Received: 19 August 2022 | Revised: 20 February 2023 | Accepted: 21 February 2023

DOI: 10.1002/spe.3199

RESEARCH ARTICLE

WILEY

Declarative static analysis for multilingual programs using CodeQL

Dongjun Youn[1] | **Sungho Lee**[2] | **Sukyoung Ryu**[1]

[1]School of Computing, KAIST, Daejeon, South Korea

[2]Department of Computer Science and Engineering, Chungnam National University, Daejeon, South Korea

Correspondence
Sukyoung Ryu, School of Computing, KAIST, Daejeon 34141, South Korea.
Email: sryu.cs@kaist.ac.kr

Funding information
National Research Foundation of Korea (NRF), Grant/Award Numbers: 2022R1A2C200366011, 2021R1A5A1021944; Institute for Information & Communications Technology Promotion (IITP), Grant/Award Number: 2022-0-00460; Samsung Electronics Co., Ltd, Grant/Award Number: G01210570

[Correction added on 13 April 2023, after first online publication: the funder details have been added in the Funding Information and Acknowledgement sections.]

Summary

Declarative static program analysis has become one of the widely-used program analysis techniques. Declarative static analyzers perform three steps: creating databases of facts from program source code, evaluating rules to generate new facts, and running queries over facts to extract all information related to specific properties via query systems. Declarative static analyzers can easily target diverse programming languages by modifying only databases and rules for new languages. Because query systems are independent of programming languages, they are reusable for new languages. However, even when declarative analyzers support multiple programming languages they do not currently support the analysis of multilingual programs written in two or more programming languages. We propose a systematic methodology that extends a declarative static analyzer supporting multiple languages to support multilingual programs as well. The main idea is to reuse existing components of the analyzer. Our approach first generates a merged database of facts, consisting of multiple logical language spaces. It allows existing language-specific rules to derive new facts for the corresponding language from the facts in the corresponding language space. Then, it defines language-interoperation rules that handle the language interoperation semantics. Finally, it uses the same query system to get analysis results leveraging the language interoperation semantics. We develop a proof-of-concept declarative static analyzer for multilingual programs by extending CodeQL, which can track dataflows across language boundaries. Our evaluation shows that the analyzer successfully tracks dataflows across Java-C and Python-C language boundaries and detects genuine interoperation bugs in real-world multilingual programs.

KEYWORDS

declarative static program analysis, JNI programs, multilingual programs, Python-C programs

1 | INTRODUCTION

Advances in modern declarative logical specification languages such as Datalog, Soufflé[1] and QL[2] have facilitated *declarative static analyses*. A declarative static analysis specifies rules for "what" the analysis results in rather than defining

Abbreviations: API, Application Programming Interface; DB, DataBase; FFI, Foreign Function Interface; FN, False Negative; FP, False Positive; JNI, Java Native Interface; LOC, Line of Code; ML, Machine Learning; QL, Query Language; RQ, Research Question.

algorithms for "how" the analysis is performed. Typically, it completes three steps. First, it transforms a program source code into a database containing *facts* that are tuples of values. Second, it derives new facts from known facts in the database by applying *rules*. Each rule specifies a set of base facts and a derived fact that will be added to the database if satisfying all the base facts. Finally, the analysis applies *queries* for specific properties via a query system and produces the results of the queries as its analysis results.

Thanks to its declarative characteristic, declarative static analysis has become one of the widely-used techniques. Datalog is a well-known domain-specific language for static analysis,[3-13] which can alleviate challenges and burdens in crafting a static program analyzer. For example, DOOP,[3] a static program analyzer using Datalog, shows remarkable scalability for the Java points-to analysis, and the scalability comes from the modularity and declarativeness of Datalog.[14] As a result of successful declarative static analysis using Datalog, various declarative static analyzers that are inspired by or based on Datalog have emerged. A declarative semantic code analysis engine maintained by GitHub, called CodeQL,[15] is scalable enough to detect security vulnerabilities in Java programs of over millions of lines of code. Recently, Meta developed Glean,[16] an experimental declarative query system on which users can query information about code structures on a large-scale codebase for JavaScript and Hack programs.

Declarative static analyzers have broadened their analysis targets from a single programming language to diverse programming languages. To support a new language, one should consider the three steps of declarative static analysis. Because the query system is independent of analysis target languages, one can reuse the existing query system for the new language. Therefore, the first step is to define a new schema of the database containing facts for the new language and implement a front-end component that transforms programs written in the new language into facts. Then the next step is to define new language-specific rules for the new language. With these modifications, declarative analyzers can analyze programs written in the new language, using the existing queries and the query system. DOOP currently supports Python program analysis to detect tensor shape mismatching bugs in TensorFlow-based Python ML models.[17] CodeQL now can track dataflows not only in Java but also in C++, C#, JavaScript, Ruby, and Python programs. Furthermore, Glean plans to support diverse programming languages, including Python, Java, C++, Rust, and Haskell.

While the analyzers support multiple programming languages, they do not directly support the analysis of *multilingual programs* written in two or more programming languages. Multilingual programs are now widely developed in various application domains.[18,19] However, multilingual programs are often vulnerable to bugs or security issues more than monolingual programs. A large-scale study on the code quality of multilingual programs[18] reported that using multiple languages together correlates with higher error-proneness. Moreover, Grichi et al.[20] showed that two or three times more bugs and security issues had been reported in the language interoperation than in the intraoperation in widely-used open-source JNI programs such as OpenJ9 and VLC.

In this paper, we propose a systematic methodology that extends a declarative static analyzer supporting multiple languages to support multilingual program analysis as well. Our goal is to maximize the reuse of already existing components. First, it generates a merged database of facts that can be separated into multiple logical language spaces. Each language space consists of original facts from its corresponding language database, and existing language-specific rules derive new facts from the facts in the corresponding language space. Then, to handle the language interoperation semantics in multilingual programs, we define language-interoperation rules referring to the language interoperation semantics. The language-interoperation rules derive new facts from the facts across language spaces. The extensions enable the query system to extract facts in multilingual programs, taking the language interoperation semantics into account. Finally, the same query is evaluated under the same query system to get analysis results.

To evaluate the practicality of our approach, we develop a proof-of-concept declarative static analyzer, called MultiQL, for multilingual programs by extending CodeQL. MultiQL tracks dataflows across language boundaries for two types of multilingual programs: Java-C programs written in Java and C and Python-C programs written in Python and C. The extension is simple enough in that it requires only a few lines of automated modifications of CodeQL and additional language-interoperation rules. We also implement a bug checker on top of MultiQL to detect dataflow-related interoperation bugs in real-world Java-C programs. The evaluation shows that our tool successfully tracks dataflows across Java-C and Python-C language boundaries. The evaluation shows that MultiQL is scalable; it takes only about 12 min to analyze multilingual programs of three million lines of code, which the state-of-the-art analyzer fails to analyze in 8 h. Also, it detects 33 genuine interoperation bugs in real-world Java-C programs, including 12 new bugs that the existing analyzer could not detect due to the lack of scalability.

The contributions of this paper are as follows:

1. We propose a systematic methodology that extends a declarative static analyzer supporting multiple languages to support multilingual program analysis.
2. We implement a proof-of-concept declarative static dataflow analyzer, called MultiQL, for two types of multilingual programs, Java-C and Python-C, by extending CodeQL.
3. We show that MultiQL can successfully detect dataflow-related bugs at language boundaries of real-world multilingual programs, including new bugs that the state-of-the-art analyzers could not detect due to the lack of scalability.

2 | BACKGROUND

Figure 1 presents an overview of how a declarative static analysis works. The analysis consists of three steps. First, a given program gets converted into syntactic facts. Second, new facts are generated by iteratively applying rules to the set of known facts until no new facts are derived. Finally, the query system takes a query and evaluates the rules with the given facts, producing an analysis result for the query. The following paragraphs explain each step in detail, with examples written in the Datalog-like syntax.

Step 1: Extracting syntactic facts. The first step is to extract syntactic facts from a given program source. Syntactic facts are the facts that can be statically determined from the syntax of the given program. For example, consider the following code:

```
1 int f() {
2   return 42;
3 }
4
5 int val = f();
```

For function definitions, the Syntactic Fact Extractor can extract syntactic facts of a form `Return(functionName, retExpr)`, where Return denotes the relation between a string and an expression, `functionName` denotes the name of a function, and `retExpr` denotes the return expression of the function. Therefore, the extractor can extract `Return("f", 42)` from the code. Another example of syntactic fact is `Call(callExpr, functionName)`, which denotes that `callExpr` is a call expression to a function named `functionName`. For instance, the extractor can extract the following syntactic fact from the code on line 5: `Call(f(), "f")`.

Step 2: Deriving new facts by applying rules. The next step is to derive new facts from known facts by applying rules. A rule defines a relation that a fact, called a *derivable fact*, can be derived from a set of facts, called *base facts*. If all the base facts belong to known facts, the derivable fact is derived and added to the known facts by the rule. This process of deriving new facts is repeated until no more new facts are found.

For example, consider a fact `Step(x, y)` that denotes a direct dataflow from node `x` to node `y`. Here, nodes represent program entities that can hold runtime values, such as variables, literals, expressions, and function parameters. For instance, the literal `42` on line 2, the function call `f()` on line 5, and the variable `val` on line 5 are all examples of nodes. A direct dataflow can be established via the relation between a function call expression and its callee function's return expression because a function call expression evaluates to its callee function's return value on runtime. We can represent such dataflows as the following rule:

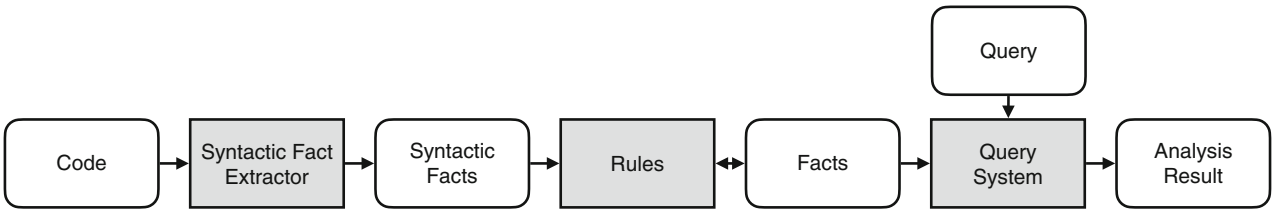

FIGURE 1 Overview of a declarative static analysis

```
Step(retExpr, callExpr) :- Call(callExpr, functionName),Return(functionName,retExpr)
```

`Step(retExpr, callExpr)` on the left side of the rule is a derivable fact, and `Call(callExpr, functionName)` and `Return(functionName, retExpr)` on the right side of the rule are base facts. This rule indicates that the new fact `Step(retExpr, callExpr)` is derivable if the two base facts are known. Since we already have `Calls(f(), "f")` and `Return("f", 42)` as known facts, the rule derives a new `fact Step(42, f())`. Similarly, the assignment expression on line 5 makes the value of the right side flow into the variable on the left side. We can also define a rule for assignment expressions to derive the fact `Step(f(), val)`, which denotes a direct flow from the result of `f()` to val.

Derived facts by rules are also known facts used to derive new facts. For example, we can think of another, `Flow(x, z)`, that denotes a dataflow from node `x` to node `z`. We can define rules for the dataflow fact as the transitive closure of `Step`:

```
Flow(x, z) :- Step(x, z)
Flow(x, z) :- Step(x, y), Flow(y, z)
```

The first rule indicates that the fact `Flow(x, z)` is derivable if the fact `Step(x, z)` is known, and the second rule indicates that the fact `Flow(x, z)` is also derivable if the two facts `Step(x, y)` and `Flow(y, z)` are known. Thus, the rules derive `Flow(42, f())`, `Flow(f(), val)`, and `Flow(42, val)` from known facts, because `Step(42, f())` and `Step(f(), val)` are known.

Step 3: Performing queries. The final step is to perform the query via the query system. A query is a set of facts containing variables, and the query system finds every variable assignment that makes each fact in the query belong to known facts after substituting all free variables in the query with the assigned constants. This step corresponds to actually obtaining the final result of a static analysis in a declarative style. For example, one can make a specific query:

```
?- Flow(42, X)
```

that queries all nodes into which the integer literal `42` flows. When accepting the query as an input, the query system finds every node `n` such that `Flow(42, n)` belongs to known facts generated in the previous step. In this example, the query would give the query result `X ∈ {f(), val}`.

3 | DECLARATIVE STATIC ANALYSIS FOR MULTILINGUAL PROGRAMS

This section outlines our approach to extending an existing declarative static analysis for multilingual program analysis. We show how an example declarative static analysis works for call graph construction in two different monolingual programs, and how we extend the analysis to construct a unified call graph encompassing language boundaries.

3.1 | Overview

Figure 2 illustrates how we support multilingual analysis in a declarative style in the case of two languages as an example. The declarative analyzer now gets two sets of syntactic facts extracted from two different languages. In addition, new language-interoperation rules are defined on top of the original language-specific rules from two languages to take the interoperation semantics into account. Then, the same query is performed to get analysis results.

3.2 | Example declarative static analysis

An example declarative static analysis consists of the following Datalog-like facts, rules, and queries:

$$
\begin{aligned}
f &::= p(\overline{k \mid x}) && \text{FACT}\\
r &::= f' \ :- \overline{f \mid \neg f} && \text{RULE}\\
q &::= ?- \ \overline{f} && \text{QUERY}
\end{aligned}
$$

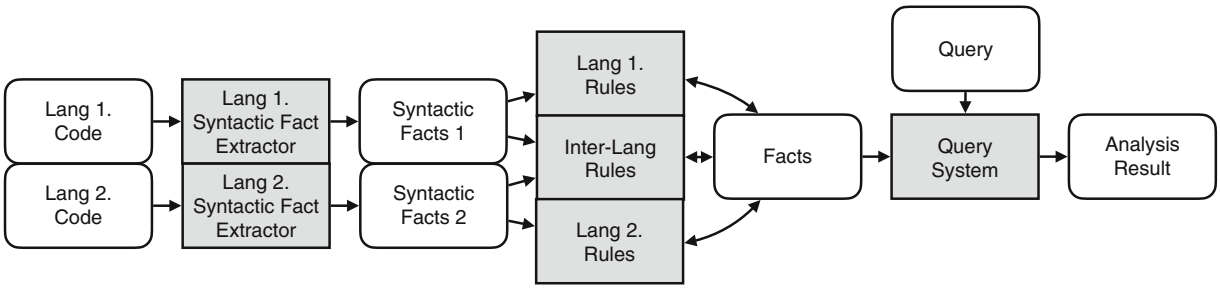

FIGURE 2 Declarative analysis for multilingual programs

```
1 import f
2 def m1():
3     f()
4 def m2():
5     return None
6 m1()
```

(A) Code in Python-like language A

```
7 void f() {
8     CallMethod("m2");
9 }
```

(B) Code in C-like language B

FIGURE 3 Multilingual code example written in both A and B

We use the overline notation to indicate 0 or more repetitions. For example, $\overline{x}$ is a shorthand for $x_1, x_2, \ldots, x_n$ for some $n \geq 0$. The fact $p(\overline{k} \mid \overline{x})$ denotes a relation p between constants $\overline{k}$, such as string and integer literals, or variables $\overline{x}$. A rule $f' :- \overline{f} \mid \overline{\neg f}$ consists of two parts: the head of the rule f' and the body of the rule $\overline{f} \mid \overline{\neg f}$, where the optional prefix $\neg$ denotes the negation.* It denotes that the head is derivable from the body: f' is derivable if all the facts $f'' \in \{\overline{f}\}$ belong to known facts, and all the facts $f'' \in \{\overline{\neg f}\}$ do not belong to known facts. Note that a derived fact should not contain any free variables **in** its argument. The body of the rule can be empty; in that case, the head of the rule is vacuously added to known facts. Such bodiless rules are the initial input facts for the analysis. The query $? - \overline{f}$ finds all possible variable assignments that make all facts in $\overline{f}$ belong to known facts if replacing variables in $\overline{f}$ with their corresponding constants.

3.3 | Declarative call graph construction for monolingual programs

Figure 3 shows a multilingual code example written in (A) Python-like language A and (B) C-like language B. In A code, there are two functions, `m1` on line 2 and `m2` on line 4. The function `m1` calls the function `f` in B code via language interoperation between A and B, and the function `m2` does nothing. On line 6 in A, the expression `m1()` calls the function `m1` defined on line 2 to invoke the function `f` in B code. The function `f` in B code calls the CallMethod function with a string literal `"m2"` to invoke the function `m2` in A code, using the string literal argument as the target function name.

A declarative static analysis first creates a database of facts for the code. Initial facts are just syntactic information extracted from the code. For example, there are two kinds of facts required for call graph construction: `FunctionAt(lineNum, name)` denotes function definition information containing its line number `lineNum` and the function name `name`, and `CallAt(lineNum, name)` denotes callsite information containing the callsite line number `lineNum` and its target function name `name`. The initial facts for the code in Figure 3A are as follows:

```
FunctionAt(2, "m1")
CallAt(3, "f")
FunctionAt(4, "m2")
CallAt(6, "m1")
```

*A syntactic restriction exists for negation and recursive rules: a fact should not be derived from its negation. For example, the rule $R(x, y) :- \neg R(x, y)$ is not a syntactically valid rule since $R(x, y)$ cannot be derived from $\neg R(x, y)$.

Then, the analysis requires rules for the initial facts to derive new facts about function call relations. In this example, one kind of facts, `CallEdge(lineNumCall, lineNumFunc)`, denotes a call relation from a callsite on line `lineNumCall` to a function definition on line `lineNumFunc`. To derive function call relation facts, we define a rule as follows:

```
CallEdge(lineNumCall, lineNumFunc) :-
  CallAt(lineNumCall, name),
  FunctionAt(lineNumFunc, name)
```

The rule derives a new fact `CallEdge(lineNumCall, lineNumFunc)` when a callsite on line `lineNumCall` has the same target function name as the name of a function defined on line `lineNumFunc`. Therefore, a new fact can be derived from the initial facts as follows:

```
CallEdge(6, 2)
```

Finally, a query system extracts a set of function call relation facts when we make a query. For example, the following query finds all possible literals for the variables `X` and `Y`:

```
?- CallEdge(X, Y).
```

Since only one `CallEdge` fact, `CallEdge(6,2)`, belongs to known facts, the query system produces `(X, Y)` $\in$ { `(6, 2)` } as a result of the query.

Let us consider the language *B* case. The initial facts for the code in Figure 3B are as follows:

```
ProcedureAt(7, "f")
CallAt(8, "CallMethod")
Argument(8, 0, "m2")
```

The fact `ProcedureAt(lineNum, name)` denotes that a procedure of name `name` is defined on line `lineNum` similarly to `FunctionAt(lineNum, name)` for the language *A*. The fact `Argument(lineNum, i, arg)` denotes that the value of the `i`-th argument of a function call on line `lineNum` is `arg`.

Then, we define a slightly different rule to derive a function call relation from the initial facts as follows:

```
CallEdge(lineNumCall, lineNumFunc) :-
  CallAt(lineNumCall, name),
  ProcedureAt(lineNumFunc, name)
```

This example shows a language-specific rule: the same fact `CallEdge` can be derived from different facts for different languages. When we make the query `?- CallEdge(X, Y)`, the query system produces `(X, Y)` $\in$ {} as a result since it has no facts to satisfy the rule.

3.4 | Extension of declarative call graph construction for multilingual programs

The first step of our extension for multilingual program analysis is to create a merged database that consists of two logical language spaces, *A* and *B*. Then, we store the initial facts of the code in Figure 3A,B to their corresponding language spaces, respectively. The following shows the merged database containing the initial facts, where the subscripts of the facts denote the logical language spaces in which the facts reside:

```
FunctionAt_A(2, "m1")
CallAt_A(3, "f")
FunctionAt_A(4, "m2")
CallAt_A(6, "m1")
ProcedureAt_B(7, "f")
CallAt_B(8, "CallMethod")
Argument_B(8, 0, "m2")
```

The next step defines generalized rules to derive function call relation facts. We define them by composing language-specific rules for A, language-specific rules for B, and language-interoperation rules as follows:

```
CallEdge(lineNumCall, lineNumFunc) :- CallEdge_A(lineNumCall, lineNumFunc)
CallEdge(lineNumCall, lineNumFunc) :- CallEdge_B(lineNumCall, lineNumFunc)
```

```
CallEdge(lineNumCall, lineNumFunc) :- CallEdge_AB(lineNumCall, lineNumFunc)
CallEdge(lineNumCall, lineNumFunc) :- CallEdge_BA(lineNumCall, lineNumFunc)
```

We slightly modify the language-specific rules for A and B to derive the $\texttt{CallEdge}_A$ and $\texttt{CallEdge}_B$:

```
CallEdge_A(lineNumCall, lineNumFunc) :-
  CallAt_A(lineNumCall, name),
  FunctionAt_A(lineNumFunc, name)

CallEdge_B(lineNumCall, lineNumFunc) :-
  CallAt_B(lineNumCall, name),
  ProcedureAt_B(lineNumFunc, name)
```

In addition, we define additional language-interoperation rules to derive $\texttt{CallEdge}_{AB}$ and $\texttt{CallEdge}_{BA}$. The fact $\texttt{CallEdge}_{AB}$ denotes a function call relation from a function written in the language A to a function written in the language B, and $\texttt{CallEdge}_{BA}$ denotes a function call relation in the opposite direction. Since the function call semantics from A to B is the same as the normal function call semantics in A, we define the interoperation rule from A to B as follows:

```
CallEdge_AB(lineNumCall, lineNumFunc) :-
  CallAt_A(lineNumCall, name),
  ProcedureAt_B(lineNumFunc, name)
```

On the other hand, the function call semantics from B to A is different from the normal function call semantics in B. The language B calls a function written in A by calling an interoperation API function `CallMethod` with a target function name as the first argument. Thus, we define the interoperation rule from B to A as follows:

```
CallEdge_BA(lineNumCall, lineNumFunc) :-
  CallAt_B(lineNumCall, "CallMethod"),
  Argument_B(lineNumCall, 0, name),
  FunctionAt_A(lineNumFunc, name)
```

Finally, we can make the same query `?- CallEdge(X, Y)` as we did for the monolingual programs in Section 3.3. Note that this query is also valid for the multilingual program since we specifically defined the new rule to derive the fact having the same name, `CallEdge`. The query system now produces `(X, Y)` $\in$ {`(6, 2), (3, 7), (8, 4)`} as the function call relation results, which include not only intra-language call relations but also inter-language call relations.

This simple example shows what our extension for multilingual program analysis requires. Our extension reuses most components in an existing declarative static analysis with slight modifications. Because the modifications of a database and language-specific rules are trivial, we can fully automate the modification process. The only manual part in our extension is to define additional language-interoperation rules. We explain how we extend CodeQL to support multilingual program analysis for Java-C and Python-C programs in the next section.

4 | MULTIQL: EXTENSION OF CODEQL FOR MULTILINGUAL PROGRAM ANALYSIS

In this section, we present MultiQL, a prototype extension of CodeQL[15] for multilingual program analysis. As a prototype implementation, MultiQL does not aim for a sound or complete analyzer, yet it can effectively track dataflows in practical multilingual programs using the simple approach explained in the previous section. MultiQL tracks dataflows across language boundaries in two types of multilingual programs: (1) Java-C programs interoperating via Java Native Interface (JNI)[21] and (2) Python-C programs interoperating via Python Extension Module.[22] For simplicity, we use the Java-C program analysis to explain the common parts and emphasize the Python-C program analysis in Section 4.5.

4.1 | CodeQL

CodeQL is a declarative static analysis engine that transforms source code into a database of facts and performs analyses by evaluating queries written in the declarative and object-oriented language called QL (Query Language). Using QL,

one can depict rules by defining *predicates* and *classes*. The following QL code defines the classical Datalog-style unary predicate `isOneOrTwo`:

```
1 predicate isOneOrTwo(int n) {
2   n = 1 or n = 2
3 }
```

which states that the fact `isOneOrTwo(n)` is derivable from either of the two facts, `n = 1` and `n = 2`.

QL allows syntactic definitions of predicates which, like C functions, can have return types, which are desugared into classical predicates by adding an additional argument named `result`. For example, consider the following predicate `addOne`:

```
1 int addOne(int n) {
2   isOneOrTwo(n) and result = n + 1
3 }
```

The predicate is a syntactic sugar equivalent to the following classical predicate:

```
1 predicate addOnePred(int n, int result) {
2   isOneOrTwo(n) and result = n + 1
3 }
```

Then, every use site of the original predicate is rewritten to use the desugared predicate. For example, an equality formula `m = addOne(n)` is desugared to `addOnePred(n, m)`.

One characteristic feature of QL is class definitions similar to the ones in object-oriented languages like Java. A QL class should extend another type and define a special predicate called a *characteristic predicate*. It is similar to a constructor in object-oriented languages; it has the same name as its enclosing class and does not have any return type. A QL class can define member predicates like methods in Java. The following QL code defines a class named `OneOrTwo` that extends `int` with the characteristic predicate and one member predicate:

```
1 class OneOrTwo extends int {
2   // characteristic predicate
3   OneOrTwo() { this = 1 or this = 2 }
4   // member predicate
5   int add(OneOrTwo that) { result = this + that }
6 }
```

Despite its syntax, a QL class is also a syntactic sugar of classical predicates. For example, the characteristic predicate `OneOrTwo` is desugared into the following unary predicate, `isOneOrTwo`:

```
1 predicate isOneOrTwo(int this) {
2   this = 1 or this = 2
3 }
```

Note that the type of the argument `this` is `int`, which the class `OneOrTwo` extends. Similarly, The member predicate `add` is desugared into a ternary predicate as follows by introducing two more arguments, `this` and `result`:

```
1 predicate addPred(OneOrTwo this, OneOrTwo that, int result) {
2   result = this + that
3 }
```

Then, every use site of the class is rewritten to use the desugared predicates. For example, `z = x.add(y)` where x has type `OneOrTwo` can be desugared into addPred(x, y, z). For more detailed information about CodeQL and its language QL, refer to the paper of Avgustinov et al.[2] or the official document.[15]

Figure 4 presents the overall structure of MultiQL for JNI programs. First, it generates databases for both languages, C[†] and Java, and merges them into one database. This corresponds to the step of extracting the initial syntactic facts from the source code. Then, MultiQL merges the common rules in both languages, which are parts of their libraries that CodeQL provides, into one merged library. Finally, using the merged database and merged library, a user can write a query to perform a client-analysis and evaluate it to produce its analysis result.

[†]Even though MultiQL analyzes JNI programs written in Java and both C and C++, this paper refers to C only for presentation brevity.

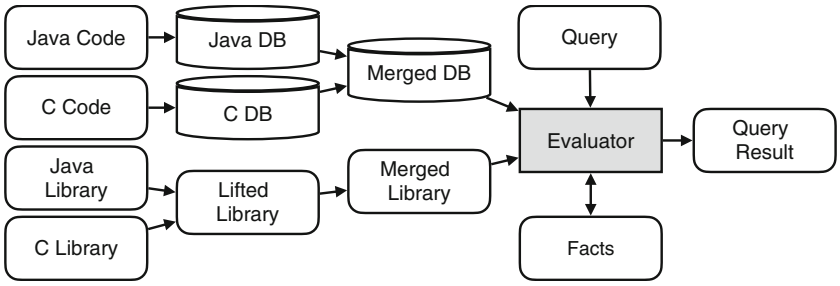

FIGURE 4 Overall structure of MultiQL for JNI programs

4.2 | Creating databases

For compiled languages such as C and Java, CodeQL generates their databases by compiling source programs. When a compiler compiles a program, CodeQL monitors the compiler to extract necessary information and creates a database with the extracted information. For scripting languages like Python, CodeQL uses its own extractor to directly extract necessary information from source code. CodeQL creates a database for a single language in two steps: (1) it stores the extracted information in a single *trap* file, a human-readable format file for the CodeQL database, and (2) it transforms the trap file into a database in binary format. For example, the following demonstrates a sample trap file:

```
#10001=@"class;myClass.MyClass"
#10002=@"type;int"
primitives(#10002,"int")
#10003=@"callable;{#10001}.myMethod({#10002}){#10002}"
#10004=@"params;{#10003};0"
params(#10004,#10002,0,#10003,#10004)
paramName(#10004,"myParam")
...
```

which describes the parameter information of the method `myMethod`. It also shows facts that will eventually be stored in tables: `primitives(...)`, `params(...)`, and `paramName(...)`. The database manages tables based on fact types. For example, it stores the fact `primitives(...)` in a table named `@primitives`, and `params(...)` in a different table named `@params`.

To create a single merged database for both C and Java, MultiQL maintains a separate trap file for each language and then merges them into a trap file. One problem is that if each trap file has a table with the same name, the merge fails due to the name collision. To avoid the problem, MultiQL renames each table to a globally unique name by appending a language-specific prefix to the table name before the merge. For example, if a table named `@params` exists in both C and Java trap files, MultiQL renames the table in C to `@c_params` and the table in Java to `@java_params`. After renaming such tables, MultiQL merges the trap files into a database on which predicates and queries are evaluated.

4.3 | Lifting libraries

CodeQL provides various libraries for C and Java, including pre-defined predicates and classes for users to implement their own analyses. A dataflow analysis library is such a library, which supports both C and Java. For example, Figure 5A,B show sample QL libraries for C and Java, respectively, which use the same class name `Node` and the same predicate name `localFlowStep`. However, even with the same name, the class `Node` in C and the class `Node` in Java are different classes, which are incompatible. The same applies to the predicate `localFlowStep`. In other words, we can not use the class `Node` in C as an argument of the predicate `localFlowStep` in Java or vice versa.

To make classes and predicates in C and Java compatible, MultiQL lifts each library to the common level. First, as on lines 1 and 2 of Figure 5C, MultiQL encapsulates the original dataflow libraries for C and Java in separate modules

```
class Node { ... }

predicate localFlowStep(Node from, Node to) {
  // Expr -> Expr
  exprToExprStep_nocfg(from.asExpr(), to.asExpr())
  or
  // Assignment -> LValue post-update node
  ...
}
```

(A) c/dataflow/internal/DataFlowUtil.qll

```
class Node { ... }

predicate localFlowStep(Node node1, Node node2) {
  // Variable flow steps through assignment expression
  node2.asExpr().(AssignExpr).getSource() = node1.asExpr()
  or
  // Variable flow steps through adjacent def-use and use-use pairs.
  ...
}
```

(B) java/dataflow/internal/DataFlowUtil.qll

```
module C { /* original classes and predicates from C lib */ }
module JAVA { /* original classes and predicates from Java lib */ }

private newtype TNode =
  TJavaNode(JAVA::Node n)
  or
  TCNode(C::Node n)
class Node extends TNode {
  JAVA::Node asJavaNode() { this = TJavaNode(result)}
  C::Node asCNode() { this = TCNode(result) } ...
}

predicate localFlowStep(Node node1, Node node2) {
  JAVA::localFlowStep(node1.asJavaNode(), node2.asJavaNode())
  or
  C::localFlowStep(node1.asCNode(), node2.asCNode())
}
```

(C) jni/dataflow/internal/DataFlowUtil.qll

FIGURE 5 QL libraries for C and Java

named `C` and `Java`, respectively[‡]. After the encapsulation, the original classes and predicates can be referenced via the module containing them. For example, `Java::Node` refers to the original class `Node` in the module `Java`. Lines 4 to 11 of Figure 5C demonstrate the lifted class of the original class Node. MultiQL lifts a class by first defining a sum type[§], denoting that the lifted class comes either from C or Java, and then making the lifted class of that type. The lifted class

[‡]https://codeql.github.com/docs/ql-language-reference/modules/
[§]https://codeql.github.com/docs/ql-language-reference/types/#algebraic-datatypes

defines two member predicates, `asCNode` and `asJavaNode`, that downcast the elements of the lifted class into the elements of the original C or Java class. Lines 13 to 17 of Figure 5C demonstrate the lifted predicate of the original predicate `localFlowStep`. Similarly, MultiQL lifts a predicate by combining two original predicates with the `or` connective. For each predicate, arguments and return values are downcasted to the elements of the class of their corresponding language. After lifting, lifted predicates show the equivalent behaviors as the original ones if all the arguments are from the same language.

MultiQL can fully automate the lifting process using QL compiler error messages. When compiling a query without importing the library, the QL compiler reports error messages containing required classes and predicates, and signatures of the predicates. Using the information, MultiQL automatically synthesizes lifted classes and predicates.

4.4 | Merging libraries: Java-C

After lifting libraries for different languages, we manually extend predicates to reflect the interoperation semantics between multiple languages. For the Java-C program analysis, we identified various interactions from Java to C and vice versa via JNI, and extended predicates to model their behaviors. For example, the following shows how we extend the CodeQL predicate named `viableCallable`:

```
1 DataFlowCallable viableCallable(DataFlowCall c) {
2   result.asJavaNode()     = JAVA::viableCallable(c.asJavaNode())
3   or result.asCNode()     = C::viableCallable(c.asCNode())
4   or result.asCNode()     = viableCallableJ2C(c.asJavaNode())
5   or result.asJavaNode() = viableCallableC2J(c.asCNode())
6 }
```

It finds call edges from call expressions to their targets. Lines 2 and 3 show the results of lifting using original predicates from the dataflow libraries. They handle intra-language call edges from Java to Java and from C to C, respectively. Lines 4 and 5 show the results of merging libraries, representing inter-language call edges. The predicate `viableCallableJ2C` finds call edges from Java to C, and `viableCallableC2J` finds call edges from C to Java.

Java to C. In Java-C programs, one can make interactions from Java to C by calling native functions in C from Java code. The target of such a function call is determined in a static manner. The target function should follow the JNI naming convention, which is adding `Java_` as prefix, followed by a fully qualified name of its class and the additional `_`, to the method name. For example, the target function name for a function call of `cfunction` would be `Java_fully_qualified_class_name_cfunction`. With this convention, we can define `viableCallableJ2C` so that `f = viableCallableJ2C(call)` holds when `f.toString() = "Java_" + call.getTarget().className() + "_" + call.getTarget().getName()` holds.

C to Java. The interaction from C to Java is more complex and requires more careful implementation than the interaction from Java to C. The primary difference is that a method call from C to Java requires the runtime values of variables, which may not be always possible. First, C code calls the interface function `GetMethodID(name, sig)` to get the "method ID" of the Java method whose name matches the first argument and the type signature matches the second argument passed to this function. This method ID is stored at a variable, say `mid`, and an actual method call is invoked by another interface function, `Call<type>Method(obj, mid, args…)`. Calling this interface function corresponds to calling the method that `mid` indicates with `obj` as "this object" and `args` as the arguments.

To correctly handle this method call, we should be able to answer these questions: "When we call `GetMethodID`, what are the string values of `name` and `sig`?" and "When we call `Call<type>Method`, what is the method ID value of `mid`?" Soundly answering these questions requires inter-language dataflow analysis because the string or method ID values may be passed across language boundaries. However, we observed that such a pattern rarely happens in practice, and using only intra-language dataflow analysis within C code is enough for most cases. Therefore, we decided to sacrifice soundness by using intra-language analysis instead of inter-language analysis. In rare cases, this may result in missing some call edges as a trade-off for a more lightweight and simpler implementation. We implemented two intra-language flow analysis modules for C, which find (1) dataflows from string literals to the arguments of interface functions and (2) dataflows from interface function call results to the arguments of interface functions. Using these modules, we can

implement the predicate `viableCallableC2J` by adding a call edge from a `Call<type>Method` call to the method `m`, if there is a flow to `mid` from a call to `GetMethodID`, and string values that flow into the arguments `name` and `sig` of `GetMethodID` that correspond to the name and the type signature of the method `m`.

In addition to the predicate `viableCallable`, we extended more predicates to consider other JNI interface functions such as `findClass` and `GetFieldID`. Most of such extended predicates are specialized `step` predicates. We extended them in a similar way to the calls from C to Java described above.

4.5 | Merging libraries: Python-C

To analyze Python-C programs, we extended predicates to model the interoperation semantics of Python Extension Module. The following shows `viableCallable` predicate we extended for the Python-C program analysis:

```
DataFlowCallable viableCallable(DataFlowCall c) {
  result.asPythonNode()     = PYTHON::viableCallable(c.asPythonNode())
  or result.asCNode()       = C::viableCallable(c.asCNode())
  or result.asCNode()       = viableCallableP2C(c.asPythonNode())
  or result.asPythonNode() = viableCallableC2P(c.asCNode())
}
```

The only different parts from the Java-C program analysis are the predicates `viableCallableP2C` and `viableCallableC2P`.

Python to C. Similar to the interactions from Java to C, one can make interactions from Python to C by importing and calling functions from C. Python Extension Module provides a pre-defined C struct `PyMethodDef` to export a C function to Python:

```
1 struct PyMethodDef methods[] = {
2   {
3     .ml_name = "cfunction",
4     .ml_meth = cfunction_impl, ...
5   }, ...
6 }
```

The member field `ml_name` has a visible name of a C function to Python, and the member field `ml_meth` has the pointer to the actual C function. Thus, the function `cfunction_impl` defined in C code is invoked when importing and calling `cfunction` in Python code. To model the behavior, we define the CodeQL class `PyMethodDef` and the predicate `viableCallableP2C`. The class `PyMethodDef` corresponds to the C structure `PyMethodDef`. It has two member predicates `getName()` and `getFunc()` whose results are the values of the fields `ml_name` and `ml_meth`, respectively. Then, we make a rule for `viableCallableP2C` such that for some `def` of type `PyMethodDef`, if `def.getName() = call.getTarget().toString()` for some `call`, then `def.getFunc() = viableCallableP2C(call)` holds.

In addition, we define rules that connect dataflows from arguments of a Python-to-C function call to parameters of its target C function. Unlike the Java-to-C function call semantics, Python Extension Module packs arguments in a Python tuple object and propagates the object to a single parameter of the target C function. Then, the target C function unpacks the tuple object into individual Python objects by calling the `PyArg_ParseTuple` API. This means that the usual way of connecting arguments and parameters via their positions does not work, and we need another way. We modelled this behavior by defining `VirtualArgNode`, a subclass of `Node`, that corresponds to the Python tuple object. Then, we define some predicates for representing flows in and out of this node. First, we define a step that represents the flow from values of the original argument nodes to virtual argument nodes. We then define a step that represents the flow from the values of virtual argument nodes to a single parameter node in a C function. By doing so, the flows from original argument nodes to use sites will be found in three steps: (1) from an argument node to a virtual argument node, (2) from the virtual argument node to the parameter node of a C function, and (3) from the parameter node to the out node of `PyArg_ParseTuple` API function call.

C to Python. The interaction from C to Python requires the runtime values of variables, similar to the interaction from C to Java. To invoke Python functions in C, C code calls the interface function `PyObject_CallObject(func,`

`args)`, where `func` is a Python function object, and `args` is a Python tuple object that contains all arguments. Because C code can get Python objects as function arguments of the Python-to-C function calls, we define two "intra-language flows" analyses to identify Python objects assigned to `func`: (1) dataflows from a Python function object to the argument of a C function call, and (2) dataflows from the parameters of a C function to the first argument of `PyObject_CallObject`. Using these intra-language flow analyses, we can find actual targets that should be invoked for `PyObject_CallObject` API function calls.

4.6 | Discussion

We discuss possible future research directions to enhance MultiQL.

Soundness We manually implemented the dataflow steps that model the behaviors of the inter-language API functions by referring to the JNI specification[21] and Python's extension module specification.[23] We implemented a few selected API functions that are enough to practically analyze all of our benchmark suites and did not fully cover all the API functions. This may lead to the unsoundness of MultiQL. One breakthrough may use the mechanized specification approach[24-28] and automatically generate the modeling of such API functions from mechanized specifications.

Another source of unsoundness may come from dynamically generated Python code. Because MultiQL is a purely static analyzer, it cannot analyze statically invisible code. A possible direction may take a hybrid approach leveraging dynamic analysis.[29]

Extension MultiQL can analyze interoperations between multiple languages via explicit Foreign Function Interfaces (FFIs). Therefore, as long as analysis target languages use FFIs, our approach can be easily extended to analyze interactions between languages other than Java-C or Python-C and even interactions between three or more programming languages since providing inter-language FFI call relations enables CodeQL dataflow analysis. However, as Li et al.[30] explain, various interoperation mechanisms beyond explicit FFIs, such as implicit interactions between languages using interprocess communications (e.g., socket-based message passing) or languages embodying other languages (e.g., web browser engines), exist in real-world multi-language software systems. Supporting these kinds of interoperation would be a promising future direction. One possible approach could be dynamically collecting various kinds of language interactions, simulating the interactions as explicit FFIs, and analyzing them using MultiQL. Even though dynamically collecting behaviors is unsound and costly, such a hybrid analysis may be able to analyze more language interactions than MultiQL.

5 | EVALUATION

To show the effectiveness of our approach, we evaluate MultiQL with the following two research questions:

1. **RQ1: Feasibility.** Does MultiQL correctly analyze multilingual programs that use various interoperations?
2. **RQ2: Usefulness.** Does MultiQL correctly analyze inter-language flows to detect bugs in real-world multilingual programs?

5.1 | RQ1: Feasibility

We evaluate the feasibility of MultiQL by dataflow analysis on two benchmark suites for each of Java-C and Python-C analyses. For the Java-C analysis, we use NativeFlowBench[31,32] and real-world JNI Android apps downloaded from F-Droid[33] as the benchmark suites. For the Python-C analysis, we use ExtModuleFlowBench, which we developed, and real-world C-Python programs downloaded from GitHub.

5.1.1 | Feasibility: Java-C

The first benchmark suite for Java-C program analysis is NativeFlowBench,[31,32] consisting of 23 JNI Android applications (apps) that use various JNI interoperations. They contain sensitive data leakage from *sources* to *sinks* across

TABLE 1 Analysis results of NativeFlowBench

Benchmark	JN-SAF	MultiQL	Benchmark	JN-SAF	MultiQL
icc_javatonative	✓	✗(FN)	native_noleak	✓	✓
icc_nativetojava	✓	✗(FN)	native_noleak_array	✗(FP)	✓
native_complexdata	✓	✓	native_nosource	✓	✓
native_complexdata_stringop	✗(FN)	✗(FN)	native_pure	✓	✓
native_dynamic_register_multiple	✓	✓	native_pure_direct	✓	✓
native_heap_modify	✓	✓	native_pure_direct_customized	✓	✓
native_leak	✓	✓	native_set_field_from_arg	✓	✓
native_leak_array	✓	✗(FN)	native_set_field_from_arg_field	✓	✓
native_leak_dynamic_register	✓	✓	native_set_field_from_native	✓	✓
native_method_overloading	✓	✓	native_source	✓	✓
native_multiple_interactions	✓	✓	native_source_clean	✓	✓
native_multiple_libraries	✓	✓			

language boundaries via JNI interoperations. We compare the analysis results of MultiQL to those of the state-of-the-art Java-C program dataflow analyzer, JN-SAF.[32] We use compiled versions for JN-SAF because it targets compiled JNI programs.

Table 1 summarizes the analysis results of JN-SAF and MultiQL on NativeFlowBench. **Benchmark** columns show the benchmark names and JN-SAF and **MultiQL** columns show the analysis results of JN-SAF and MultiQL, respectively. If an analysis reports all data leakages correctly without any false positives (FP) or false negatives (FN), it is a success (✓). Otherwise, it is a failure (✗). MultiQL finds data leakages correctly for 19 benchmarks but reports false negatives for four benchmarks, while JN-SAF analyzes 21 benchmarks correctly. One common failure comes from string concatenation: *native_complexdata_stringop* generates a Java field name by concatenating two string values via a built-in function. Because MultiQL does not handle built-in functions, it fails to analyze the benchmark. *icc_javatonative* and *icc_nativetojava* leak data via the Android inter-component communication, which is beyond the scope of MultiQL. The remaining different failures of MultiQL and JN-SAF come from their different array analysis policies. While both *native_leak_array* and *native_noleak_array* store sensitive data in an array, the former leaks the data, but the latter does not. Because JN-SAF over-approximates dataflows on arrays, it analyzes *native_leak_array* correctly but reports a false positive for *native_noleak_array*. By contrast, because CodeQL under-approximates dataflows on arrays, MultiQL analyzes *native_noleak_array* correctly but reports a false negative for *native_leak_array*.

The results show that MultiQL can successfully analyze JNI programs that use various JNI interoperations, and its correctness is comparable with that of JN-SAF.

The second benchmark suite consists of real-world Java-C Android apps downloaded from F-Droid, a repository of open-source Android apps.[33] We first downloaded all available apps from F-Droid, and classified downloaded apps into JNI and non-JNI apps. Then, we selected all 42 apps that can be compiled without any errors as our analysis targets. For this real-world benchmark, we compare the analysis results of MultiQL to those of JN-Sum,[34] the state-of-the-art general-purpose Java-C program analyzer. We use JN-Sum as a comparison target instead of JN-SAF, since JN-SAF is not scalable enough to analyze the real-world benchmark.

Table 2 shows the analysis results of MultiQL on 25 apps that have interoperations from C to Java. The first column shows app names with their indices, the second shows the lines of code, the third shows database creation time, the fourth shows query processing time, and the fifth shows the total analysis time. **C->Java Function Call** and **C->Java Field Access** show the numbers of C-to-Java function calls and C-to-Java field accesses, respectively. We collectively call them *JNI uses*. The sub-columns **#Precise**, **#Resolved**, and **Total** represent the numbers of precisely resolved, resolved, and the total JNI uses, respectively. We considered a resolved JNI use as precise, when MultiQL finds a single target method or a single field at the JNI use. MultiQL resolves 1076 out of 1171 (92%) JNI uses, including 347 out of 404 (86%) C-to-Java function calls and 729 out of 767 (95%) C-to-Java field accesses. In addition, 1008 (86%) resolved JNI uses are precise.

TABLE 2 Analysis results of real-world android JNI applications

		Time (s)			C->Java function call			C->Java field access		
Application	**LOC**	**DB creation**	**Query**	**Total**	**# Precise**	**# Resolved**	**Total**	**# Precise**	**# Resolved**	**Total**
1. Agram	1930	11.183	6.829	18.012	0	0	2	4	4	4
2. AndIodine	10,536	15.031	8.114	23.145	1	1	1	0	0	0
3. APV PDF Viewer	717,166	89.097	35.688	124.785	4	4	4	15	15	16
4. CommonsLab	113,087	61.464	20.554	82.018	4	5	5	0	0	0
5. CrossWords	115,256	74.693	29.106	103.799	68	68	70	9	10	14
6. Document Viewer	2,620,307	257.745	75.934	333.679	6	6	6	23	23	24
7. DroidZebra	32,766	30.732	12.608	43.340	4	5	5	0	0	0
8. FBReader	143,654	148.872	30.072	178.944	0	0	0	0	0	1
9. Fwknop2	15,332	30.717	10.446	41.163	0	0	0	0	13	13
10. Graph 89	384,541	122.544	598.645	721.189	1	1	1	0	0	0
11. Irssi ConnectBot	43,548	19.652	11.196	30.848	1	1	1	0	0	2
12. Lumicall	279,071	70.916	27.104	98.020	4	4	4	2	2	13
13. Navit	213,352	99.253	46.264	145.517	16	22	55	0	0	0
14. NetGuard	16,536	41.871	12.716	54.587	0	9	9	3	27	27
15. Overchan	66,562	32.542	15.143	47.685	1	2	4	0	0	1
16. Plumble	165,015	57.201	29.253	86.454	0	0	0	610	610	610
17. PrBoom	200,105	75.382	32.001	107.383	7	7	15	0	0	0
18. Rtl-sdr driver	27,708	45.404	11.751	57.155	2	2	2	0	0	0
19. Sipdroid	95,063	43.092	18.577	61.669	2	2	2	2	2	16
20. Son of Hunky Punk	359,041	75.795	31.102	106.897	50	50	52	10	10	10
21. Taps Of Fire	31,882	15.540	8.145	23.685	0	0	0	2	2	2
22. Tileless Map	830,915	579.649	146.806	726.455	50	58	59	3	4	5
23. Timidity AE	183,680	48.165	28.452	76.617	16	16	16	0	0	0
24. Tux Paint	2,925,311	538.863	184.591	723.454	80	83	89	4	4	6
25. VotAR	2652	10.273	6.532	16.805	1	1	2	3	3	3
					318	347	404	690	729	767

MultiQL fails to resolve 95 (8%) JNI uses because of complex language semantics such as arrays and function pointers, on which CodeQL does not track dataflows.

By contrast, JN-Sum fails to analyze one app because of an error, four apps due to the lack of memory spaces, and three apps even in 8 h. In the remaining 17 apps, JN-Sum resolves 71% JNI uses and precisely resolves 46% of JNI uses, which shows significantly more imprecise results than MultiQL.

MultiQL is scalable in that it can analyze large-scale programs. The analysis time was 161.3 s on average for each app, including 103.8 s for DB creation and 57.5 s for query processing. DB creation took more time than query processing except for *Graph 89*, and the DB creation time was almost linear to the code size. MultiQL took about 12 min at most to analyze *Tileless Map* having about one million lines of code.

Figure 6 shows the analysis time of MultiQL compared to JN-Sum. The x-axis denotes the indices of apps and the y-axis denotes the analysis time in seconds. We omit eight apps JN-Sum fails to analyze. JN-Sum analyzes 12 apps faster than MultiQL, but the difference is less than 2 min. On the other hand, MultiQL analyzes five apps faster than JN-Sum, and JN-Sum spends much time in summary generation for three apps among them. In addition, while JN-Sum fails to analyze all the large-scale apps that have more than about 400,000 lines of code in eight hours, MultiQL analyzes about

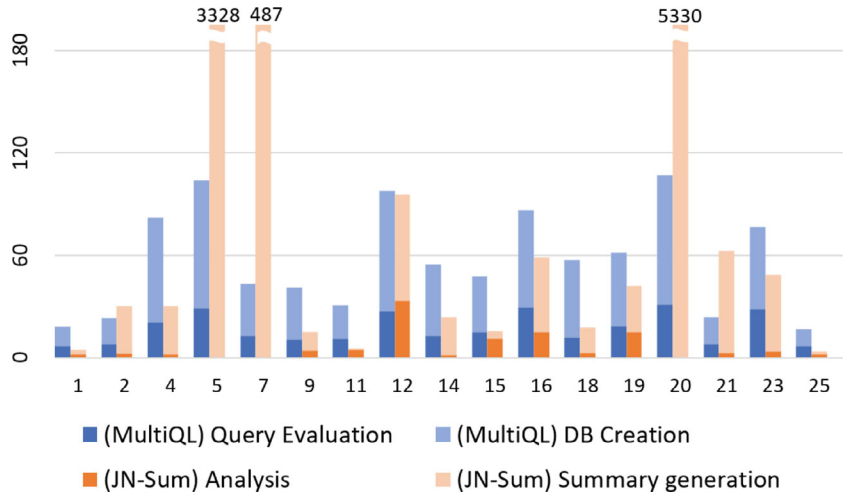

FIGURE 6 Analysis time of MultiQL and JN-Sum

3 million lines of code in about 12 min. JN-Sum shows scalability issues due to its complex semantic summary extraction from C functions.

In summary, the evaluation with the second benchmark suite shows that MultiQL can analyze real-world JNI applications more precisely than JN-Sum. In addition, MultiQL is more scalable than JN-Sum in that it can analyze larger apps much faster than JN-Sum.

5.1.2 | Feasibility: Python-C

As for the Java-C analysis, we evaluate MultiQL for Python-C programs with two benchmark suites: hand-crafted benchmarks to reveal interoperation features and real-world C-Python programs. While several hand-crafted Python-C benchmarks, such as PyCBench used for PolyCruise,[29] are available, they do not apply to MultiQL since its underlying analyzer, CodeQL, cannot support them. Therefore, we developed ExtModuleFlowBench consisting of 20 Python-C programs and used it as the first benchmark suite. We converted each Java-C benchmark in NativeFlowBench into a corresponding Python-C program, except for eight apps containing JNI-specific interoperation features. Additionally, we made five Python-C programs that contain interoperations specific to Python Extension Module.

Table 3 summarizes the analysis results of MultiQL on 20 Python-C programs in ExtModuleFlowBench. **Benchmark** columns show the benchmark names and **Dataflow** columns show the analysis results. MultiQL analyzes dataflows

TABLE 3 Analysis results of ExtModuleFlowBench

Benchmark	Dataflow	Benchmark	Dataflow
callback	✓	multiple_module	✓
cobject_member	✓	noleak	✓
cobject_method	✓	noleak_array	✓
complexdata	✓	nosource	✓
complexdata_stringop	✗	parse_arg	✓
heap_modify	✓	set_field_from_arg	✗
import_module	✓	set_field_from_arg_field	✗
leak	✓	set_field_from_native	✗
leak_array	✗	source	✓
multiple_interactions	✓	source_clean	✗

TABLE 4 Analysis results of real-world Python-C programs

					# API calls	
Repository	**LOC**	**DB creation (s)**	**Query (s)**	**Total (s)**	**# Resolved**	**Total**
Bitarray	12,325	43.311	113.687	156.998	38	48
Bounter	3190	40.112	96.756	136.868	50	72
Cvxopt	83,357	119.047	149.319	268.366	109	276
Distance	1882	39.885	91.717	131.602	6	20
Immutables	5850	40.807	101.466	142.273	20	43
Japronto	8661	52.379	110.656	163.035	12	81
Noise	1766	39.686	93.955	133.641	0	6
Numpy	911,373	339.516	—	—	—	—
Pyahocorasick	7844	41.099	101.465	142.564	33	43
Pyo	272,712	191.341	501.478	692.819	7778	11,009
Python-Levensh	5718	41.278	95.086	136.364	4	13
Python-llist	5411	40.517	100.064	140.581	18	30
Simplejson	6134	41.698	106.307	148.005	14	84
Total					8082	11,725

correctly for 14 benchmarks, but reports false negatives for six benchmarks. MultiQL fails in *complexdata_stringop* and *leak_array* due to the same reason for NativeFlowBench. For the other four benchmarks, MultiQL fails to track dataflows for the fields of Python objects. In the failed benchmarks, the C code changes the values of the fields in Python objects that are passed to the parameters of C functions. MultiQL does not handle such cases because the use of a virtual argument node prevents the propagation of changed values back to the field of the original Python object, and it reports false negatives for them.

The results show that our approach can analyze Python-C programs using various interoperations, just like it can analyze JNI programs using various JNI interoperations.

The second benchmark suite consists of 13 real-world Python-C programs collected from public GitHub repositories. Six of them are the analysis targets of Monat et al.[35] work, and seven of them are a compilable subset of the targets of Li et al.[29] work.

Table 4 summarizes the dataflow analysis results on 13 real-world Python-C programs. The first column shows repository names, the second shows the lines of code, the third shows database creation time, the fourth shows query processing time, the fifth shows the total analysis time, and the last two columns denote analysis results. **# Resolved** column denotes the number of resolved API function calls in C code and **Total** column denotes the total number of Python Extension Module API function calls in C code. Because all the API functions take a Python object as the first argument, we considered an API function call as resolved, when MultiQL finds its first argument correctly.

Out of 13 programs, all programs but one were analyzed within the time limit of 300 h. The only exception was *NumPy*, the largest program in the benchmark suite. We inspected the evaluation log and found that the analysis for NumPy was particularly slow because of the inefficient query optimization of CodeQL's query system. In the remaining twelve programs, MultiQL resolves 8082 out of 11,725 (68.9%) total API function calls. The main reason for failing to resolve the rest of the API function calls was the implicit method calls in Python. Python code can call some methods implicitly, a class destructor for example, and C code can register C functions as such implicitly called methods. Because MultiQL does not handle C functions implicitly called in Python, it fails to resolve API function calls in them.

In summary, the evaluation with the second benchmark suite shows that MultiQL can analyze real-world Python-C programs with high precision. It can precisely analyze the Python object values that C variables can have.

5.2 | RQ2: Usefulness

We evaluate the usefulness of MultiQL by dataflow analysis involving interoperation bugs in real-world Java-C programs and Python-C programs.

5.2.1 | Bugs in Java-C programs

For Java-C programs, we show that MultiQL can detect JNI interoperation bugs in 42 real-world Android apps. We chose to explore the same JNI interoperation bugs in our analysis as those used in previous research:[34,36]

1. *NullDeref*: dereferencing the `null` value of Java in C
2. *MissingFun*: calling a missing Java method from C
3. *TypeMismatch*: declaring a C function with a different signature from its corresponding Java native method
4. *WrongSig*: calling a Java method using a method ID with a wrong signature in C

We implemented a checker to detect the bugs on top of MultiQL using the query language of CodeQL.

Table 5 summarizes the bug detection results of 19 out of 42 apps, on which MultiQL reports possible bugs. The first column shows app names, the second and the third columns show the numbers of true and false positives for *NullDeref*, respectively, and the fourth to the sixth columns show the numbers of true positives for *MissingFun*, *TypeMismatch*, and *WrongSig*, respectively. We omit the columns denoting false positives for the three bugs in the table because MultiQL does

TABLE 5 Bug detection results of real-world Android JNI apps

	Bug kind				
	NullDeref		MissingFun	TypeMismatch	WrongSig
Application	# TP	# FP	# TP	# TP	# TP
APV PDF Viewer	0	1	0	2	0
CrossWords	3	1	0	0	0
Document Viewer	0	0	0	1	0
DroidZebra	0	1	0	0	0
FBReader	0	9	0	1	0
Graph 89	1	0	0	3	0
KeePassDroid	1	0	0	0	0
Lumicall	0	0	3	0	0
Lunary	0	1	0	0	0
Ministro	0	0	0	1	0
Navit	0	0	0	3	0
Hacker's Keyboard	0	1	0	0	0
NetGuard	0	1	0	0	0
ObscuraCam	0	0	2	0	0
PrBoom	0	0	0	0	1
Sipdroid	0	0	4	0	0
Tileless Map	0	2	0	3	0
Tux Paint	0	4	0	3	0
VotAR	0	0	0	0	1
Total	5	21	9	17	2

not report any false positives for them. We confirmed the true and false positives by manually inspecting the source code. We color each cell with non-zero true positives, indicating that the corresponding application had the corresponding bug kind.

MultiQL reports 33 genuine bugs and 21 false positives in 19 apps. It reports five true and 21 false alarms of *NullDeref* in 11 apps, nine true alarms of *MissingFun* in three apps, 17 true alarms of *TypeMismatch* in eight apps, and two true alarms of *WrongSig* in two apps. We manually checked that the false positives come from various over-approximation of the analysis. One of the main causes is conditionally sanitized variables. Many apps use a code pattern that assigns a value to a variable only if the variable has the `null` value. Because our analysis does not support path-sensitivity that analyzes each execution path separately, it reports that the variable may still have `null` even after the conditional assignment. By contrast, JN-Sum fails to detect 12 out of 33 genuine bugs due to memory or scalability issues during analysis.

Figure 7 demonstrates four kinds of JNI interoperation bugs MultiQL detects from real-world apps.

Figure 7A is an excerpt from the app, Graph 89, demonstrating the *NullDeref* bug. The Java code may call a C function nativeInitGraph89 with `null` as its last argument, because the variable `tmp` has `null` when the method `GetInternalAppStorage` returns `null`. The C function calls a JNI function `GetStringUTFChars` with the value as its second argument, without checking whether it is `null`. However, because the JNI specification describes that the second argument of the functions must not be `null`,[37] the function may behave unexpectedly. Note that calling JNI functions with wrong arguments may introduce various unexpected behaviors, since JVMs do not validate the arguments because of performance overhead.[28]

Figure 7B is an excerpt from the app, ObscuraCam, demonstrating the *MissingFun* bug. As described in Section 4.4, JNI has a C function naming convention for JVMs to link Java native methods to their corresponding C functions. While a Java class JpegRedaction declaring a native method redactRegions belongs to a package org.witness.obscuracam.photo.jpegredaction, the corresponding C function is named with a wrong package name org.witness.securesmartcam.jpegredaction . When calling the native method, JVM fails to link it because of the wrongly named C function.

Figure 7C is an excerpt from the app, Document Viewer, demonstrating the *TypeMismatch* bug. While the return type of a Java native method search is List<PageTextBox>, the return type of the corresponding C function search is jobjectArray corresponding to the Java built-in array container. The return type mismatch has no effect on linking between Java native methods and C functions. However, since it is an unspecified case in the JNI specification, interoperations via such native methods may behave differently on different JVMs.[34]

Figure 7D is an excerpt from the app, PrBoom, demonstrating the *WrongSig* bug. The C code tries to get an ID of a Java method OnMessage with a signature (LJava/lang/String;I)V, but the method has a different signature (LJava/lang/String;)V with one String argument. Because the signatures do not match, the C code always receives null instead of a method ID, and this example throws a Java exception. In addition, it may introduce errors in subsequent instructions when using the return value without null checking or calling JNI functions without handling the thrown Java exception.[38]

The evaluation shows that MultiQL can detect various JNI interoperation bugs in real-world JNI apps better than JN-Sum.

5.2.2 | Bugs in Python-C programs

For Python-C programs, we show that MultiQL can analyze inter-language flows to detect interoperation bugs in 12 real-world Python-C programs. We chose to explore the same interoperation bugs in our analysis as those used in Li et al.[29] work. The bugs involve two kinds of inter-language flows: a Python value is passed to C code and used as either (1) an argument to a "dangerous" function, such as strncpy, strncmp, vsprintf, and malloc, or (2) an index to array access. We call such flows *inter-uses*. Among inter-uses, some flows can trigger bugs or security vulnerabilities, which we call *danger-uses*. Because precisely detecting danger-uses is beyond the scope of MultiQL, we implemented a detector to find inter-uses on top of MultiQL and evaluate whether MultiQL can detect all the known danger-uses.

Table 6 summarizes the inter-uses detection results of 12 Python-C programs. The first column shows program names, the second column shows the total number of detected inter-uses, and the third and fourth columns show the numbers of detected and known danger-uses, if any. Out of 12 programs, five have known danger-uses.

```
//EmulatorActivity.java
String tmp = null;
String folder = Util.GetInternalAppStorage(activity);
if (folder != null) {
  tmp = folder + "tmp";
  Util.CreateDirectory(tmp);
}
EmulatorActivity.nativeInitGraph89(..., tmp);
```

```
//wrappercommonjni.c
void nativeInitGraph89(..., jstring tmp_dir) {
   (*env)->GetStringUTFChars(env, tmp_dir, 0); ...
}
```

(A) NullDeref

```
//JpegRedaction.java
package org.witness.obscuracam.photo.jpegredaction;

public class JpegRedaction {
  private native void redactRegions(...); ...
}
```

```
//JpegRedaction.cpp
void Java_org_witness_securesmartcam_jpegredaction_
   JpegRedaction_redactRegions(...) { ... }
```

(B) MissingFun

```
//MuPdfPage.java
private native static List<PageTextBox> search(...);
```

```
//mupdfdroidbridge.c
jobjectArray search(...){ ... }
```

(C) TypeMismatch

```
//PrBoomActivity.java
void OnMessage(String text);
```

```
//jni_doom.h
#define CB_CLASS_MSG_SIG  "(Ljava/lang/String;I)V"

//jni_doom.c
mSendStr = (*env)->GetMethodID(env, jNativesCls,
                        "OnMessage", CB_CLASS_MSG_SIG);
```

(D) WrongSig

FIGURE 7 Four kinds of JNI interoperation bugs

TABLE 6 Inter-use detection results of real-world Python-C programs

Program	Inter-use	Danger-use	
		Detected	Known
Bitarray	36	—	—
Bounter	11	1	1
Cvxopt	84	3	4
Distance	10	—	—
Immutables	41	1	1
Japronto	2	1	1
Noise	0	—	—
Pyahocorasick	5	—	—
Pyo	176	2	2
Python-Levenshtein	11	—	—
Python-llist	0	—	—
Simplejson	1	—	—
Total	377	8	9

```
1  // cvxopt/src/C/cholmod.c
2  static PyObject* getfactor(PyObject *self, PyObject *args)
3  {
4    if (!PyArg_ParseTuple(args, "O", &F)) return NULL;
5    ...
6    if (!PyCapsule_CheckExact(F) || !(descr = PyCapsule_GetName(F)))
7        err_CO("F");
8    if (strncmp(descr, "CHOLMOD FACTOR", 14))
9        PY_ERR_TYPE("F is not a CHOLMOD factor");
10   ...
11 }
```

FIGURE 8 One danger-use that MultiQL did not detect

The result shows that MultiQL can correctly detect eight of the nine known danger-uses. Figure 8 shows one danger-use from the program *cvxopt* that MultiQL did not detect. The call to strncmp on line 8 is an inter-use since the first argument descr comes from Python code. Also, this call is a danger-use because it may incorrectly conclude that the strings descr and `"CHOLMOD FACTOR"` are equal, even if they are not, since string terminator ('\0') is not considered. The correct code should use 15 `(strlen("CHOLMODE FACTOR")+1)` rather than 14 as the last argument. MultiQL did not detect this inter-use simply because the function getfactor was unreachable. Because no Python test files call this function, MultiQL could not find any Python value that descr can point to. We confirmed that when we manually add a Python test file that calls getfactor, MultiQL correctly detects the previously missed danger-use.

The evaluation shows that MultiQL can detect danger-uses that may trigger bugs or security vulnerabilities in real-world Python-C programs.

6 | RELATED WORK

For Java-C program analysis, ILEA[36] extends the Java Virtual Machine Language (JVML) and Jlint, a dataflow analyzer for Java bytecode, to compile both Java and C code to the extended JVML and analyze the integrated programs. Since the modest extension of JVML cannot support the full C semantics like the C memory model, it extremely over-approximates C operations such as reads and writes through pointers. Lee et al.[34] proposed a general approach to analyzing multilingual programs written in both *host* and *guest* languages. Their approach first uses a guest language analyzer to extract

semantics summaries from the parts written in the guest language, C in their implementation, translates and integrates the summaries into a host language, Java, and then performs the whole-program analysis using a host language analyzer.

Researchers also studied binaries rather than C/C++ source code for Java-C program analysis. Fourtounis et al.[39] proposed a lightweight reverse engineering technique to recover Java method calls from binaries instead of performing heavy analyses on binary code. Their reverse engineering generates datafacts of Java method calls from binaries, which can be used in further declarative-style Java analyses. While the approach is lightweight but targets only Java method call identification in binaries, our approach seamlessly analyzes dataflows across language boundaries between Java and C. JN-SAF[32] defines a unified dataflow summary to represent dataflows in both Java bytecode and binary. It extracts summaries from each Java method with a Java static analyzer and from each native function with a binary symbolic execution, and composes the summaries in a bottom-up manner to find data leakages over language boundaries.

Android hybrid apps are written in both Java and JavaScript, taking advantage of the portability from JavaScript and device resource accessibility from Java. HybriDroid,[40] implemented on top of WALA,[41] analyzes Java and JavaScript code seamlessly to detect programmer errors on interoperations and track data leakages across language boundaries. Bae et al.[42] tackled the expensive analysis of HybriDroid and proposed a lightweight type system detecting the same kinds of programmer errors in Android hybrid apps. Jin et al.[43] proposed static detection of code injection attacks from JavaScript to Java. They manually modeled Java frameworks and performed a taint analysis for JavaScript with the models.

Python supports an interoperability mechanism with C.[44] Developers often import performance-critical C code to high-level Python code. Recent work[35] proposed a Python-C analyzer by reusing existing Python and C analyzers built on top of the same framework, MOPSA.[45] They leverage the full features of the analyzers within the same framework to perform precise context-sensitive value analyses. PolyCruise[29] is a Python-C analyzer that enables an efficient dynamic information flow analysis (DIFA). The analysis performs two phases: it first computes symbolic dependencies between sources and sinks, then uses that information to make efficient instrumentations for online dynamic analysis. In contrast to these works, our work focuses on declarative style dataflow analysis.

7 | CONCLUSION

Declarative static analysis has become a widely-used analysis technique but has not supported multilingual programs actively developed in diverse application domains. In this paper, we present a practical extension methodology for a declarative static analyzer that supports multiple languages to analyze multilingual programs. The first step is to create a merged database consisting of multiple logical language spaces. Each language space stores facts transformed from source code written in its corresponding language. Then, the second step is to define language-interoperation rules to derive facts across language boundaries. Our prototype implementation, MultiQL built on top of CodeQL, successfully tracks dataflows over language boundaries in both Java-C and Python-C programs. Using MultiQL, we found 33 true bugs and vulnerabilities from real-world JNI applications, 12 of which are from the applications that JN-Sum, the state-of-the-art Java-C program analyzer, failed to analyze due to the lack of scalability. We believe that our approach is applicable to various multilingual programs, beyond Java-C and Python-C, and to multiple types of analyses.

ACKNOWLEDGMENTS

This research was supported by National Research Foundation of Korea (NRF) (Grants 2022R1A2C200366011 and 2021R1A5A1021944), Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (2022-0-00460), and Samsung Electronics Co., Ltd (G01210570).

AUTHOR CONTRIBUTIONS

Dongjun Youn: Conceptualization, Methodology, Software, Evaluation, Writing - original draft, Writing - review & editing. **Sungho Lee:** Conceptualization, Methodology, Supervision, Writing - review & editing. **Sukyoung Ryu:** Conceptualization, Methodology, Supervision, Writing - review & editing.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in codeql at https://github.com/kaist-plrg/codeql.

ORCID

Dongjun Youn https://orcid.org/0000-0002-5766-2035
Sukyoung Ryu https://orcid.org/0000-0002-0019-9772

REFERENCES

1. Jordan H, Scholz B, Subotić P. Soufflé: on synthesis of program analyzers. Paper presented at: International Conference on Computer Aided Verification, Lecture Notes in Computer Science, Springer; 2016:422-430.
2. Avgustinov P, De Moor O, Jones MP, Schäfer M. QL: object-oriented queries on relational data. Paper presented at: 30th European Conference on Object-Oriented Programming (ECOOP 2016), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik; 2016.
3. Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses. Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. ACM; 2009.
4. Hajiyev E, Verbaere M, de Moor O. codeQuest: scalable source code queries with datalog. Paper presented at: 20th European Conference on Object-Oriented Programming (ECOOP 2006), Springer; 2006.
5. Allen N, Krishnan P, Scholz B. Combining type-analysis with points-to analysis for analyzing Java library source-code. Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis. ACM; 2015:13-18.
6. Allen N, Scholz B, Krishnan P. Staged points-to analysis for large code bases. Paper presented at: International Conference on Compiler Construction, Springer; 2015:131-150.
7. Alpuente M, Feliú MA, Joubert C, Villanueva A. Datalog-based program analysis with BES and RWL. Paper presented at: International Datalog 2.0 Workshop, Springer; 2010:1-20.
8. Dawson S, Ramakrishnan CR, Warren DS. Practical program analysis using general purpose logic programming systems–a case study. Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, ACM; 1996:117-126.
9. Naik M, Aiken A, Whaley J. Effective static race detection for Java. Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM; 2006:308-319.
10. Reps T. Solving demand versions of interprocedural analysis problems. Paper presented at: International Conference on Compiler Construction, Springer; 1994:389-403.
11. Smaragdakis Y, Kastrinis G, Balatsouras G. Introspective analysis: context-sensitivity, across the board. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM; 2014:485-495.
12. Whaley J, Avots D, Carbin M, Lam MS. Using Datalog with binary decision diagrams for program analysis. Paper presented at: Asian Symposium on Programming Languages and Systems, Springer; 2005:97-118.
13. Scholz B, Jordan H, Subotić P, Westmann T. On fast large-scale program analysis in datalog. Proceedings of the 25th International Conference on Compiler Construction, ACM; 2016:196-206.
14. Smaragdakis Y, Bravenboer M. Using datalog for fast and easy program analysis. Paper presented at: International Datalog 2.0 Workshop, Springer; 2010:245-251.
15. Semmle. CodeQL; 2021. https://semmle.com/codeql
16. Meta. Glean: System for collecting, deriving, and querying facts about source code; 2022. https://glean.software
17. Lagouvardos S, Dolby J, Grech N, Antoniadis A, Smaragdakis Y. Static analysis of shape in TensorFlow programs. Paper presented at: 34th European Conference on Object-Oriented Programming (ECOOP 2020), Schloss Dagstuhl-Leibniz-Zentrum für Informatik; 2020.
18. Kochhar PS, Wijedasa D, Lo D. A large scale study of multiple programming languages and code quality. Paper presented at: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE; 2016:563-573.
19. Mergendahl S, Burow N, Okhravi H. Cross-language attacks. Paper presented at: 29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24–28, 2022, The Internet Society; 2022.
20. Grichi M, Abidi M, Jaafar F, Eghan EE, Adams B. On the impact of interlanguage dependencies in multilanguage systems empirical case study on java native interface applications (JNI). *IEEE Trans Reliab*. 2020;70(1):428-440.
21. Oracle. Java Native Interface Specification; 2022. https://docs.oracle.com/en/java/javase/14/docs/specs/jni
22. Foundation PS. Extending Python with C or C++; 2022. https://docs.python.org/3/extending/extending.html
23. Python Software Foundation. Extending Python with C or C++; 2022. https://docs.python.org/3/extending/extending.html
24. Park J, Park J, An S, Ryu S. JISET: JavaScript IR-based semantics extraction toolchain. Paper presented at: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE/ACM; 2020:647-658.
25. Park J, An S, Youn D, Kim G, Ryu S. JEST: N+1-version differential testing of both JavaScript engines and specification. Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE/ACM; 2021:13-24.
26. Park J, An S, Shin W, Sim Y, Ryu S. JSTAR: JavaScript specification type analyzer using refinement. Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE/ACM; 2021.
27. Park J, An S, Ryu S. Automatically deriving JavaScript static analyzers from specifications using meta-level static analysis. Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM; 2022.
28. Hwang S, Lee S, Kim J, Ryu S. JUSTGen: effective test generation for unspecified JNI behaviors on JVMs. Proceedings of IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE/ACM; 2021.
29. Li W, Ming J, Luo X, Cai H. PolyCruise: a cross-language dynamic information flow analysis. Paper presented at: 31st USENIX Security Symposium (USENIX Security 22), USENIX Association, Boston, MA; 2022:2513-2530.

30. Li W, Li L, Cai H. On the vulnerability proneness of multilingual code. Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM; 2022:847-859.
31. arguslab. NativeFlowBench; 2019. https://github.com/arguslab/
32. Wei F, Lin X, Ou X, Chen T, Zhang X. JN-SAF: precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of android applications with native code. Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ACM; 2018:1137-1150.
33. F-Droid. F-Droid - Free and Open Source Android App Repository; 2019. https://f-droid.org
34. Lee S, Lee H, Ryu S. Broadening horizons of multilingual static analysis: semantic summary extraction from C code for JNI program analysis. Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, IEEE/ACM; 2020:127-137.
35. Monat R, Ouadjaout A, Miné A. A multilanguage static analysis of python programs with native C extensions. Paper presented at: Static Analysis Symposium (SAS), Springer, Chicago, Illinois, United States; 2021.
36. Tan G, Morrisett G. ILEA: inter-language analysis across Java and C. Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, ACM; 2007:39-56.
37. Oracle. Java Native Interface Specification - Chapter 4. JNI Functions; 2021. https://docs.oracle.com/en/java/javase/14/docs/specs/jni/functions.html.
38. Oracle. Java Native Interface Specification - Chapter 2. Design Overview; 2021. https://docs.oracle.com/en/java/javase/14/docs/specs/jni/design.html
39. Fourtounis G, Triantafyllou L, Smaragdakis Y. Identifying java calls in native code via binary scanning. Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM; 2020:388-400.
40. Lee S, Dolby J, Ryu S. HybriDroid: static analysis framework for Android hybrid applications. Paper presented at: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE/ACM; 2016:250-261.
41. IBM. T.J. Watson libraries for analysis; 2021. http://wala.sourceforge.net/wiki/index.php/Main_Page
42. Bae S, Lee S, Ryu S. Towards understanding and reasoning about android interoperations. Paper presented at: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE/ACM; 2019:223-233.
43. Jin X, Hu X, Ying K, Du W, Yin H, Peri GN. Code injection attacks on HTML5-based mobile apps: characterization, detection and mitigation. Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM; 2014:66-77.
44. van Rossum G. the Python development team: Python/C API Reference Manual; 2021. https://docs.python.org/3.8/c-api/index.html
45. Journault M, Miné A, Monat R, Ouadjaout A. Combinations of reusable abstract domains for a multilingual static analyzer. Paper presented at: Working Conference on Verified Software: Theories, Tools, and Experiments, Springer; 2019:1-18.

How to cite this article: Youn D, Lee S, Ryu S. Declarative static analysis for multilingual programs using CodeQL. *Softw: Pract Exper*. 2023;53(7):1472-1495. doi: 10.1002/spe.3199