records values for all parameters and definition sites, comparable to related work [20]. The instrumentation logs runtime information of reference variables (type and instance id) together with the location of the instrumentation. The recorded data provides a dynamically-recorded ground truth for points-to-sets of all possible definition sites in AXA-Benchmark.

We validated the points-to sets obtained from both analyses, OPAL's Java-only analysis and AXA's cross-language analysis, against this ground truth. The results are presented in Table 3, with Java/JS and Java/Native test cases measured separately. The code required for the `ScriptEngine` instantiation resulted in comparably more definition sites for Java/JS test cases. The cross-language analysis significantly reduces the number of false negatives in comparison to OPAL's single language points-to analysis. AXA was able to significantly increase recall for both language combinations without increasing the number of false positives.

Our cross-language analysis inherits one false positive from OPAL's Java points-to analysis, which incorrectly provides a points-to set for an unreachable definition site.

▪ *AXA increases the recall of the integrated analyses without compromising precision.*

4.4 Threats to Validity

In measuring the reusability of existing analyses, we limit our metric to added/changed LOCs (lines of code). To fully gauge the effort of integrating an anlysis with AXA, an extensive representative user study would be required. This, however, exceeds the scope of this paper. Measuring LOCs still reveals important insights into how far existing analysis code can be reused.

AXA-Benchmark contains testcases categorized into several interaction patterns, yet we do not assert the completeness of these patterns. With the benchmark based on a literature study and checked by a second author, we are confident to have identified relevant patterns.

We have not yet evaluated AXA on real-world code, except for the Android app Droidzebra [6]. This app consists of 50 Java- and 5 C-files. The analysis of Droidzebra took less than ten seconds, finding and handling four Java-to-native and two native-to-Java calls. A systematic evaluation of AXA on real-world benchmarks remains for future work.

We also did not compare our implementation of AXA with other state-of-the-art analyses, because to the best of our knowledge, no state-of-the-art analysis can analyze both Java-Native interaction and Java-JavaScript interaction via the Java `ScriptEngine`. A direct comparison with the approach by Lee et al. [16], which was evaluated through a microbenchmark consisting of Android applications with JNI calls was not possible because of limitations within OPAL's analyses. We instead evaluated our implementation against OPAL's single-language analysis, revealing how AXA cross-language analyses improve recall.

5 RELATED WORK

In this section, we discuss approaches to cross-language analysis.

Using Handcrafted Summaries for Foreign Function Calls. Many frameworks like OPAL [7], Soot [12], and WALA [10] include *handcrafted summaries* of native methods in the Java Class Library. However, hand-crafting summaries is time-consuming and summaries become invalid when summarized methods change. As a result, this approach only applies to foreign code in widely-used libraries that change only seldomly and not to multi-language application code. In contrast, AXA does not require summaries because it reuses single-language analyses for all analyzed languages.

Cross-Language Analyses Implemented in a Single Framework. Some cross-language analyses are implemented within a *single static-analysis framework* [3, 15, 21, 30]. For example, the WALA analysis framework supports single-language Java and JavaScript analyses. While WALA does not directly support for cross-language interactions, Lee et al. [15] combined two single-language analyses in WALA to a cross-language analysis called HybriDroid. Implementing a cross-language analysis in a single framework achieves a high level of integration and promises high precision and soundness. However, when an analysis does not exist in the framework of choice, it needs to be reimplemented; existing analyses outside the framework cannot be reused. Such reimplementation results in the duplication of possibly years of research and development effort, in which precision, soundness, and performance have been carefully fine-tuned. For instance, the JavaScript analysis framework TAJS has been actively developed for more than 13 years [1, 8, 11, 24]. Furthermore, porting existing analyses to other frameworks can also be

[6]https://f-droid.org/en/packages/com.shurik.droidzebra/

difficult, because frameworks follow different analysis paradigms and styles. In contrast, AXA allows reusing analyses implemented in different frameworks, saving the effort of reimplementing analyses within a single framework and allowing access to the results of years of development effort.

Summary Reification. Yet other approaches implement cross-language analyses by extracting *summaries* for guest-language code and *reifying* them as code of a host-language. For example, Sui and Xue [27] extract summaries for native JNI code with Facebook's Infer analyzer [4], reify them as Java code, insert the code at the native call site, and analyze the residual Java program with FlowDroid [2]. Lyons and Becaj [19] use a similar approach to implement a cross-language taint analysis for Python and C using Infer/Quandry to extract C summaries and Pyt for Python taint analysis. Lee et al. [16] analyze multi-language programs written in Java and native code by translating analysis summaries of native functions to Java code. Summary reification allows reusing existing analyses, but it also has several limitations:

- First, summary reification only works for cross-language interactions with explicit language boundaries. Specifically, calls from the host to the guest language must be easily detectable before running the host analysis to insert the reified code at the correct place in the host program (cf. [16, Section 6.4.]). However, cross-language interactions between Java and JavaScript are implicit since Java to JavaScript calls are not easily distinguishable from intra-language Java calls and may contain dynamic strings, making summary-reification unsuitable. AXA solves this problem by detecting cross-language calls on-the-fly while the analysis is running.
- Second, if there are cyclic dependencies across language borders, they cannot be trivially summarized. This can be dealt with by full reanalysis until a fixed-point is reached, but this requires significant amounts of recomputation. AXA instead directly computes a cross-language fixed-point, avoiding recomputation where possible.
- Third, summaries of the guest language must be soundly reifiable as code of the host language. However, this may not always be possible if the guest language has complex or incompatible semantics. For example, C's manual memory management, unsafe casts, and pointer arithmetic are not easily reifiable within Java code. These features are over- or under-approximated when reifying summaries, leading to imprecision and unsoundness. In contrast, AXA delegates the handling of complex language features to single-language analyses, retaining their precision and soundness.

Translation into a Common Representation. Finally, some approaches *translate* all code of a multi-language program into a unified representation, then analyze this representation. JScan [6] translates C, C++, and Java to LLVM intermediate representation (LLVM IR) [14], while Lara [28] translates C, C++, Java, and JavaScript to a custom object-oriented language. To create library dependency graphs, Shatnawi et al. [26] translate all components of multi-language Java Enterprise Edition applications into a common meta-model. LiSa's [22] front-ends translate all parts of a program written in different languages into a common internal representation. However, translating multi-language programs into a unified representation loses high-level information from the source languages, rendering these analyses less precise and less sound. For example, single-language analyses for Java typically handle reflection explicitly for better precision and soundness. By translating Java code to an intermediate representation like LLVM IR, reflection is now handled generically by an LLVM analyzer, leading to reduced precision and soundness. In contrast, AXA analyzes each language individually without a need for translation and hence avoiding a precision and soundness penalty.

6 CONCLUSION

In this paper, we proposed AXA, an approach that enables implementing cross-language analyses by integrating existing tried and tested single-language analyses from different frameworks. We implemented AXA in the static analysis framework OPAL and used OPAL's blackboard as the coordinator.

To evaluate AXA, we implemented a cross-language points-to analysis for Java applications that interact with JavaScript code via Java's `ScriptEngine` and with native code via the Java Native Interface (JNI). To this end, we integrated the points-to analysis of OPAL for Java, the JavaScript analysis TAJS, and the points-to analysis of SVF for LLVM bitcode. AXA allowed substantial reuse (over 98%) of the integrated analyses. Additionally, we created AXA-Benchmark, a hand-crafted benchmark of relevant cross-language interaction patterns between Java, JavaScript, and native code. Our evaluation showed that AXA supports complex interactions and significantly increased the recall of the integrated analyses without compromising precision. Last but not least, we established requirements for integrating further static analyses. This can guide developers during the development of static analysis frameworks to ease integration into AXA and, as a result, increase the relevance of the integrated static analyses as they can then be used on multi-language software.

In the future, we will develop further analyses, e.g., data-flow analyses, on top of AXA as well as integrate existing analysis tools for more languages, e.g., WebAssembly. Furthermore, we will extend our implementation to deal with other cross-language interfaces, like remote method invocations.

ACKNOWLEDGMENTS

This work was supported by the DFG as part of CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

REFERENCES

[1] Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 17–31. https://doi.org/10.1145/2660193.2660214

[2] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. 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, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[3] Achim D. Brucker and Michael Herzberg. 2016. On the Static Analysis of Hybrid Mobile Apps - A Report on the State of Apache Cordova Nation. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9639)*, Juan Caballero, Eric Bodden, and Elias Athanasopoulos (Eds.). Springer, 72–88. https://doi.org/10.1007/978-3-319-30806-7_5

[4] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2009. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 289–300. https://doi.org/10.1145/1480881.1480917

[5] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3444)*, Shmuel Sagiv (Ed.). Springer, 21–30. https://doi.org/10.1007/978-3-540-31987-0_3

[6] Andrea Fornaia, Stefano Scafiti, and Emiliano Tramontana. 2019. JSCAN: Designing an Easy to use LLVM-Based Static Analysis Framework. In *28th IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises, WETICE 2019, Naples, Italy, June 12-14, 2019*, Sumitra Reddy (Ed.). IEEE, 237–242. https://doi.org/10.1109/WETICE.2019.00058

[7] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. 2020. Modular collaborative program analysis in OPAL. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 184–196. https://doi.org/10.1145/3368089.3409765

[8] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5673)*, Jens Palsberg and Zhendong Su (Eds.). Springer, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

[9] Sven Keidel, Dominik Helm, Tobias Roth, and Mira Mezini. 2024. A Modular Soundness Theory for the Blackboard Analysis Architecture. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14577)*, Stephanie Weirich (Ed.). Springer, 361–390. https://doi.org/10.1007/978-3-031-57267-8_14

[10] Rahul Krishna, Raju Pavuluri, Saurabh Sinha, Divya Sankar, Julian Dolby, and Rangeet Pan. 2023. Towards Supporting Universal Static Analysis using WALA. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[11] Erik Krogh Kristensen and Anders Møller. 2019. Reasonably-most-general clients for JavaScript library analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 83–93. https://doi.org/10.1109/ICSE.2019.00026

[12] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Vol. 15.

[13] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study. In *2017 IEEE/ACM 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE'17)*. IEEE, 507–518.

[14] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[15] Sungho Lee, Julian Dolby, and Sukyoung Ryu. 2016. HybriDroid: static analysis framework for Android hybrid applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 250–261. https://doi.org/10.1145/2970276.2970368

[16] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening Horizons of Multilingual Static Analysis: Semantic Summary Extraction from C Code for JNI Program Analysis. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* IEEE, 127–137. https://doi.org/10.1145/3324884.3416558

[17] Wen Li, Na Meng, Li Li, and Haipeng Cai. 2021. Understanding Language Selection in Multi-language Software Projects on GitHub. In *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021.* IEEE, 256–257. https://doi.org/10.1109/ICSE-Companion52605.2021.00119

[18] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2513–2530.

[19] Damian M. Lyons and Dino Becaj. 2021. A Meta-level Approach for Multilingual Taint Analysis. In *Proceedings of the 16th International Conference on Software Technologies, ICSOFT 2021, Online Streaming, July 6-8, 2021*, Hans-Georg Fill, Marten van Sinderen, and Leszek A. Maciaszek (Eds.). SCITEPRESS, 69–77. https://doi.org/10.5220/0010543800690077

[20] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. 2001. Dynamic Points-to Sets: A Comparison with Static Analyses and Potential Applications in Program Understanding and Optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE '01)*. Association for Computing Machinery, New York, NY, USA, 66–72. https://doi.org/10.1145/379605.379671

[21] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2021. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis - 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12913)*, Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi (Eds.). Springer, 323–345. https://doi.org/10.1007/978-3-030-88806-0_16

[22] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. 2023. LiSA: a generic framework for multilanguage static analysis. In *Challenges of Software Verification*. Springer, 19–42.

[23] Allen Newell. 1962. *Some problems of basic organization in problem-solving programs*. Rand Corporation.

[24] Benjamin Barslev Nielsen and Anders Møller. 2020. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.16

[25] Joanna C. S. Santos and Julian Dolby. 2022. Program analysis using WALA (tutorial). In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1819. https://doi.org/10.1145/3540250.3569449

[26] Anas Shatnawi, Hafedh Mili, Manel Abdellatif, Yann-Gaël Guéhéneuc, Naouel Moha, Geoffrey Hecht, Ghizlane El Boussaidi, and Jean Privat. 2019. Static code analysis of multilanguage software systems. *arXiv preprint arXiv:1906.00815* (2019).

[27] Yulei Sui and Jingling Xue. 2020. Value-Flow-Based Demand-Driven Pointer Analysis for C and C++. *IEEE Trans. Software Eng.* 46, 8 (2020), 812–835. https://doi.org/10.1109/TSE.2018.2869336

[28] Gil Teixeira, João Bispo, and Filipe F. Correia. 2021. Multi-language static code analysis on the LARA framework. In *SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021*, Lisa Nguyen Quang Do and Caterina Urban (Eds.). ACM, 31–36. https://doi.org/10.1145/3460946.3464317

[29] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, Stephen A. MacKay and J. Howard Johnson (Eds.). IBM, 13.

[30] Dongjun Youn, Sungho Lee, and Sukyoung Ryu. 2023. Declarative static analysis for multilingual programs using CodeQL. *Softw. Pract. Exp.* 53, 7 (2023), 1472–1495. https://doi.org/10.1002/spe.3199