On the Impact of Inter-language Dependencies in Multi-language Systems

Empirical case study on **Java Native Interface Applications (JNI)**

Manel Grichi*, Mouna Abidi*, Fehmi Jaafar†, Ellis E. Eghan*, Bram Adams‡
*Department of Computer and Software Engineering, Polytechnique Montreal - Canada
†The Computer Research Institute of Montreal - Canada
‡School of Computing, Queen's University - Canada
*{manel.grichi,mouna.abidi,ellis.eghan}@polymtl.ca, †fehmi.jaafar@crim.ca, ‡bram.adams@queensu.ca

***Abstract*—Nowadays, developers are often using multiple programming languages to exploit the advantages of each language and to reuse code. However, dependency analysis across multi-language is more challenging compared to mono-language systems. In this paper, we introduce two approaches for multi-language dependency analysis: S-MLDA (Static Multi-language Dependency Analyzer) and H-MLDA (Historical Multi-language Dependency Analyzer), which we apply on ten open-source multi-language systems to empirically analyze the prevalence of the dependencies across languages *i.e.,* inter-language dependencies and their impact on software quality and security.**

Our main results show that: the more inter-language dependencies, the higher the risk of bugs and vulnerabilities being introduced, while this risk remains constant for intra-language dependencies; the percentage of bugs within inter-language dependencies is three times higher than the percentage of bugs identified in intra-language dependencies; the percentage of vulnerabilities within inter-language dependencies is twice the percentage of vulnerabilities introduced in intra-language dependencies.

***Index Terms*—Multi-language, Static code analysis, change history, Dependency analysis, Co-changes**

I. Introduction

Nowadays, developers are often choosing to use multiple programming languages to implement features in software systems. These systems need to adapt to new changes continually and fix issues [1]. Software changes may lead to the introduction of quality or security issues. Thus, it is important to perform change impact analysis during the software system maintenance [2]. A fundamental part of program maintenance consists of analyzing dependencies between source-code entities [3]. Such dependencies reveal the entities potentially impacted by a maintenance task, assist developers in their maintenance activities, and allow tracking the propagation of their changes [4].

Since each programming language has its own rules (*i.e.,* lexical, semantic, and syntactical), change impact analysis of multi-language systems becomes more complex, and the maintenance activities become more challenging [1]. This is because generic dependency analyses are no longer able to follow (in)direct function calls in order to determine dependencies, i.e., developers need to understand the specific calling convention between, for example, Java and C++ [5]. In contrast to the dependency analysis in mono-language systems that has been studied extensively [6], [7], [8] using a variety of techniques such as static code analysis and mining software repositories, dependency analysis for multi-language systems is not well established yet [5] and is still subject to further research [9].

In this paper, we empirically study ten Java Native Interface (JNI) open-source multi-language systems to identify the inter-language dependencies, analyze their prevalence in multi-language systems, and their impact on software quality and security. We focus, in this study, on JNI since it is a mature technology (appeared in 1996 with the JDK v1.0) that is often used in industry to call native C/C++ functions from Java and vice versa. Abidi *et al.* [10] show that Java - C/C++ is the most common combination of languages used in multi-language systems.

To identify the inter-language dependencies, we introduce two approaches, which we applied on the ten JNI systems: S-MLDA (Static Multi-language Dependency Analyzer) that performs a static dependency analysis using heuristics and naming conventions to detect direct inter-language dependencies (a direct static relationship between two multi-language files), and H-MLDA (Historical Multi-language Dependency Analyzer) that performs historical dependency analysis based on software co-changes to identify the indirect inter-language dependencies (a relationship that could not be detected statistically *i.e.,* hidden). Consequently, we address the following research questions:

RQ1. *How common are direct and indirect inter-language dependencies in multi-language systems?*

RQ2. *Are inter-language dependencies more risky for multi-language software system in terms of quality?*

RQ3. *Are inter-language dependencies more risky for multi-language software system in terms of security?*

Our results show that:

1) The more inter-language dependencies, the higher the risk of bugs and vulnerabilities, while this risk remains constant for intra-language dependencies.
2) Indirect inter-language dependencies are 2.7 times more common than direct inter-language dependencies.
3) The proportion of bugs introduced in inter-language dependencies is three times higher than in intra-language dependencies, with values ranging between 13,70% and 46,66%.
4) The proportion of security vulnerabilities introduced in inter-language dependencies is two times higher than in intra-language dependencies, where values range between 11,27% and 22,18%.

The contributions of the paper are as follows:

- To the best of our knowledge, we present in this paper the first work that combines two methodologies (static and historic) to study the dependencies in multi-language systems (the case of JNI systems).
- We propose S-MLDA and H-MLDA approaches to detect the (in)direct inter-language dependencies.
- We analyze the impact of the inter-language changes on the software system's quality and security.

II. Motivating example

While existing static code analysis tools such as *ImpactMiner* [7], *Modisco* [11], and *Understand*[1] support multiple programming languages, they can only analyze *one language at a time.* Analyzing the inter-connection between languages is a complicated task as it requires a deeper knowledge of the programming languages involved (including the valid use/call/implementation of dependencies according to the languages' rules) as well as their inter-language calling conventions.

Figure 1 shows an example of JNI source code from Conscrypt. Sub-figure (a) shows the Java class "NativeCrypto", which contains a JNI native method declaration `EVP_PKEY_type(...)`, while Sub-figure (b) shows the C++ file "Native_crypto.cpp", which implements this native function. The Java file that declares the JNI native methods should have relations across C/C++ files [12], which are what we call "direct inter-language dependencies" due to the rules of JNI [13]. Figure 2 shows the corresponding dependency call graph of Conscrypt for Figure 1 using the "Understand" tool. We observe that the JNI dependency between `EVP_PKEY_type(...)` and `NativeCrypto_EVP_PKEY_type(...)` (i.e., the red arrow) is missing.

Many previous works on mono-language systems [6], [14] highlighted the importance of identifying the indirect dependencies that are hidden from the existing static means. In our study, we identify indirect inter-language dependencies by the dependant files that changed together in time but could not be detected by static analysis *i.e.,* in our case study, there is no JNI dependency between these files. Figure 2 shows an example of indirect inter-language dependencies between the files "Conscrypt/Native_crypto.cpp" and "Conscrypt/OpenSSLProvider.java" (*i.e.,* the green arrow). Although these files had no JNI code present (could not be detected statistically), they were changed together frequently. In another example from one of the studied projects in this paper, "Seven-Zip", the ".../sr-

[1] https://scitools.com/

```
public final class NativeCrypto {
  static native int EVP_PKEY_type(...);}
```

(a) JNI native method declaration.

```
static int NativeCrypto_EVP_PKEY_type(JNIEnv*
    env, jclass, jobject pkeyRef) {
  EVP_PKEY* pkey =
      fromContextObject<EVP_PKEY>(env, pkeyRef);
  JNI_TRACE("EVP_PKEY_type(%p)", pkey);
  if (pkey == nullptr) {return -1;}
  int result = EVP_PKEY_type(pkey->type);
  return result;}
```

(b) JNI implementation function.

Figure 1: Example of JNI source code (Conscrypt).

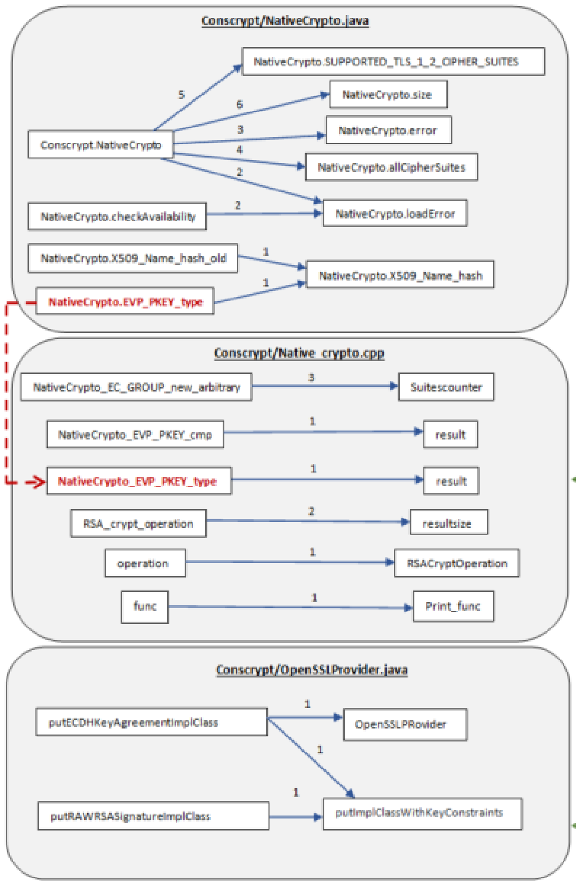

Conscrypt/NativeCrypto.java
NativeCrypto.SUPPORTED_TLS_1_2_CIPHER_SUITES
NativeCrypto.size
NativeCrypto.error
NativeCrypto.allCipherSuites
NativeCrypto.loadError
Conscrypt.NativeCrypto
NativeCrypto.checkAvailability
NativeCrypto.X509_Name_hash_old
NativeCrypto.X509_Name_hash
NativeCrypto.EVP_PKEY_type
5
6
3
4
2
2
1
1
Conscrypt/Native_crypto.cpp
NativeCrypto_EC_GROUP_new_arbitrary
Suitescounter
NativeCrypto_EVP_PKEY_cmp
result
NativeCrypto_EVP_PKEY_type
result
RSA_crypt_operation
resultsize
operation
RSACryptOperation
func
Print_func
3
1
1
2
1
1
Conscrypt/OpenSSLProvider.java
putECDHKeyAgreementImplClass
OpenSSLProvider
putRAWRSASignatureImplClass
putImplClassWithKeyConstraints
1
1
1

Figure 2: Dependency call graph for Figure 1.

c/SevenZipJBinding.cpp" and ".../src/net/sf/ArchiveFormat.java" files changed together in nine commits while they do not contain any static dependency. This kind of dependency could only be detected using historical analysis. We present more details on the identification methodology of the indirect inter-language dependencies in Section III-B2.

Not being able to identify inter-language dependencies increases the complexity of the maintenance activities, as developers may be unaware of the need to change or maintain these dependencies adequately. The goal of this study is to empirically study the inter-language dependencies (direct and indirect) and the extent to which they increase the risk of introducing bugs and/or vulnerabilities.

III. Methodology

This section discusses our methodology to empirically analyze the prevalence of direct and indirect inter-language dependencies and to examine their impact on the quality and the security of software systems.

A dependency is a relationship link between entities inside the same software project. In this study, we consider the following types of dependencies:

- Inter-language dependency (inter-LD): a relationship between files, written in different programming languages, where the dependency identification relies on the (third party) technology used to integrate different programming languages (*e.g.,* Python - C extension). Inter-language dependencies could be direct or indirect.
  - Direct inter-language dependencies (DILD): an inter-language dependency ensuring a static direct call between two files according to the multi-language conventions (*e.g.,* a change in native Java class requires changing the native C/C++ function). We use S-MLDA to identify them.
  - Indirect inter-language dependencies (IILD): an inter-language dependency that is hidden from the static code analysis (*e.g.,* a change in Java class propagated to the native C/C++ function, that in turn impacted foreign C/C++ and/or Java files). We use H-MLDA to identify these dependencies.
- Intra-language dependency (intra-LD): a relationship between files written in the same programming language. There is no specific (third party) technology used to ensure the communication between them.

A. Data Collection

We used the OpenHub API[2] for querying all OpenHub's projects to get the list of the projects that have at least two programming languages, in particular Java and C/C++ (they could have others in addition). We chose OpenHub because it provides the list of the programming languages

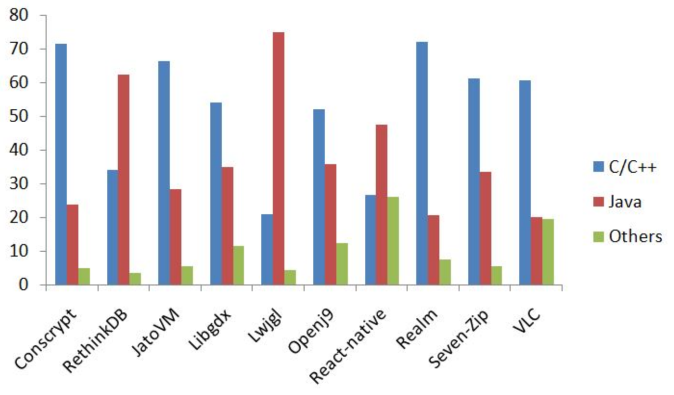


Figure 3: %Programming languages used in each project.

involved in each open source project. From the obtained results, we took the first 100 systems sorted by "Rating" (an option provided by Openhub). We further used the project status to exclude inactive or abandoned systems from the 100 selected projects. We analysed different systems based on their size (number of lines of code). We picked four systems $\leq$ 100kLoC (*i.e.,* Conscrypt, Lwjgl, VLC, and Realm), four systems between 100k and 1M LoC (*i.e.,* Seven-Zip, React-native, Libgdx, and JatoVM), and two systems $\geq$ 1MLoC (*i.e.,* Openj9 and RethinkDB).

Figure 3 shows, for each of the ten selected systems, the programming languages used with their respective proportions. We limit the presentation of our analysis in Figure 3 to three languages per system. The "Others" category combines the rest of programming languages. We present an overview of the collected projects in Table I.

B. Inter-language Dependencies Identification

We use static and historic code analysis to identify the inter-language dependencies. We designed a first approach called S-MLDA based on JNI's rules as defined by Liang [13] to identify the direct inter-language dependencies. We designed a second approach called H-MLDA to identify the indirect inter-language dependencies that are hidden for static code analysis using the changes history.

1) Static Dependency Analysis:

Overview: S-MLDA is a static analyzer written in Java and based on PADL (Pattern and Abstract-level Description Language) [15]. It takes as input a set of multi-language files and statically analyzes their source code using an algorithm based on JNI rules. It provides as output a set of data, i.e., sets of files involving direct inter-language dependencies, presented in a dependency call graph showing the general relationships between files and the specific ones between methods. Figure 4 illustrates the reproduced output of the example presented in Section II.

Motivation: To the best of our knowledge, there is no existing static analysis tool that analyzes the inter-language dependencies for JNI systems and generates a

[2]https://www.openhub.net/

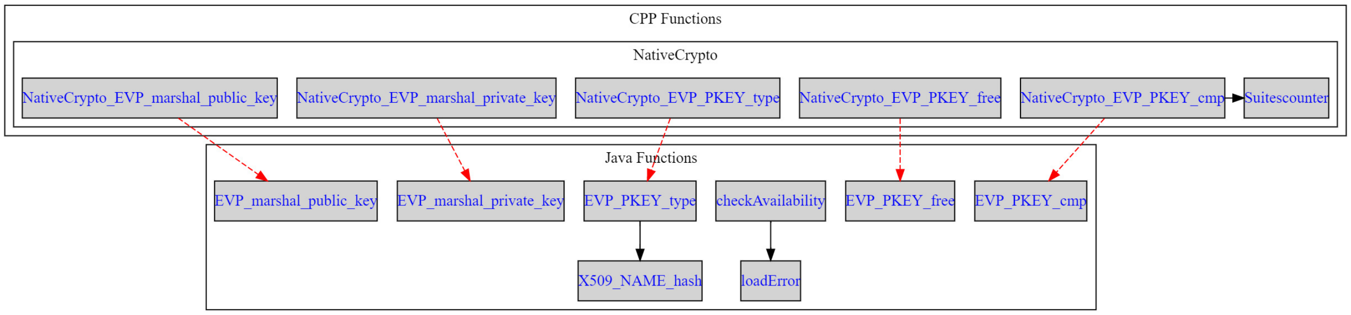


Figure 4: Dependency call graph of a part of Conscrypt generated by *S-MLDA*.

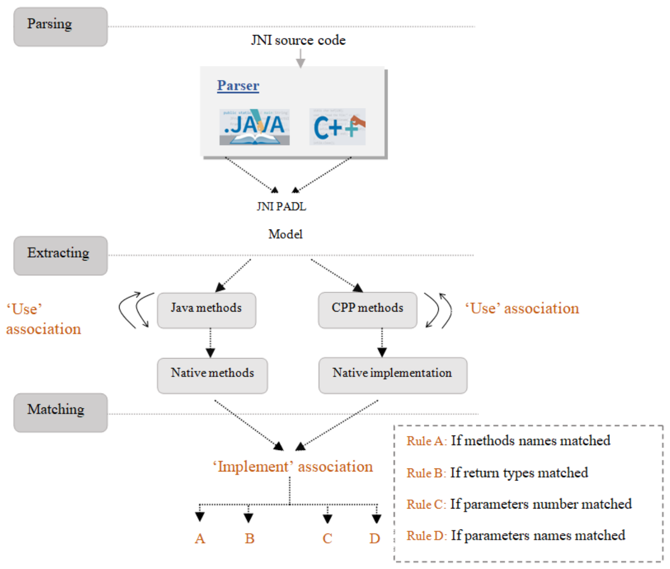


Figure 5: S-MLDA approach.

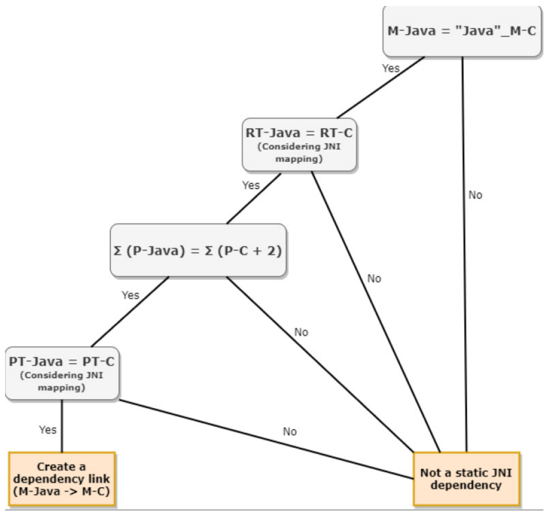

M-Java = "Java"_M-C
Yes
RT-Java = RT-C
(Considering JNI mapping)
No
Yes
Σ (P-Java) = Σ (P-C + 2)
No
Yes
No
PT-Java = PT-C
(Considering JNI mapping)
Yes
No
Create a dependency link (M-Java -> M-C)
Not a static JNI dependency

Figure 6: S-MLDA matching rules.
(M-Java: Java method name; M-C: C function name; RT-Java: Java method return type; RT-C: C function return type; P-Java: Java method parameters; P-C: C function parameters; PT-Java: Java method parameter types; PT-C: C function parameter types)

dependency call graph output [9]. Thus, S-MLDA is the first static analyzer able to report static dependencies between artefacts written in different languages.

Approach: S-MLDA consists of the following steps as shown in Figure 5:

- Parsing: We parse a given JNI system to create a model that contains all constituents of that system *e.g.,* packages, classes, methods, parameters, fields, and relationships (inheritance, implementation, etc.).
- Extracting: We identify the Java methods and C++ functions. From the obtained Java methods, we identified the JNI native methods *i.e.,* methods that contain the keyword "native" in their signatures, and from the obtained C++ functions, we identified the native implementation functions, *i.e.,* functions that implement the native methods.
- Matching: We identify the matched Java native methods with the respective implementations in C++ files based on the JNI rules that we illustrate in Figure 6:
  - Rule A: we verify if the Java method names (M-Java) match with the C++ function names (M-C) using the JNI naming convention.
  - Rule B: we verify the return types (using the JNI mapping types) from the obtained Java (RT-Java) and C++ methods (RT-C) from the previous step to keep only the matching types.
  - Rule C: we verify the number of parameters. We consider the matching when the number of parameters of the C++ function (P-C) equals the number of parameters of the Java method (P-Java) plus two. In JNI, the C++ function implementing the native method contains two more JNI parameters *e.g.,* JNIEnv, jobject.
  - Rule D: Last, we verify the parameter types of the Java method (PT-Java) and the C++ method (PT-C). We consider the methods/functions when we found that the mapping of the parameter type in both methods is matching.

We built, using the obtained relationships, a dependency call graph (example shown in Figure 4) with different hierarchy levels: classes level and file level. In each class, nodes are the methods and the edges are the dependencies.

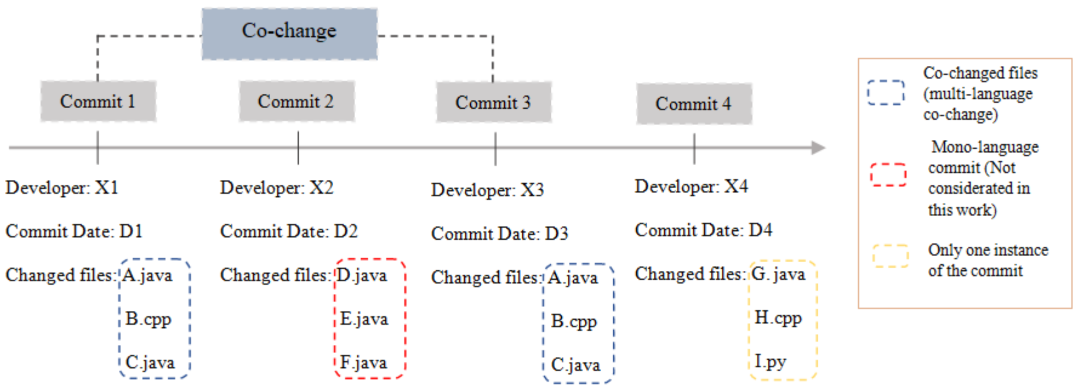


Figure 7: Mono/multi-language co-changes analysis.

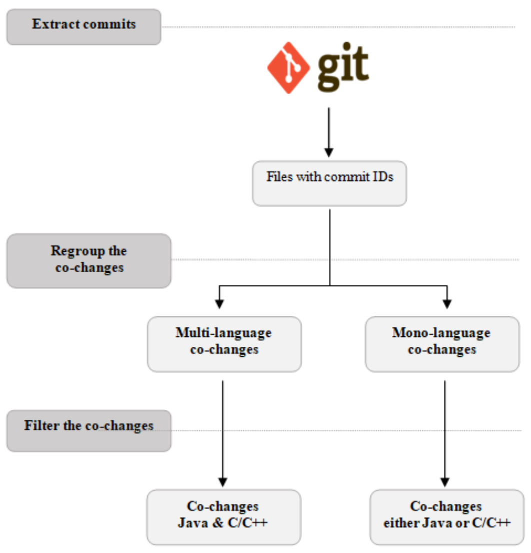


Figure 8: H-MLDA approach.

Edges between nodes across two sub-graphs are inter-language dependencies at methods level. We consider two files having direct inter-language dependencies if they involve at least one edge across the two sub-graphs. From the obtained sets, we remove the redundant dependent files.

2) Historical Dependency Analysis:
Overview: H-MLDA studies the change history of a multi-language project's Git repository to reveal the indirect inter-language dependencies. It identifies the multi-language co-changes (involving multi-language files), converts them into sets of multi-language files, and removes the sets in common with the output of S-MLDA (example is shown in Figure 9). These steps allow retrieving the indirect inter-language dependencies not detected by S-MLDA (*i.e.,* potentially hidden for static analysis).

Motivation: Historical code analysis is one of the common methodologies followed to analyze dependencies [6], [16], [17]. In particular, the concept of co-change is a useful method to recommend dependent files potentially relevant to a future change request when they are detected by static code analysis. The purpose in this analysis is to identify a set of files changing together over time often enough (within commits) to derive an assumption that these files could be historically dependent.

Approach: H-MLDA consider a co-change to be a commit involving source code files that have been observed to change together [18] (set of files changing together) to exhibit some form of logical coupling, *i.e.,* a temporal relationship among files changed over time [14]. Let's consider Figure 7, which shows four different co-changes in a given multi-language system. Commit 1 and 3 are instances of the same changed three files *i.e.,* a co-change, involving the same three files. Commit 2 is a mono-language co-change *i.e.,* involving mono-language files. Commit 4 presents the case of a multi-language files with only one instance *i.e.,* a set of files changed together only once. To reduce the number of false positives, we did not consider the cases where files are co-changed accidentally without a significant reason *i.e.,* files appeared changing together (at commit level) only one time.

Figure 8 presents a summary of the main parts of H-MLDA. We query Git repository for extracting all the commits and analyze the data obtained. We split the co-changed set of files in two groups: one for the inter-language co-changes involving Java and C/C++ files and the second one for the intra-language co-changes involving Java or C/C++ files.

To allow a logical comparison at files level between S-MLDA and H-MLDA outputs (since, by default, H-MLDA concerns commits, while S-MLDA concerns files) and to identify the indirect inter-language dependencies, we convert the multi-language co-changes into sets of multi-language files as shown in Figure 9 in step (1). Then, we remove the ones in common with S-MLDA sets *i.e.,* the direct inter-language dependencies as presented in step (2) in Figure 9, and last, we consider the remaining sets the indirect inter-language dependencies. This is illustrated in step (3) in the same Figure 9.

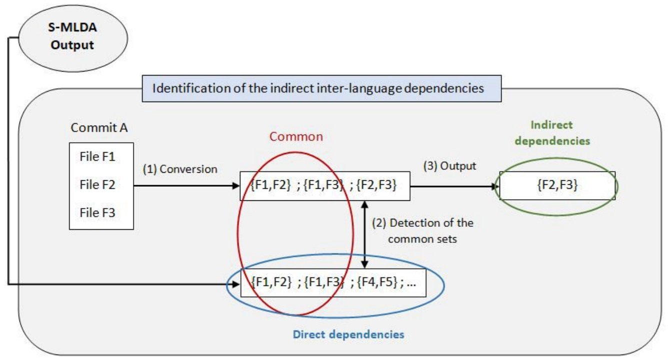

S-MLDA Output
Identification of the indirect inter-language dependencies
Commit A
File F1
File F2
File F3
(1) Conversion
Common
{F1,F2} ; {F1,F3} ; {F2,F3}
(3) Output
Indirect dependencies
{F2,F3}
(2) Detection of the common sets
{F1,F2} ; {F1,F3} ; {F4,F5} ; ...
Direct dependencies

Figure 9: Identification of indirect dependencies.

C. *Quality Issues And Security Vulnerabilities*

We aim to understand if the inter-language dependencies in multi-language systems introduce more bugs and/or security vulnerabilities than intra-language dependencies. We also study the relation between DILD and IILD with the project quality (introduction of bugs) and the project security (introduction of security vulnerabilities). To achieve this, we collected the bug reports of the studied systems (during the collection step of the systems, we ensured that their respective bug reports as well as their commit messages are accessible).

We take advantage of the existing SZZ algorithm [18] to identify bug-introducing dependencies. The SZZ algorithm identifies changes that are likely to introduce issues, it uses the issue report information to find such bug-introducing changes. Using the results of the SZZ algorithm, we prepared an issue data-set that contains essential information about the bug such as its bug ID, files affected, when it was reported and fixed.

Regarding security issues, we analyzed five vulnerabilities: memory faults, null-pointer exceptions, initialization checkers, race conditions, and access control problems [19]. We implemented a script where we used these vulnerabilities as search keywords (in the collected bug reports and commit messages) in addition to the following keywords: threat(s), vulnerability(ies), and security. It should be noted that, to increase the confidence in our results, none of the bug reports identified by the vulnerability search are present in the aforementioned issue data-set (the two data-sets are completely disjoint.

We put all our data and scripts in the following online website[3] for replication.

IV. Results

The following section presents our results and summarizes them per research question.

[3]https://drive.google.com/drive/folders/1bRXdPgpicEkP6CxxUY5DH96D4WxAb8hM

Table I: #Mono-/Multi-language Commits.

	Systems	#Total Commits	#Multi-language Commits	#Mono-language Commits
1	Conscrypt	3860	2952(76,47%)	908(23,53%)
2	RethinkDB	19898	13,002(65,34%)	6896(34,66%)
3	JatoVM	4135	2996(72,45%)	1139(27,55%)
4	Libgdx	13580	11,332(83,40%)	2248(16,60%)
5	Lwjgl	3884	2108(54,27%)	1776(45,73%)
6	Openj9	6219	4992(80,27%)	1227(19,73%)
7	React-native	16298	6772(41,55%)	9526(58,45%)
8	Realm	8172	5893(72,11%)	2279(27,89%)
9	Seven-Zip	909	432(47,52%)	477(52,48%)
10	VLC	11866	8676(73,11%)	3190(26,89%)

Table II: Proportions of identified inter-LD and intra-LD.

	Systems	%Inter-LD (out of multi-language commits)	%Intra-LD (out of multi-language commits)
1	Conscrypt	70,59	29,41
2	RethinkDB	73,33	26,67
3	JatoVM	83,34	16,66
4	Libgdx	68,74	31,26
5	Lwjgl	51,61	48,38
6	Openj9	71,99	28,01
7	React-native	55,55	44,45
8	Realm	62,07	37,93
9	Seven-Zip	63,63	36,36
10	VLC	48,27	51,73

RQ1. *How common are direct and indirect inter-language dependencies in multi-language systems?*

Motivation: The goal of this research question is to measure the prevalence of inter-language dependencies. For that, we need to identify the dependent multi-language files (inter-language dependencies) in order to reveal the direct and indirect ones using both historic and static analysis. We aim to compare the prevalence of static direct dependencies to indirect dependencies (potentially hidden for the static analysis).

Approach: We present in Table I the results related to the commits identified by H-MLDA. We show the total number of the multi-language commits and the total number of the mono-language commits, with their respective percentage out of all the commits. The results of the inter-LD and intra-LD are illustrated in Table II, where the two columns show the percentage of inter-LD and intra-LD identified out of the total number of multi-language commits. Table III shows (i) the number of DILD (identified by S-MLDA) and (ii) the number of IILD (generated via H-MLDA). Lastly, we evaluate the performance of S-MLDA and H-MLDA in order to validate their precision and recall.

Results: We observed that **multi-language co-changes are common in multi-language systems, with values ranging between 47,52% and 83,40% (relative to the total number of commits)**. Developers are changing files written in diverse languages at the same time, which indicates a strong logical coupling between these multi-language files. Multi-language commits involve more than 50% of inter-LD in 90% of the systems (except the case of VLC where the %inter-LD is 48,27%). The values range between 48,27% and 83,34%.

The results from Table III show that **the number of indirect inter-language dependencies is higher (average of 2.7 times) than the number of direct inter-language dependencies in 90% of the cases (nine systems), while it is nearly equal for the case of Libgdx.** The high number of indirect inter-language dependencies can be precarious for system maintenance activities; since these dependencies are hidden from static code analysis tools, any change to them can negatively impact the system.

During the IILD identification (*i.e.,* generation of sets of files from co-changes and removal of the common sets with DILD), **we found that all of the DILD (*i.e.,* the sets of files Java and C/C++) were included *i.e.,* a part of in the multi-language co-changes** (SMLDA $\subset$ HMLDA).

Discussion: We discuss the accuracy of the results of H-MLDA and S-MLDA by evaluating the precision and the recall.

We manually evaluate the precision by randomly selecting a sample of data for each approach (using the sampling methodology in [20]). To select the sample, we set a confidence level of 95% and an error margin of 5%. Our final samples contained 379 DILD for S-MLDA and 382 IILD for H-MLDA.

Case of S-MLDA: We manually checked the source code of the direct inter-language dependencies sets for the existence of one or more of the following JNI elements as they identify the existence of JNI source code [13]: JNI header (*i.e.,* #include ⟨jni.h⟩); JNI pointer (*i.e.,* JNIEnv); JNI keyword (*i.e.,* native(; and JNI functions (*i.e.,* FindClass(), GetMethodID(), etc.).

Case of H-MLDA: We manually reviewed the source code to verify that no JNI dependencies were present in the sample of 382 IILD. Then, we validated the file dependencies based on one of the following elements:

- Similarity of the files names. We checked if the IILD files in these sets could have a same or similar names *e.g.,* a sub-string of a file name A included in file B.
- The intent of the source code files. We reviewed the source code and the comments inside for each set *i.e.,* Java and C(++), to find if there is a behavior between the files that could explain the indirect dependency.
- Existence of external information sources. We searched in the bug reports and developers discussions if the files indirectly dependent were involved in the same issue or were reported as related.

For the recall, we considered all the sets (DILD and IILD) presented in Table III.

Case of S-MLDA: We implemented a script to count occurrences of the JNI header presented in all the source code and compared it with the number of JNI headers found in C(++) files involved in the IILD sets.

Table III: #(In)Direct Inter-language Dependencies.

	Systems	#DILD (S-MLDA)	#IILD (H-MLDA)
1	Conscrypt	1341	2827
2	RethinkDB	4321	8299
3	JatoVM	1128	3154
4	Libgdx	6232	5803
5	Lwjgl	733	3149
6	Openj9	4438	7364
7	React-native	2116	6416
8	Realm	2266	6177
9	Seven-Zip	174	513
10	VLC	3172	9004

Table IV: %Bugs and vulnerabilities within co-changes.

Systems	%Buggy Inter-LD	%Buggy Intra-LD	%Vulnerable Inter-LD	%Vulnerable Intra-LD
Conscrypt	33,33	12,03	21,66	5,02
RethinkDB	40,90	10,72	22,18	3,76
JatoVM	46,66	11,42	0	9,44
Libgdx	18,18	10,33	19,27	8,33
Lwjgl	12,5	13,7	15,5	4,31
Openj9	38,88	12,66	21,11	7,83
React-native	13,33	9,77	16,66	3,52
Realm	16,66	11,82	0	0
Seven-Zip	16,66	9,78	18,57	11,27
VLC	7,14	12,87	11,45	0

Case of H-MLDA: We implemented a script to identify all the inter-language dependencies sets that have similar names and that are not JNI. From the sets found, we excluded the sets successfully detected by H-MLDA. The remaining ones are the inter-language dependencies not detected by H-MLDA which are considered in calculating the recall.

The final results show a precision of 100% and a recall of 78% for S-MLDA, and a precision of 68% and a recall of 87% for H-MLDA.

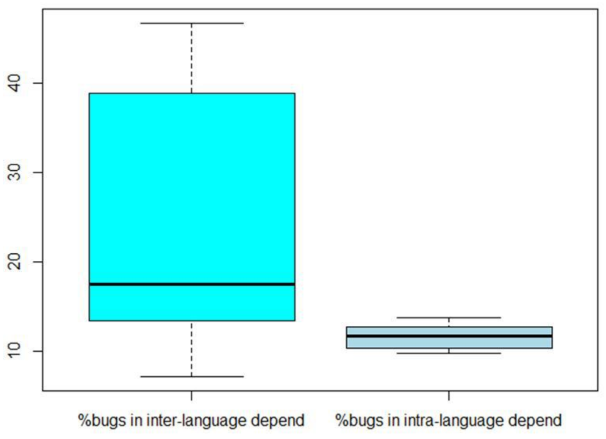


Figure 10: %Buggy dependencies.

RQ2. *Are inter-language dependencies more risky for multi-language software system in terms of quality?*

Motivation: Dealing with dependencies in multi-language systems is a challenging task as different programming languages are involved, in which tracking the dependencies requires specific effort. Through this research question, we aim to understand if bug introduction is frequent in these kinds of dependencies (a case study of JNI) and we aim to identify its consequence on the system quality.

Approach: Considering the result of RQ1 (*i.e.,* S-MLDA is a subset of H-MLDA), in the following, we studied the interaction between the (intra)inter-language dependencies and the quality issues.

We show in Table IV the percentage (out of %inter-LD and %intra-LD) of the buggy inter-/intra-language dependencies.

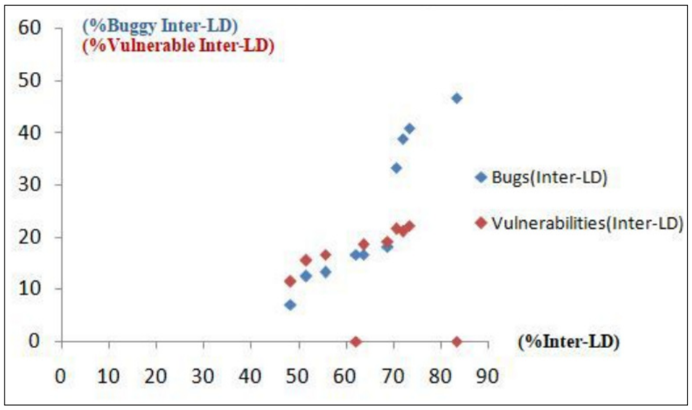
Results: The percentage of bug-introducing co-changes was as high as 46,66% in inter-language dependencies and 13,7% in the case of intra-language dependencies. We report that when the number of inter-language dependencies increased, the number of bug-introducing commits increased with a significant correlation of 0,918 and vice-versa. Conversely, we observe that bug-introducing co-changes are constant for intra-language dependencies, with values range between 9,77 and 13,70. Hence, there is no significant correlation between bugs and intra-language dependencies.

The box-plot in Figure 10 shows the difference between the median of each set *i.e.,* bugs in inter-LD and bugs in intra-LD. Moreover, the scatter-plot in Figure 11 allows a better analysis to answer the research question with the following: **The more the X axis in Figure 11a increases for inter-language dependencies, the higher the risk of bugs (blue color) being introduced, while this risk remains constant for intra-language dependencies (Figure 11b).**

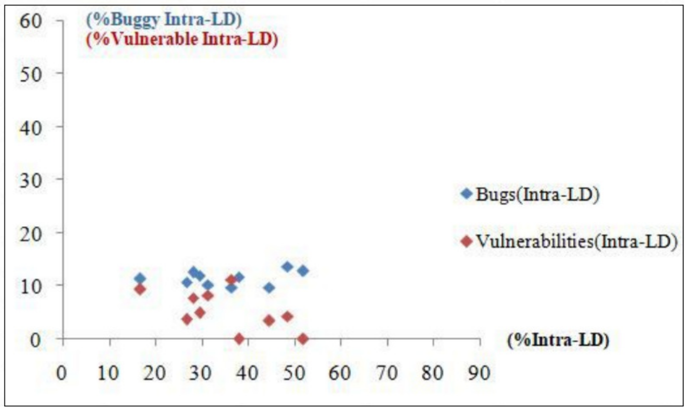
We used the Mann-Whitney U test [21] with a 95% confidence level (*i.e.,* $\alpha = 0.05$) to determine if there is a significant difference between inter-LD and intra-LD in terms of bugs. The test shows a significant difference **(p-value = 0.017)** between the percentage of bugs presented in the inter-LD and the percentage of bugs presented in the intra-LD.

The results in Figure 11b show no correlation between bugs and intra-LD. Thus, we focus our next analysis on buggy (in)direct inter-LD. We illustrate the obtained results in a scatter-plot presented in Figure 12a. We can observe that **the more the X axis increases for inter-LD, the higher the risk of quality bugs introduced in both indirect inter-LD(IILD) and direct inter-LD(DILD)**.

Discussion: The percentage of quality issues in inter-language dependencies (46,66%) is higher nearly to three times compared with quality issues in co-changes involving intra-language files (13,7%). Several previous works suggested that combining programming languages presents always a challenging activity as it increases the complexity of the software and leads to hard maintenance [1]. Thus, analyzing the impact of a change through multi-language files is important to avoid software issues. In many cases, indirect dependencies could be risky to the quality of the system as these kinds of dependencies are hard to identify via static analysis. Our results show that the risk of introducing bugs are 1.5 times higher in indirect inter-LD (not detectable by static analysis) than direct ones.



(a) Inter-language dependencies

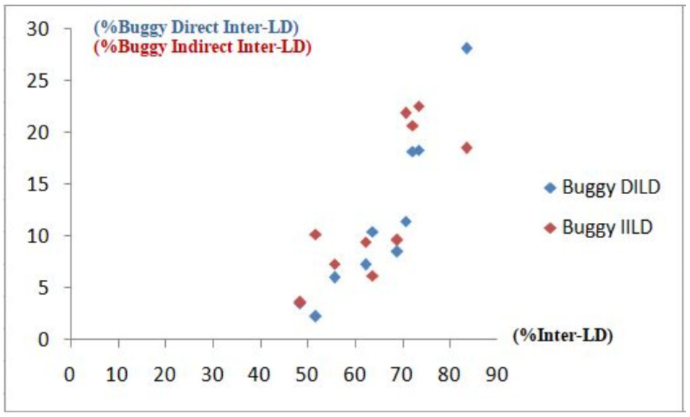


(b) Intra-language dependencies

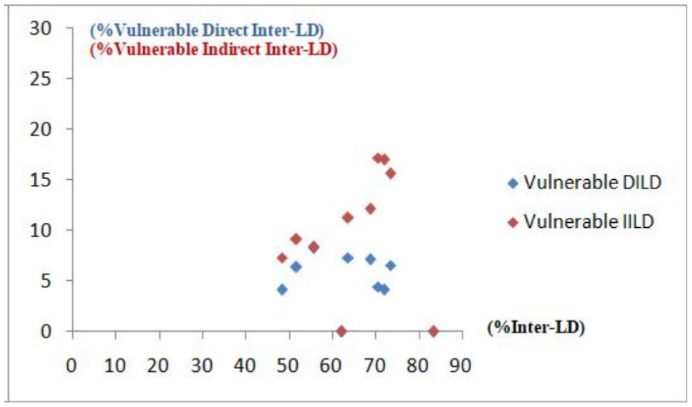
Figure 11: %Quality and security issues detected in intra- and inter-language dependencies.

The following are two examples of bugs introduced within inter-language dependencies.

- The first example was extracted from *Conscrypt*. It presents a co-change that involved four files written in Java and C++. It was responsible for the introduction of a bug because of a miss of a change in the native function NativeCrypto.EVP_DigestVerifyFinal() implemented in org_conscrypt_NativeCrypto.cpp. The author of the change modified the signature of the native Java method EVP_DigestVerifyFinal() and missed the corresponding modification in the CPP file that results from the inter-language dependency. We explain it by the fact that the author of the change was not able to identify the JNI dependencies and to track the change propagation *i.e.,* a miss of the dependency analysis.
- The second example concerns a co-change extracted from *Realm*. It involved inter-language dependencies



(a) Buggy DILD and IILD



(b) Vulnerable DILD and IILD

Figure 12: %Quality and security issues detected in (in)direct inter-language dependencies.

with a total of six Java files and three CPP files. Our analysis shows that this change introduced a bug when the new return type of the Java native method in Realm.java did not match with the old return type of the corresponding implementation of CPP file. These kinds of changes are subject to bugs especially when several changes are involved (*i.e.,* on nine files that were indirectly dependent). JNI practices [12] should be well mastered by the developers as they present another way to protect the source code from quality decrease.

RQ3. *Are inter-language dependencies more risky for multi-language software system in terms of security?*

Motivation: A major concern of practitioners and researchers nowadays is security vulnerabilities especially since the emblematic "Heartbleed bug", which is a security flaw that exposed millions of passwords and personal information. As today almost the software systems are multi-language systems, security vulnerabilities in those systems became a priority for developers [10]. This research question aims to identify the security impact of the inter-language dependencies in multi-language systems.

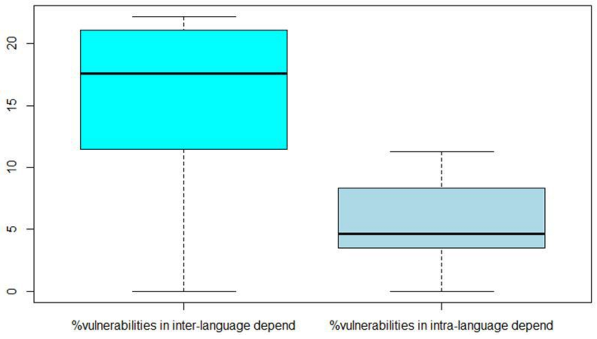


Figure 13: %Vulnerable dependencies.

For this, we present the relationship between the inter-/intra-language dependencies with vulnerabilities.

Approach: Security software vulnerabilities are weaknesses in software systems that can be exploited by a threat actor, such as an attacker, to perform unauthorized actions within a computer system. Software vulnerabilities can be defined as incorrect internal states detected in the software source code that could allow an attacker to compromise its integrity, availability, or confidentiality. Most software security vulnerabilities fall into one of a small set of categories:

- Memory faults: such as buffer overflows and other memory corruptions which impact the security of a software system.
- Null-pointer exceptions: which can threat the system confidentiality when it is used to reveal debugging information.
- Initialization checkers: which can threat the system integrity when the software components are used without being properly initialized.
- Race conditions: related to weak time checking between software tasks and can allow an attacker to obtain unauthorized privileges.
- Access control problems: related to weak specifications of privileges defining the access or the modification of software files .

We show in Table IV the percentage (out of inter-LD and intra-LD) of the dependencies involving vulnerabilities.

Results: We observe that 80% of the studied systems revealed security issues introduced within inter- and intra-language dependencies. **The percentage of vulnerabilities in inter-language dependencies can reach up to nearly 22,18%, and 11,27% in intra-language dependencies.** We present the findings in a scatter-plot, Figure 11, for a better visual analysis. We observe that **the more we have inter-language dependencies, the higher the risk of vulnerabilities being introduced**. Without considering JatoVM and Realm (where vulnerabilities could not be found), a correlation of 0,961 was found between inter-language dependencies and security vulnerabilities.

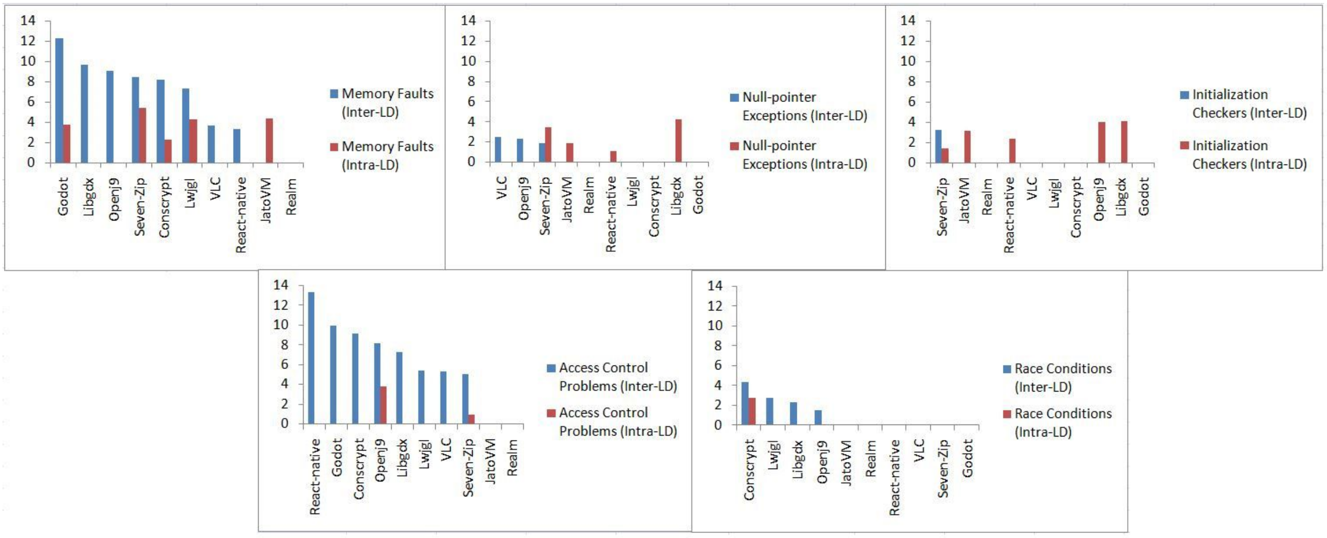


Figure 14: Distribution of the five security vulnerabilities categories.

The box-plot presented in Figure 13 and the p-value of the Mann-Whitney U test **(p-value = 0.007)** shows a significant difference between the percentage of vulnerabilities in inter-LD and the percentage of vulnerabilities in intra-LD. Regarding the difference between the risks of vulnerabilities in direct inter-LD and indirect inter-LD, we report from Figure 12b that **the more we have inter-LD the higher is the risk of vulnerabilities introduced in indirect inter-LD comparing with direct intra-LD where it remains nearly the same.**

Discussion: From the results, we observe that vulnerabilities in inter-language dependencies (22,18%) are twice as common as in intra-language dependencies (11,27%). This leads to the conclusion that inter-language dependencies are more risky than intra-language dependencies and developers should consider the challenge provided with multi-language systems.

We observe from Table IV that no vulnerabilities were found in inter-language dependencies of Realm and JatoVM. Realm is written mostly in Java (82.3%) where C++ presents 8%. Java does memory management automatically, the compiler catches more compile-time errors, and it does not allocate direct pointers to memory. We observed that the selected software written mostly in Java is less vulnerable than C or C++ to memory security vulnerabilities. Indeed, a similar observation was reported previously by other researchers [22]. The results of Lwjgl are similar to Realm. Lwjgl is mostly written in Java (85,1%) but the difference is that the second language used is C and not C++ (Object-oriented programming (OOP)). We are exploring in an ongoing work if the fact of having dependencies among files written in Java and procedural programming language instead of an OOP language may increase the existence of vulnerabilities in multi-language systems.

JatoVM is the implementation of the Java virtual machine. Vulnerabilities have not been found within inter-language dependencies, however, they were detected within intra-language dependencies. JatoVM is written mostly in the C language (73%). C is a low-level programming language that provides access to low-level IT infrastructure. We noticed that previous researchers reported that manipulating C language is critical in the software security context [23]. Conduction further empirical studies can better explain the fact that we are founding several security vulnerabilities propagated between C files. In future work, we will study the reasons behind not finding vulnerabilities within the inter-language dependencies to investigate whether if this low vulnerabilities is related to the architecture of the system *i.e.,* how it is designed or if it is related to the domain, as it is a java virtual machine.

Figure 14 shows the distribution of the percentage of the vulnerabilities for each category *i.e.,* Memory faults, Null-pointer exceptions, Initialization checkers, Access control problems, Race conditions. We can observe from the barplot that **the most vulnerabilities in inter-LD are the Memory faults and the Access control (presented in 80% of the systems) followed by Race conditions (presented in 40% of the systems) while the rest are shown in less than 30% of the systems**. However, for intra-LD, Memory faults and Initialization checkers are presented in 50% of the systems while Null-pointer exceptions were found in 40% of the systems and the rest are under of 30%. Abidi *et al.* through their study [24], support this finding as Memory faults *e.g.,* buffer_overflow are the most known vulnerability subject of security issues in multi-language systems. The pool of practitioners who participated in that survey explained this fact by claiming that these programming languages do not provide security protection against overwriting data in memory and do not automatically check that data written to an array is within the boundaries of that array. Moreover, Tan *et al.* [25] discussed the importance of caring about violating access control rules in JNI systems as native methods can access and modify any memory location in the heap.

V. Threats to validity

We now discuss threats to the validity of our study [26].

Threats to internal validity: We relied on the literature to extract the JNI rules and to apply the co-changes method. We evaluate the accuracy of the approaches used through the study where we found precision (recall) values of 100% (68%) for S-MLDA and 68% (87%) for H-MLDA.

Threats to construct validity: The process followed in collecting the data may introduce some inaccuracies. The use of PADL meta-model, co-changes method, SZZ algorithm, and vulnerability classification may present a threat to construct the study and it may exist other means. However, we mentioned that many previous works relied on these means and validated them. For this reason, the precision and the recall of these means is a concern that we agree to accept.

Threats to external validity: Our results may not be representative of all multi-language projects, since we only studied the case of JNI on ten open-source projects. We accept this threat as software system' characteristics could vary depending on different criteria. However, to mitigate this threat, we varied the selected projects based on project status, size, and languages used.

Threats to reliability validity: We provided a companion website[3] with all the needed data and results to replicate this study. Meta-models and tools used in our study are open source and free to access.

VI. Related Work

Cossette and Walker [3] argued that dependency analysis is very important in determining the change-impact in software. They identified and reported the limitations of existing techniques for supporting multi-language dependencies. They applied island grammars to detect dependencies in multi-language software systems.

Nguyen *et al.* [27] studied multi-language dependencies in web applications where they focused on analysing dynamically the dependencies using slicing techniques. They introduced an WebSlice to compute program slicing across PHP, SQL, HTML, and JavaScript. Our study propose a static and historic analysis of dependencies using respectively dependency call graphs and co-changes techniques.

Deruelle *et al.* [28] proposed a model for change propagation applied to heterogeneous databases applications. Their model is based on graphs rewriting and deals with centralized and distributed environments. They used a CORBA-based framework because it contains three different databases and environments. They treats the multiplicity in only heterogeneous databases and does not consider the multiplicity in languages.

Shatnawi *et al.* [5] analyzed the challenges that a multi-language system could have and that make the static code analysis a hard task for the developer. They proposed a solution based on KDM (Knowledge Discovery Meta-model) where they identified dependencies between different artifacts. Their work was limited to the container services where they studied the case of server-side Java with client-side Web dialects (JSP, JSF, etc.).

Sayagh *et al.* [29] highlighted the challenge of identifying configuration options through a multi-layer software. They were the first researchers to perform an empirical study toward identifying the configuration dependencies through multiple layers as configuration options in each layer might contradict each other. One of the main finding was that there is more indirect use of configuration options than direct use where they concluded that the detection and fixing of configuration errors could become more difficult. We follow a part of their methodology to identify the indirect inter-language dependencies in multi-language systems.

Jaafar *et al.* [6] presented a novel approach called *Macocha* to validate two change patterns from analyzing the co-changes: the asynchrony change pattern, corresponding to macro co-changes (MC), that is, of files that co-change within a large time interval (change periods), and the dephased change pattern, corresponding to dephased macro co-changes (DC), that is, MC that always happens with the same shifts in time. They applied *Macocha* on seven diverse systems. The authors considered only mono-language files, either Java or C/C++ source code.

Zimmerman *et al.* [8] and Ying *et al.* [9] presented an approach based on association rules to identify co-changing files. They argued that co-changes could be useful to recommend dependent entities potentially relevant for future change. They used the co-change history in CVS to extract co-changing files. The authors did not consider the dependency aspects of the identified co-changes. Moreover, their algorithm could only be applied to one language at a time *e.g.,* not a multi-language concept.

VII. Conclusion and Future Work

Multi-language systems present challenges in terms of code analysis and code maintenance. Developers are required to have knowledge in every language used in the software to identify all the dependencies. This paper aims to analyse inter-language dependencies and their relation with the software quality and security. We empirically applied two approaches based on historical dependency analysis (H-MLDA) and static dependency analysis (S-MLDA) on ten open-source multi-language systems (i) to identify the indirect (IILD) and direct (DILD) inter-language dependencies and to (ii) to study their impact on the quality and the security of multi-language systems. We evaluated the accuracy of the results and we found precision (recall) values of 100% (68%) for S-MLDA, and 68% (87%) for H-MLDA.

Our main results showed that IILD are 2.7 times more common than DILD. The more inter-LD, the higher the risk of bugs and vulnerabilities, while this risk remains constant for intra-LD. Bugs introduced in inter-LD are three times higher than in intra-LD. Security vulnerabilities introduced in inter-LD are two times higher than in intra-LD. We recommend to the multi-language developers to first use specific approaches (*i.e.,* S-MLDA) during their multi-language changes to identify the inter-LD especially the (hidden) indirect ones. Secondly, they are recommended to take care of the quality aspect while changing a multi-language software *i.e.,* specific approaches are needed to help spot quality issues during the change tasks. Last, we recommend to emphasize more the control on the memory fault and access control problem when dealing with multi-language systems.

In future work, we plan to (1) generalize this study by analyzing more combinations of programming languages; (2) investigate more vulnerabilities and their relationship with the inter-language dependencies; and (3) conduct studies with developers to understand their perspective on multi-language developments.

Acknowledgments

This work has been partly funded by an NSERC Discovery grant, the Computer Research Institute of Montreal, and Polytechnique Montreal. We thank Yann-Gaël Guéhéneuc for his valuable inputs.

References

[1] F. Boughanmi, "Multi-language and heterogeneously-licensed software analysis," in *17th Working Conference on Reverse Engineering*, 2010.

[2] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change.* Academic Press Professional, Inc., 1985.

[3] B. Cossette and R. J. Walker, "Polylingual dependency analysis using island grammars: A cost versus accuracy evaluation," in *IEEE International Conference on Software Maintenance*, 2007.

[4] S. Hassaine, F. Boughanmi, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "A seismology-inspired approach to study change propagation," in *27th IEEE International Conference on Software Maintenance*, 2011.

[5] A. Shatnawi, H. Mili, M. Abdellatif, Y.-G. Guéhéneuc, N. Moha, G. Hecht, G. E. Boussaidi, and J. Privat, "Static code analysis of multilanguage software systems," *arXiv preprint arXiv:1906.00815*, 2019.

[6] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "Detecting asynchrony and dephase change patterns by mining software repositories," *Software: Evolution and Process*, 2014.

[7] B. Dit, M. Wagner, S. Wen, W. Wang, M. Linares-Vásquez, D. Poshyvanyk, and H. Kagdi, "Impactminer: A tool for change impact analysis," in *Companion Proceedings of the 36th International Conference on Software Engineering.* ACM, 2014.

[8] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering.* IEEE Computer Society, 2004.

[9] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, Sep. 2004.

[10] M. Abidi, M. Grichi, F. Khomh, and Y.-G. Guéhéneuc, "Code smells for multi-language systems," in *Proceedings of the 24th European Conference on Pattern Languages of Programs*, ser. EuroPLop '19. New York, NY, USA: Association for Computing Machinery, 2019.

[11] H. Brunelière, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, 2014.

[12] M. Grichi, M. Abidi, Y.-G. Guéhéneuc, and F. Khomh, "State of practices of java native interface," in *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '19. IBM Corp., 2019.

[13] S. Liang, *Java Native Interface: Programmer's Guide and Reference.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[14] F. Jaafar, Y. Gueheneuc, S. Hamel, and G. Antoniol, "An exploratory study of macro co-changes," in *2011 18th Working Conference on Reverse Engineering*, Oct 2011, pp. 325–334.

[15] Y. Guéhéneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, 2008.

[16] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 241–250.

[17] N. Ali, F. Jaafar, and A. E. Hassan, "Leveraging historical co-change information for requirements traceability," in *20th Working Conference on Reverse Engineering (WCRE)*, 2013.

[18] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, 2005.

[19] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities," in *Conference on Availability, Reliability and Security*, 2009.

[20] R. L. Scheaffer, W. Mendenhall III, R. L. Ott, and K. G. Gerow, *Elementary survey sampling.* Cengage Learning, 2011.

[21] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods.* John Wiley & Sons, 2013, vol. 751.

[22] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang, "Safe Java Native Interface," in *In Proceedings of the 2006 IEEE International Symposium on Secure Software Engineering*, 2006.

[23] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proceedings of the 17th Conference on Security Symposium.* USA: USENIX Association, 2008.

[24] M. Abidi, M. Grichi, and F. Khomh, "Behind the scenes: Developers' perception of multi-language practices," ser. CASCON '19, 2019, p. 72–81.

[25] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang, "Safe java native interface," in *Proceedings of IEEE International Symposium on Secure Software Engineering*, vol. 97, 2006, p. 106.

[26] R. K. Yin, *Applications of case study research.* sage, 2011.

[27] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Cross-language program slicing for dynamic web applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 369–380.

[28] L. Deruelle, M. Bouneffa, N. Melab, and H. Basson, "A change propagation model and platform for multi-database applications," in *Proceedings IEEE International Conference on Software Maintenance*, 2001.

[29] M. Sayagh and B. Adams, "Multi-layer software configuration: Empirical study on wordpress," in *15th International Working Conference on Source Code Analysis and Manipulation*, 2015.