Reviewing: 1  
  
Comments to the Author  
Overall evaluation:  
This is an interesting paper which extends the tool CodeQL to support multi-lingual programs (resultant system seems to be called MultiQL).  
This extension is well evaluated, according to various research questions, using a range of benchmark suites, and compared to existing tools JN-SAF and JN-Sum.  
(The latter tool seems to be an improved reimplementation of JN-SAF, by the same group as the author.)  
The results show that adding multi-lingual features to CodeQL gives better performance than previous tools JN-SAF and JN-Sum.  
This ideas are worth publishing in SP&E.  
The English is generally of high quality.  
  
However, I argue that the paper needs some restructuring and improved presentation before being accepted to SP&E.  The reason is that it is currently hard for a reader unacquainted to the tools and concepts used by the author to read the paper linearly.  
Overall, I recommend acceptance subject to presentation improvements.  Is this a major or minor revision?  This is unclear to me: it is conceptually simple (make the paper more accessible), but it needs changes in many places in the paper.  
For example the Introduction and technical background does not seem to introduce the material well; as examples I would note that the word Datalog does not appear until p17 when its ideas are pervasive, and "Declarative Program Analysis" is not very well defined.  
(I know this is an upcoming topic, but there is no Wikipedia entry for  "Declarative Program Analysis" and google only finds 4250 hits (including the quotes).  So a tutorial introduction is required.  
I will note other more minor oddities below, but to me Sections 1 and 4 need the greatest change.  
  
I suspect the first author is relatively inexperienced;  
I wish you luck, and perhaps my advice can be summarised like this.  Think of a paper as a function, f, from the reader's knowledge before reading the paper linearly, to his/her knowledge afterwards.    Of course f(author) = author.  But the question is what is f(x) for other readers, x?  We would like f(x) = f(author), noting that f(x) = x is undesirable for for readers x unfamiliar with the subject!  
Please aim to make the paper accessible to a large readership who know something about program analysis and the difficulties of multi-language coding, but perhaps have never seen concepts in Datalog (or CodeQL).  
==========  
Title: I wonder whether "Multilingual static program analysis using CodeQL"  
might attract more people to read the paper.  
Or even "Multilingual static program analysis: Declarative CodeQL vs <whatever>"?  
- We agree. Mentioning CodeQL in the title would be more appealing to the potential readers. We decide to change the title into “Declarative static analysis for multilingual programs using CodeQL”.

Abstract [and many other places].  I was brought up with compilers which have a source language, a target language and an implementation language.  While I agree that this distinction is less clear for interpreters and static analysers (which do not have target languages) I would argue that all your uses of "target language" should be "source language" for consistency.  
(By all means argue, but if you wish to keep "target" then please add a sentence or two justifying this.).

- We agree that the term “target language” would be a confusing term. We decided to refrain from using the term “target language” in most parts, and any remaining “target language” is used in the context of “analysis target language”, to avoid the confusion.

Abstract: perhaps mention Datalog-style as this is generally well known.

-> 막상 넣으려고 하니, abstract의 흐름이 끊기는 느낌입니다.

Intro: please could you re-write this.  The first paragraph assumes I am an expert  
(but I did not know that DOOP was a declarative program analyser, nor exactly how it compared with Soufflé nor with CodeQL, or even Datalog).  It will put off many readers who would benefit from the paper and enjoy its results.

|

p2: "rules derive new facts".  There are just too many assumptions on the reader here,  
especially in an introduction.

|  
Section 1 and 6.  There is always a challenge in explaining related work.  Mostly the fine details need to go in Section 6, see e.g. above above CodeQL and Java.  But Section 1 (and indeed 2 and 4) need to draw the big picture, explaining Declarative -> Datalog -> QL with CodeQL as a commercial implementation.  As an example where you lose the "big picture", search for "rules" in your paper.  It's fine if you already know about the QL (or datalog) but otherwise your readers are left guessing.

|

Section 6.  This tells me various things I'd have loved to know earlier in the paper.  
Please move tutorial stuff earlier, but keep detailed comparison with your work here.  
One issue, you've not said "CodeQL or MultiQL" e.g.  
". Avgustinov et al.10  proposed QL, a declarative and object-oriented query language that can be compiled into Datalog and runs on a relational database.  
Using QL, they developed a static analyzer [Add: CodeQL] that is scalable to large-size programs with millions of lines of code. Our approach [Add: MultiQL] inherits the benefits from declarative-style analysis and extends analysis targets from monolingual to multilingual programs."  
Note my suggested additions to the wording.  I think \*you\* know that your analyser is called MultiQL, but it's important to teach the reader that too.

- We restructured the introduction, mainly inspired by your comment. We first explain about the declarative analysis itself, then introduce the examples of actual commercial implementations, and illustrate the effectiveness of the declarative analysis.

p2: "CodeQL was originally designed for Java program analysis".  That's not what a reader needs to know.  What they need is words like:  
See: <https://codeql.github.com/docs/ql-language-reference/about-the-ql-language/#:~:text=The%20syntax%20of%20QL%20is,in%20QL%20are%20logical%20operations>.  
- This comment has two major points: 1) Regarding the unimportant information: The main topic of this paragraph is to emphasize that the analysis target is expanding from one language to multiple languages, and the information was there to explicitly demonstrate that point. We rewrote the paragraph to make it clearer.

2) Regarding more detailed explanation for CodeQL: We restructured the introduction, as mentioned before.

p2: you mention Glean twice, and this just distracts -- section 6 is fine for it.  [Unless you want to say in the intro "increasing commercial interest" instead of its current context words]  
- The main point of the second paragraph is about extending the analysis target, and we rewrote the topic sentence to make this point clearer.

p2 ""We show the practical usefulness of our analyzer in the sense that it detects dataflow-related bugs at language boundaries of real-world multilingual programs."  
A bit weak!  
How about  "Evaluation of our analyser, MultiQL,  
[perhaps: at  finding dataflow-related bugs in]  
on benchmark programs involving cross-language calls shows it to be faster and more accurate than the established tool <whatever>".  
- We are cautious about saying that ours is “faster or more accurate”, because that is not necessarily true. Instead, we emphasized that there are brand-new bugs that were not detected by the previous analyzers.

Figure 1.  This figure is fine, but I'm not sure I like the "Facts" box.  Perhaps the accompanying text needs to clarify that we populate a database with facts derived from a program syntax, and then use rules to transitively close these/take a fixed point?  
I thought there was in general a problem with forming a full database of facts  
when queries might only refer to a small subset of these.  Hence "exhaustive" vs "demand-driven" techniques, e.g.  
\* <https://link.springer.com/chapter/10.1007/3-540-45937-5_5>  
but there is more recent work.  
Perhaps you should clarify whether CodeQL is an "exhaustive" analysis -- the Facts database is fully populated before any query is accepted?  
[As a side-question, I believe that points-to (or alias) analysis can result in huge databases when exhaustive analysis is used; it is the case that your analyses produce sensibly-sized databases so that exhaustive analysis isn't a problem?]  
- 1) Regarding how facts are generated, we also think that the explanation of how facts are generated are not sufficient. We added more detailed explanation to the first paragraph. 2) Regarding the "exhaustive" vs "demand-driven", CodeQL’s dataflow analysis is in fact “demand-driven”. It first inspects the query, and does not generate facts which are not require to get the final result of the query. (Therefore, CodeQL can generate sensibly-sized database in reasonable time for the real-world programs, as we showed in evaluation section.) However, this is a CodeQL-specific implementation detail. This section is supposed to be a high-level introduction of how a general declarative style static analysis work, so we explain in a way of exhaustive analysis.

p3.  I found "ln1" confusing.  I hadn't realised it was a variable rather than a shorthand for "line 1".  perhaps "lnA" and lnB" might work better?  Or even "LnVar1"? Partly the reason is that the syntactic conventions of QL/Datalog have not been established.  
- We agree. We changed the variable names into more explicit ones: `lineNumA` and `lineNumB` for section 2, and `lineNumCall` and `lineNumFunc` for section 3. (Sidenote: there aren’t any syntactic conventions regarding the variable name.)

p3. You talk about IR here, but elsewhere talk about source.   I agree this is not important, but it's helpful not to confuse the reader -- a footnote could clarify that by  
"source language" you also include IRs, perhaps JVM, Dalvik and the like?  
Can CodeQL extract facts equally from Java source and JVM code?

- IR관련 얘기는 아예 지우는 게 낫나라는 생각이 들었습니다.  
  
p3. "We can define a syntactic fact of the form FunctionAt(ln, name)".  Yes, fine.  
But we don't define its \*instances\* one-by-one for the database; they are automatically added.  But how is this related to you defining a "syntactic fact" above?  
THis is really the difference between defining a scheme and defining an instance.  
p3. "Step 2: Defining rules. The next step is to define rules to generate new facts out of known facts."  This isn't quite right.  The rules are \*defined\*  in advance but in step 2  
these definitions are \*used\* to generate new facts.  
- You are right. It was wrong to use the term “define” here. We changed these inappropriate terms in those paragraphs.

p3. "The rules are usually evaluated in a bottom-up and modular manner".  I'm not sure I understand "modular" here, and "bottom-up" here really just means "a rule fires when all the facts on its RHS hold, and adds a new fact to the database", and  
"rules are never combined with other rules only with facts", right?  
[The latter could happen in richer inference systems than Datalog]

- In fact, “modular” and “bottom-up” involves more CodeQL specific implementation detail, but we think that this information is rather irrelevant and confusing to be inserted here, and we removed it. (FYI: What we originally intended to say is described here: https://codeql.github.com/docs/ql-language-reference/evaluation-of-ql-programs/#:~:text=organized%20into%20a%20number%20of%20layers)  
  
p3. ". When accepting the query as an input, the query system finds all 48  
values derivable from the current set of facts for the variable X."  
Please be more precise about "current set of facts".  
Emphasising "exhaustive program analysis" [in that the DB is fully populated  
before any query] would help.  
- We decided to consistently use the term “set of known facts” for referring to the said set (also in other places in Section 2).

In general, section 2 is a good explanation of what happens in Datalog-like  
languages (but of course your paper hasn't even said "Datalog" yet).  
- Good point. We added the expression “Datalog-like” in the start of the section, which would be helpful to understand the context.  
  
Figure 2: This because clearer to me now I've convinced myself that you are doing exhaustive analysis and that "Facts" is the transitive closure of syntactic facts and the additional facts resulting from the three rule-sets.

- That is correct. We rewrote the paper so that it can be clear in the earlier part of the paper.  
  
Fig 3: The "subfigure annotation [\textbf{a}] is ugly, and doesn't correspond to your texts which says "(a) Python".  This applies to other subfigures too.  
Fig 3: please change "pass" to "return None" as this is more generally readable  
(and equivalent here).  
- Done.

sec 3.2.  If you're using x for a variable, then why not use k for a constant

(rather than "elem").   It's odd to have "x | elem".

- We decided to use the symbol “v” for denoting the values.

Also remind the reader about the overbar = sequence convention.

-Done.

Sec 3.2: I didn't understand the RHS of the "RULE" syntax.  Where does the negation symbol come in, and the question mark?  It would perhaps be simpler to write this out  
r :- A1, ... An where each A is either an f or a negated f (\neg f).

- That sounds better. We fixed the syntax of RULE.

You need to say something about not using negation inside recursion, perhaps as a footnote as it's not something you revisit.  
- Very good point. We added the footnote regarding the recursion with the negative negations. (known as the stratification requirement)

Sec 3.2 [perhaps the same point] "The optional prefix negation ¬ denotes the negative hold condition of a fact: the negated fact ¬𝑓 holds if the fact 𝑓 does not hold"  
I don't know what a "hold condition" is.

- It is replaced as “included in the set of known facts”

page 5: the query "?- callEdge(X,Y)" suddenly appears.  What are X and Y?  How do they get named?  Are they variables (seemingly not)?

- They are variables. Query finds the value assignment for X and Y [X -> v1, Y -> v2], such that callEdge(v1, v2) is included in the set of known facts.  
  
p6: "Finally, when making the same query ?- CallEdge(X, Y), the query system now produces (X, Y)∈ {(6, 2), (3, 7), (8, 4)} as function call relation results"  
Perhaps the word "same" really means "running example query" or similar (and maybe even a backwards reference to it).  Plain "Same" is unclear here.  
- It is indeed same query to the ones we made for monolingual analysis. A query is a set of facts with variables, and it is same for both cases. We added an extra explanation that the query is same as the previous ones.

Fig 6.  What are predicates?  Is "Merged DB" what you previously called "Facts"?  How does this refine fig 2?

- In this figure, “predicates” are the combination of rules and facts and “Merged DB” is the “syntactic facts”. There is a slight discrepancy between the previous figures because the implementation of CodeQL does not “explicitly” collect the known facts, in contrast to the conceptual explanation.  
  
\* Section 4.1.  I did wonder whether a short intro to CodeQL syntax might work as a subsection of section 2.  My problem here is that I don't know QL, but I do know  
Datalog, and your intro to QL is too terse for me.  E.g.  
"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 define rules by defining “predicates” and “classes.”  
This needs relating to Datalog, and perhaps the essential parts of QL explained as a "subset".  Your text here is weak.  It says you "[define] the predicate isOneOrTwo:"  
but in the next sentence you say "defines the class OneOrTwo".  Perhaps this is clever overloading, but it confuses me, one of your readers.  
It's OK to refer to "Avgustinov et al", but this should be "for more details"  
but not to excuse a weak explanation.

\* p6.  I didn't understand the language in section 4.4.  Is it CodeQL?  
Maybe you need to be clearer that Section 4 just explains enough CodeQL  
(and also relates it to Datalog in section 2) so that the reader can understand  
the code in section 5?

\* Sect 4.5 I don't think you've summarised enough CodeQL for me to understand this.  
As I say, perhaps factoring your explanation of core CodeQL into "fundamental stuff" in section 2.2, and language modelling here, may help.  
- The reason that we did not introduce CodeQL earlier is that the explaining CodeQL is implementation-specific detail, and not necessary for explaining the main topic of this paper.

Still, we think that the explanation of CodeQL was indeed insufficient. We added a bit more detailed explanation about CodeQL at the start of Section 4. We also removed some unnecessary codes in 4.4 to make it easier to understand.

Sec 4.2 "trap files". This seems a strange name.  Often it's a good convention to  
use italic for the first use of a definition \emph{trap files} rather than quotes to clarify  
it is a technical word rather than a simile.  
- Done.

p6: ". Note that both trap files may have tables with the same name. "  
Does this matter if it's just a debugging format?  Or are trap files a form of database  
(a representation of facts)?  Then you say  
"For example, if both trap files have tables named @params, \*\*\*we\*\*\* rename the table 14 from C as @c\_params, and the table from Java as @java\_params. "  
I'm happy with this.  But who is the "we".  Do you program this somehow -- does the MultiQL implementation say "if two languages create fact schema T then rename T as @lang1\_T and @lang2\_T"?

- 1) Regarding trap files: Yes, it actually matters, because the table names are not only for debugging purpose. As you mentioned, it is the actual representation of facts. This human-readable database is transformed into the binary database. 2) Regarding \*\*\*we\*\*\*: Good point. It is in fact MultiQL doing it automatically. We changed it from “we” to “MultiQL”, along with other places.

p10: I think spell out "inter-/intra-flow analysis" as "inter-/intra-language flow analysis".

- Done.  
  
Table 1. I think you should distinguish the X's into "false positive" and "false negative" columns  
- Done

section 5.1: I'm very impressed at the serious benchmarking you did.  
But please augment  
" We evaluate the feasibility of MultiQL by dataflow analysis on two benchmark suites for each of Java-C and Python-C analyses"  
with the names of the benchmark suites and say you'll explore them in 5.1.1 and 5.1.2.

-Done.  
  
sec 5.1.1 "On the contrary" -> "By contrast" here -- and I think elsewhere (do check).  
- Done.

sec 5.1.1. "In addition, the scalability of MultiQL is comparable with JN-Sum in that it can analyze larger apps much faster than JN-Sum."  
seems a curious self-referential sentence.

-   
  
BTW, I'm happy you calling Benchmark Suites "Analysis Targets", it's just  
"target language" which felt wrong to me.

- Indeed. As mentioned before, we reduce the use of the term “target language”.  
  
5.2.  
"We choose our target JNI interoperation bugs as the same as the targets of the client analysis of previous research14,16: "  
Clumsy sentence.  "We chose [past tense] to explore the same JNI interoperation bugs in our analysis as those use in previous research[...]"

- Thanks for the suggestion. That sounds much better.  
  
I'm still a little unclear exactly how many benchmark suites you used.  3?  or 4?  
List them?  
- I’m not exactly sure about what does the “benchmark suites” referring to in this comment, but if it is about the RQ2, It is just one benchmark suite, the apps downloaded from benchmark.

Table 5.  Does this benchmark suite have a name?  Perhaps give it one so you can  
reference it multiple times?

- It does not have a specific name, since it is basically a set of JNI apps downloaded from F-Droid. Perhaps

Also, what does colour denote?  Green = good?  why does Graph89 have two  
greens?  
Perhaps review the table.  
- Green is used to visualize any non-zero true positives found. Graph89 have two greens, because two different kinds of bugs were detected from Graph 89. We added the description about color.

p15: "Figure 7a shows a pattern in Graph 89 with the NullDeref bug. "  
Here and elsewhere you mean [concrete] "an extract from" rather than  
[abstract] "a pattern in".  
It would be more readable to say "extract from the \emph{Graph 89} benchmark with..."  
p17: patterns -> extracts again multiple times)

- That sounds better. We changed it.

p15: "Figure 7 demonstrates four kinds of JNI interoperation bugs MultiQL detects from real-world apps using simplified code."  
Perhaps "Figure 7 demonstrates four kinds of JNI interoperation bugs MultiQL detects.  
It examines NN simplified bodies of code, each extracted from real-world apps."?  
- This sentence was misleading. “using simplified code” is coupled with “demonstrates”, not “detects from”. The analysis was performed on the unchanged original code, and the bug detected from the analysis is demonstrated in simplified code. We removed the expression “using simplified code”.  
  
p18: ". 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."  
Failed to analyse?  Or Failed to detect [false negative?]

- We mean “failed to analyze”. JN-Sum failed to analyze some of the applications due to time or memory scalability, and the brand new 12 bugs MultiQL detected were included in those applications.   
  
References:  
[14] "c code" -> "C code"  
[31] give volume number.  LNCS?  
[37] "html"->"HTML".  
[38] "van" is not a name, His family name is "van Rossum" [which sorts as "R", not "v"]  
and his given name is "Guido". [People often get this wrong].  See bibtex for how to fix this.  
- All fixed.

Summary: A nice piece of research, with just a bit more effort required on presenting it clearly.  
  
Reviewing: 2  
  
Comments to the Author  
## Paper summary:  
This paper proposes a declarative static analysis framework for multilingual programs, with the current implementation supporting Java-C and Python-C programs. The framework extends CodeQL, a popular declarative static code analysis framework for individual programming languages. The key idea is to start with the language-specific syntactic facts and further-fact-extraction rules, followed by inter-language analysis as reflected through language interoperation rules. To show the application of the proposed declarative static analysis of multilingual programs, a bug checker was implemented for Java-C programs. Empirical results show the merits of the proposed methodology in extracting data flow in both Java-C and Python-C programs, and detecting interoperability bugs in Java-C programs.  
  
## Overall assessment  
  
Strengths:  
  
+ The topic is relevant and significant.  
  
+ The tool extending codeQL to support multilingual programs is useful and very interesting.  
  
+ The idea of reusing individual languages’ analysis facts and rules and letting additional rules to handle language interoperability is a reasonable idea of obtaining cross-language analysis facts.  
  
+ The evaluation considered a practical application of the declarative multilingual static analysis.  
  
Weaknesses:  
  
- Soundness issues with the static analysis need to be further addressed, with respect to the great diversity of language combinations and language interfacing mechanisms shown in prior work.  
  
- The novelty of the proposed IR needs to be better justified with respect to closely related work.  
  
- It is not clear how the proposed methodology would generalize (to the vast space of the multi-langugage world) given the static nature of the analysis and the dynamic nature of so many popularly used languages involved in real-world language combinations.  
  
- More fine-grained evaluation metrics should be considered in the evaluation.  
  
- The application evaluation only covers one language combination (i.e., Java-C).  
## Detailed comments  
Multilingual programs are growing increasingly prevalent in the real-world software ecosystem. Thus, tools supporting analysis and quality assurance of multilingual programs is a timely and highly valuable effort. The idea of extending a popular static analysis engine like CodeQL, which has already gained wide attention and adoption for several different induvial programming languages, to support multilingual programs is a nice one. The paper is well written and easy to follow, and the structure is reasonable. The authors have also been considerate in including an application study of using the declarative multilingual program analysis for interoperation bug check. Results show merits of the proposed methodology and the bug-checking application.  
  
In the meantime, the paper may need to be carefully improved to address the following concerns.  
  
First, the proposed methodology may suffer from practical soundness issues. Take the language combination of Python and C, which is one of the two combinations considered in this paper, for example, Python has rich dynamic constructs causing oftentimes statically invisible code [1]. As a static analysis, the absence of this kind of code at compile time clearly causes unsoundness of the analysis, which should have but have not been addressed in this paper. Not only would this lead some syntactic facts and further rule-induced facts to be missing in individual language (Python) units, missing language interactions would also cause some language interoperations to be missing. Please refer to [1] for some illustrations and discussion.  
[1]"{PolyCruise}: A {Cross-Language} Dynamic Information Flow Analysis." In 31st USENIX Security Symposium (USENIX Security 22), pp. 2513-2530. 2022.  
- Our approach is indeed unsound. (…)

Second, as recent prior study [2] clearly shows, the multi-language world is vastly diverse, including hundreds of different languages and even greater numbers of possible language combinations in constructing multi-language software. How would the proposed methodology generalize to other languages and language combinations.  
[2]"Understanding language selection in multi-language software projects on GitHub." In 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 256-257. IEEE, 2021.  
  
  
Accordingly, a variety of language interaction mechanisms are present in real-world multi-language software systems, far from being limited to explicit (native/foreign) function invocations. Please see [3] for a taxonomy of such mechanisms.  
[3] "On the Vulnerability Proneness of Multilingual Code." In ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). 2022.  
  
A key question here is how the proposed approach handles the variety of language interaction mechanism beyond those in Java-C and Python-C programs (mostly via explicit function invocations)? And how the language-interoperation rules could be defined to handle the other kinds of language interfaces and interoperability? To help see such diversity of interfaces and the interfacing mechanisms to facilitate the definition of those results, authors may find it helpful to use the PolyFax tool [4] to extract the mechanisms.  
[4] "PolyFax: A Toolkit for Characterizing Multi-Language Software." In ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). 2022.  
  
One merit of using an IR like that in this paper is that they extract information useful for further analysis while not immediately specific to particular languages’ syntax or semantics. This idea has been proposed in [1], although implemented for Python-C but easily extensible for other languages. Please discuss how the IR in this paper is different from the PolyCruise-IR [1], in order to better justify the novelty of the proposed work.  
  
The current static analysis is only for call graph construction. First, if this is the case, why have such a broad title ‘static analysis’; I would be more specific by just saying ‘declarative call graph construction’ or at most ‘declarative control flow analysis’ in the title. Second, what would need to be done if a user wants to use MultiQL to implement other kinds of static analysis of multilingual programs? This is important to answer since after all this paper is extending CodeQL which does support a wide range of static analysis.  
  
  
For evaluation, the authors created a microbench called ExtModuleFlowBench for Python-C programs. This is much smaller than PyCBench in [1], not only in terms of #benchmarks but also in terms of static analysis relevant features (e.g., object sensitivity, dynamic invocation). Also, since this is an existing, previously published benchmark, it would be less biased to use this external benchmark (just like the NativeFlowBench in [12]) instead of (or at least in addition to) the one created by the authors themselves. The PyCBench comes with ground truth, allowing for evaluation of precision and recall.

- Thanks for the kind suggestion. We inspected the suggested benchmark, and applied our analyzer to the benchmark. Unfortunately, most of the benchmarks were not applicable for our analyzer. Our analyzer is a simple dataflow analyzer that only detects source-sink relation, and the analysis that requires the run-time values (such as detecting buffer overflow) are not scope our analyzer. After selecting only applicable tests, the benchmark suite has relatively smaller number of test cases, and the ways of interoperation happening between language boundary in those tests were not various enough. Therefore we decided to use the benchmark we crafted, as it can better test the ability of our analyzer.

Finally, this paper evaluates the usefulness of the proposed analysis through call graph based interoperation bug detection, but only for Java-C programs. Since the analysis for Python-C has been implemented, why only evaluating the application against Java-C programs? Why not including Python-C programs? After all, the contribution is pitched as a framework extending CodeQL, supporting the application for just one language combination seems to be too skimp relative to a ‘framework’. I don’t see why “Because no existing work reports interoperation bugs in Python-C programs” justifies the decision of dismissing the Python-C application study.

- 지금 보니, 이 부분이 가장 문제인 것 같습니다. 리뷰어가 이미 PolyCruise라는 C-python 분석기를 언급을 했는데, 알고 보니 해당 논문에서는 이미 real world C-Python 프로그램의 버그를 찾고 있었습니다.   
  
Compared to the authors’ prior work JN-Sum [14], it seems that MultiQL’s main merits lie in scalability and efficiency? Please discuss why this is the case. What makes JN-Sum so much worse?