A Static Analysis Approach for Cross Language FiT Bugs Detection

Li Shengyang, Kenta Ishiguro, Kenji Kono
Keio University, Japan

Modern software is often developed by multi-programming languages (MPLs) to benefit their features and reuse existing libraries. A survey [1] on recent open-source projects shows that over 50% of them are developed by more than 2 languages. This inevitably introduces MPL bugs, which stem from fundamental mismatch of semantics of MPLs.

```
jclz = (*env)->FindClass(env, "JAVA_CLASS");
jmtd = (*env)->GetMethodID(env, jclz, "CALLEE_METHOD", "ARGS");
returnValue = (*env)->CallIntMethod(env, obj, jmtd, arg1, arg2);
//if((*env) -> ExceptionOccurred(env)){return;}
doSomething(returnValue);
```

Listing 1. Omit exception handling

Lst. 1 shows a classic cross language call between Java and C. In the code snippet, exception handling after calling the Java code is omitted. In the Java world, uncaught exceptions can directly interrupt program execution, but in the C world, there is no concept of exception and thus the program continues to run, which may cause unstable behaviors and bugs.

However, a large portion of MPL bugs can be detected if the entire project is written in a single programming language. In the above example, the Java compiler can detect missing exceptions if the C portion is coded in Java. Our observation among ten open-source MPL projects, FiT bugs – a type of bug that is relatively simple and can be detected without complex analysis – occupy a large portion.

Proposed Approach. We propose a scalable static analysis tool to find FiT bugs in projects developed by MPL. Existing multi-language bug detection tools [2, 3] focus on retrieving information from binaries to make static analysis possible, resulting in the loss of some semantics and additional complexity. Our approach prevents losing cross-language context by unifying simplified intermediate representations(IRs) from MPL and linking isolated parts to make a global view.

By using existing compilers and frameworks, single language IR based function summaries and control flow graphs can be easily obtained. As what we focus on is FiT bugs, many obscure language features are not necessary. Only the control flow graph, function calls, and some basic variable assignments are retained in our simplified IR. Then, based on cross-language customized linkers, the missing MPL context is supplemented and comprehensive function summaries are generated. Thus, the isolated MPL parts of one project have been unified into interoperable function summaries represented by simplified IR. Subsequently, analysis passes will be executed in the unified function summaries to detect various patterns of MPL FiT bugs, and report possible bugs.

Compared to prior work, simplified IR in our approach ensures scalability for multiple languages, as it is not complex to parse a normal IR to simplified IR. Besides, all the analysis is based on unified function summaries, which provides possibilities for extending analysis processes to deal with different bug patterns. Moreover, the entire architecture is component-based, allowing adding and removing on demand, which is undoubtedly friendly for extension.

We have developed a prototype program focusing on Java Native Interface (JNI) related bugs. Fig. 1 shows the workflow of our prototype. In the prototype program, LLVM and Sootup are used to parse C/C++ code and Java code into simplified IR based function summaries. Then, our JNI tailored linker completes details and cross language context. Finally, analysis passes are performed on the completed function summaries.

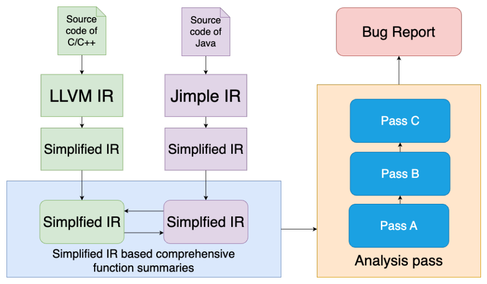

Figure 1. Workflow

Preliminary Results To evaluate the feasibility of our approach, we are currently conducting preliminary validation experiments. We use our prototype project found some bug like miss exception handling, miss return value checking successfully on a micro benchmark.

References

[1] Wen Li, Austin Marino, Haoran Yang, Na Meng, Li Li, and Haipeng Cai. 2024. How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. *ACM Trans. on Software Engineering and Methodology* 33, 3 (2024), 1–46.

[2] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. 2022. JuCify: a step towards Android code unification for enhanced static analysis. In *Proc. 44th Int. Conf. on Soft. Eng.* 1232–1244.

[3] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. JN-SAF: Precise and Efficient NDK/JNI-aware Inter-language Static Analysis Framework for Security Vetting of Android Applications with Native Code. In *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 1137–1150.