On the Impact of Interlanguage Dependencies in Multilanguage Systems Empirical Case Study on Java Native Interface Applications (JNI)

Manel Grichi, *Graduate Student Member, IEEE*, Mouna Abidi, Fehmi Jaafar, Ellis E. Eghan, and Bram Adams, *Member, IEEE*

***Abstract*—Nowadays, developers are often using multiple programming languages to exploit the advantages of each language and to reuse code. However, dependency analysis across multilanguage is more challenging compared to mono-language systems. In this article, we introduce two approaches for multilanguage dependency analysis: static multilanguage dependency analyzer) and historical multilanguage dependency analyzer, which we apply on ten open-source multilanguage systems to empirically analyze the prevalence of the dependencies across languages, i.e., interlanguage dependencies and their impact on software quality and security. Our main results show that: the more interlanguage dependencies, the higher the risk of bugs and vulnerabilities being introduced, while this risk remains constant for intralanguage dependencies; the percentage of bugs within interlanguage dependencies is three times higher than the percentage of bugs identified in intralanguage dependencies; the percentage of vulnerabilities within interlanguage dependencies is twice the percentage of vulnerabilities introduced in intralanguage dependencies.**

***Index Terms*—Change history, cochanges, dependency analysis, multilanguage, static code analysis.**

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 multilanguage 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]–[8] using a variety of techniques such as static code analysis and mining software repositories, dependency analysis for multilanguage systems is not well established yet [5] and is still subject to further research [9].

In this article, we empirically study ten Java Native Interface (JNI) open-source multilanguage systems to identify the interlanguage dependencies, analyze their prevalence in multilanguage 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] showed that Java-C/C++ is the most common combination of languages used in multilanguage systems.

To identify the interlanguage dependencies, we introduce two approaches, which we applied on the ten JNI systems: static multilanguage dependency analyzer (S-MLDA) that performs a static dependency analysis using heuristics and naming conventions to detect direct interlanguage dependencies (a direct static relationship between two multilanguage files), and historical multilanguage dependency analyzer (H-MLDA) that performs historical dependency analysis based on software cochanges to identify the indirect interlanguage 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 multilanguage systems?*

RQ2: *Are interlanguage dependencies more risky for multilanguage software system in terms of quality?*

RQ3: *Are interlanguage dependencies more risky for multilanguage software system in terms of security?*

Manuscript received August 1, 2020; accepted September 12, 2020. Date of publication November 2, 2020; date of current version March 2, 2021. This work was supported in part by the NSERC Discovery grant, in part by the Computer Research Institute of Montreal, and in part by Polytechnique Montreal. Associate Editor: W. K. Chan. *(Corresponding author: Manel Grichi.)*

Manel Grichi, Mouna Abidi, Ellis E. Eghan, and Bram Adams are with the Department of Computer and Software Engineering, Polytechnique Montreal, Montreal QC H3T 1J4, Canada (e-mail: manel.grichi@polymtl.ca; mouna.abidi@polymtl.ca; ellis.eghan@polymtl.ca; bram.adams@polymtl.ca).

Fehmi Jaafar is with the Computer Research Institute of Montreal, Montreal QC H3N 1M3, Canada (e-mail: fehmi.jaafar@crim.ca).

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2020.3024873

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

(a)

```
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)

Fig. 1. Example of JNI source code (Conscrypt). (a) JNI native method declaration. (b) JNI implementation function.

Our results show that the following condition hold.

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

The contributions of this article are as follows.

1) To the best of our knowledge, we present in this article the first work that combines two methodologies (static and historic) to study the dependencies in multilanguage systems (the case of JNI systems).
2) We propose S-MLDA and H-MLDA approaches to detect the (in)direct interlanguage dependencies.
3) We analyze the impact of the interlanguage 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 interconnection 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 interlanguage calling conventions.

Fig. 1 shows an example of JNI source code from Conscrypt. Fig. 1(a) shows the Java class "NativeCrypto", which contains a JNI native method declaration `EVP_PKEY_type(...)`,

[1][Online]. Available: https://scitools.com/

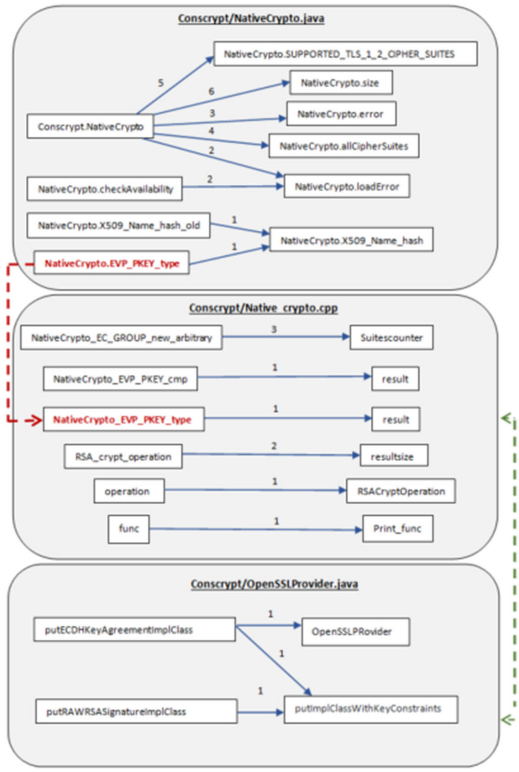


Fig. 2. Dependency call graph for Fig. 1.

while Fig. 1(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 interlanguage dependencies" due to the rules of JNI [13]. Fig. 2 shows the corresponding dependency call graph of Conscrypt for Fig. 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 interlanguage 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. Fig. 2 shows an example of indirect interlanguage 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 article, "Seven-Zip, the ".../src/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 interlanguage dependencies in Section III-B2.

Not being able to identify interlanguage 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 article is to empirically study the interlanguage 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 interlanguage 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.

1) Interlanguage 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). Interlanguage dependencies could be direct or indirect.
   a) Direct interlanguage dependencies (DILD): an interlanguage dependency ensuring a static direct call between two files according to the multilanguage conventions (e.g., a change in native Java class requires changing the native C/C++ function). We use S-MLDA to identify them.
   b) Indirect interlanguage dependencies (IILD): an interlanguage 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.
2) Intralanguage 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 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 analyzed different systems based on their size (number of lines of code). We picked four systems 100k LoC (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 ≥ 1 *M* LoC (i.e., Openj9 and RethinkDB).

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

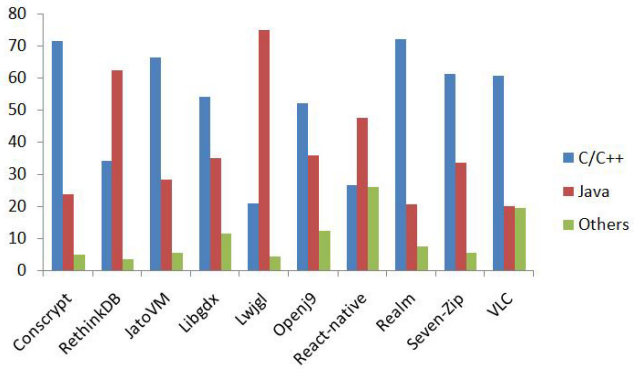


Fig. 3. %Programming languages used in each project.

Fig. 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 Fig. 3 to three languages per system. The "Others" category combines the rest of programming languages.

B. Interlanguage Dependencies Identification

We use static and historic code analysis to identify the interlanguage dependencies.

We designed a first approach called S-MLDA based on JNI's rules as defined by Liang [13] to identify the direct interlanguage dependencies. We designed a second approach called H-MLDA to identify the indirect interlanguage 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 pattern and abstract-level description language (PADL) [15]. It takes as input a set of multilanguage 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 interlanguage dependencies, presented in a dependency call graph showing the general relationships between files and the specific ones between methods. Fig. 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 interlanguage dependencies for JNI systems and generates a 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 Fig. 5.

1) *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.).
2) *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.

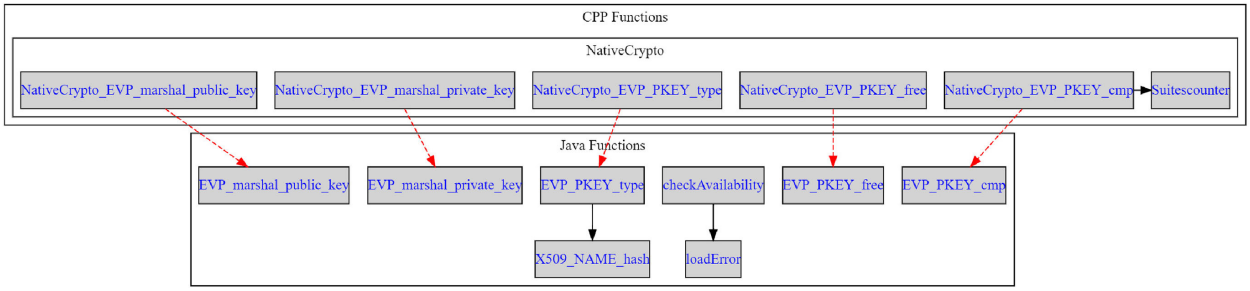


Fig. 4. Dependency call graph of a part of Conscrypt generated by *S-MLDA*.

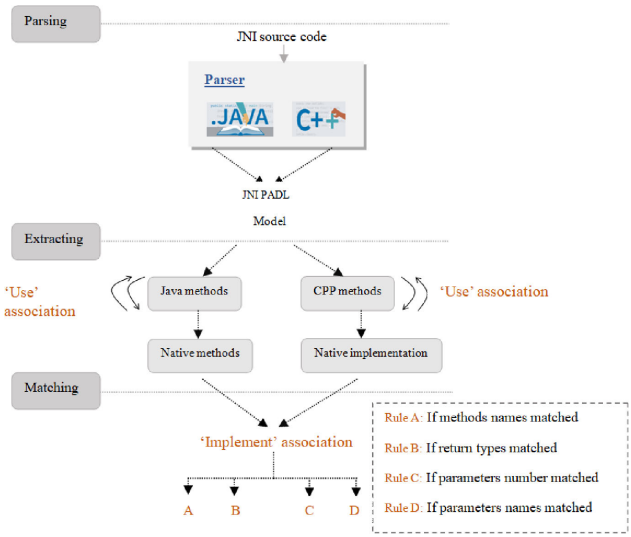


Fig. 5. S-MLDA approach.

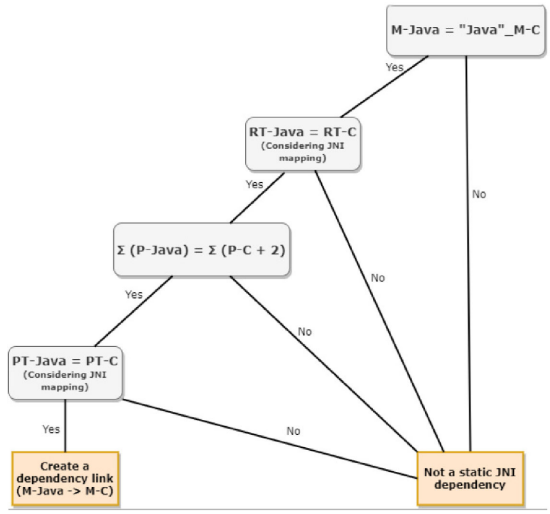


Fig. 6. S-MLDA matching rules.
(M-Java: Java method name; M-C: C function name; RT-Java: Java method return type; RT-C: C function return type; P-Java: Java method parameters; P-C: C function parameters; PT-Java: Java method parameter types; PT-C: C function parameter types).

3) *Matching*: We identify the matched Java native methods with the respective implementations in C++ files based on the JNI rules that we illustrate in Fig. 6.
   a) *Rule A*: We verify if the Java method names (M-Java) match with the C++ function names (M-C) using the JNI naming convention.
   b) *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.
   c) *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.
   d) *Rule D*: Finally, 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 Fig. 4) with different hierarchy levels: classes level and file level. In each class, nodes are the methods and the edges are the dependencies. Edges between nodes across two subgraphs are interlanguage dependencies at methods level. We consider two files having direct interlanguage dependencies, if they involve at least one edge across the two subgraphs. 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 interlanguage dependencies. It identifies the multilanguage cochanges (involving multilanguage 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 Fig. 9). These steps allow retrieving the indirect interlanguage 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 cochange is a useful method to recommend dependent files potentially relevant to a future change request when they are detected by static code analysis. The

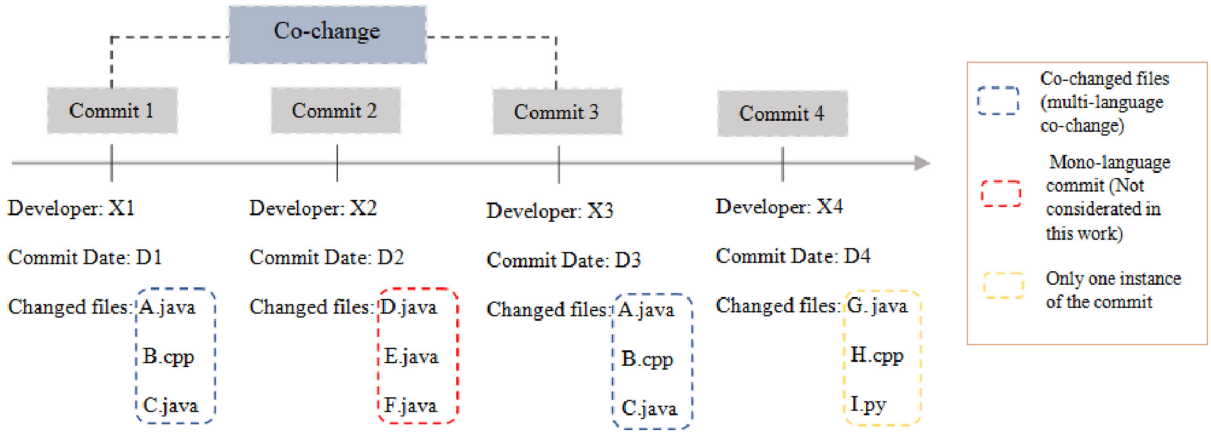


Fig. 7. Mono/multilanguage cochanges analysis.

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 cochange 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 Fig. 7, which shows four different cochanges in a given multilanguage system. Commit 1 and 3 are instances of the same changed three files, i.e., a cochange, involving the same three files. Commit 2 is a mono-language cochange, i.e., involving monolanguage files. Commit 4 presents the case of a multilanguage 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 cochanged accidentally without a significant reason, i.e., files appeared changing together (at commit level) only one time.

Fig. 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 cochanged set of files in two groups: one for the interlanguage cochanges involving Java and C/C++ files and the second one for the intralanguage cochanges 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 interlanguage dependencies, we convert the multilanguage cochanges into sets of multilanguage files, as shown in Fig. 9 in step (1). Then, we remove the ones in common with S-MLDA sets, i.e., the direct interlanguage dependencies as presented in step (2) in Fig. 9, and last, we consider the remaining sets the indirect interlanguage dependencies. This is illustrated in step (3) in the same Fig. 9.

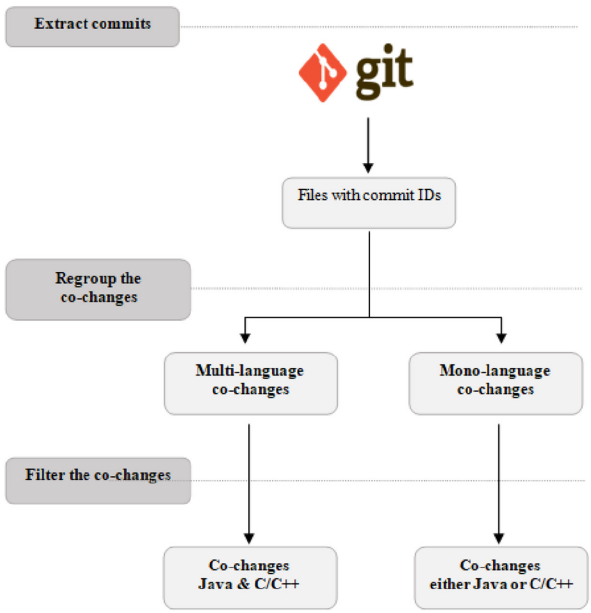


Fig. 8. H-MLDA approach.

C. Quality Issues and Security Vulnerabilities

We aim to understand if the interlanguage dependencies in multilanguage systems introduce more bugs and/or security vulnerabilities than intralanguage 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 dataset 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

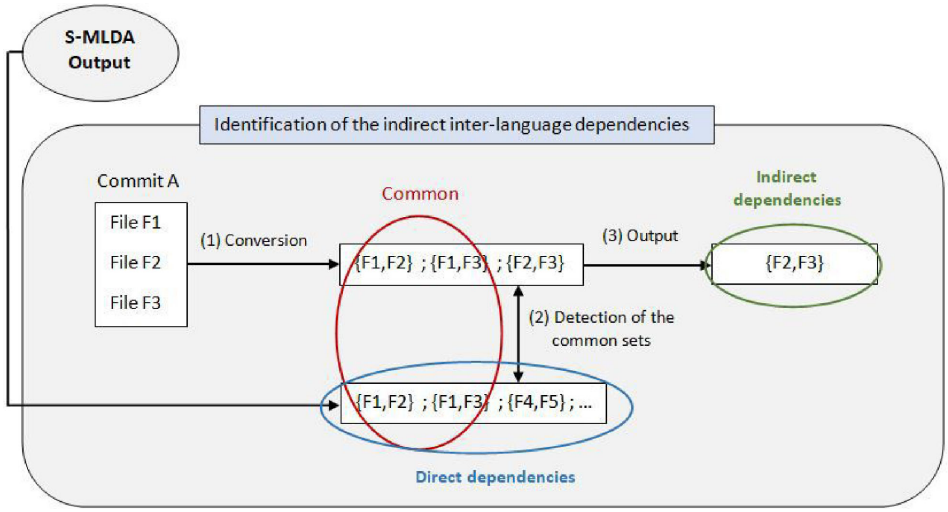


Fig. 9. Identification of indirect dependencies.

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 dataset (the two datasets 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.

RQ1: How common are direct and indirect inter-language dependencies in multilanguage systems?

Motivation: The goal of this research question is to measure the prevalence of interlanguage dependencies. For that, we need to identify the dependent multilanguage files (interlanguage 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 multilanguage commits and the total number of the monolanguage 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 multilanguage commits. Table III shows i) the number of DILD (identified by S-MLDA) and ii) the number of IILD (generated via H-MLDA). Finally, we evaluate the performance of S-MLDA and H-MLDA in order to validate their precision and recall.

[3][Online]. Available: https://tinyurl.com/y2qhkagl

TABLE I
#Mono-/Multilanguage 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

Results: We observed that *multilanguage cochanges are common in multilanguage 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 multilanguage files. Multilanguage 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%.

TABLE III
#(IN)DIRECT INTERLANGUAGE 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 COCHANGES

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

The results from Table III show that *the number of indirect interlanguage dependencies is higher (average of 2.7 times) than the number of direct interlanguage dependencies in 90% of the cases (nine systems), while it is nearly equal for the case of Libgdx.* The high number of indirect interlanguage 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 cochanges 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 multilanguage cochanges* (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 interlanguage 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.

1) Similarity of the files names. We checked if the IILD files in these sets could have a same or similar names, e.g., a substring of a file name A included in file B.
2) 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.
3) 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.

Case of H-MLDA: We implemented a script to identify all the interlanguage 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 interlanguage 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.

RQ2: Are interlanguage dependencies more risky for multilanguage software system in terms of quality?

Motivation: Dealing with dependencies in multilanguage 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)interlanguage dependencies and the quality issues.

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

Results: The percentage of bug-introducing cochanges was as high as 46,66% in interlanguage dependencies and 13,7% in the case of intralanguage dependencies. We report that when the number of interlanguage 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 cochanges are constant for intralanguage dependencies, with values range between 9,77 and 13,70. Hence, there is no significant correlation between bugs and intralanguage dependencies.

The box-plot in Fig. 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 Fig. 11 allows a better analysis to answer the research question with the following. The more the *X*-axis in Fig. 11(a) increases for interlanguage dependencies, the higher the risk of bugs (blue color) being introduced, while

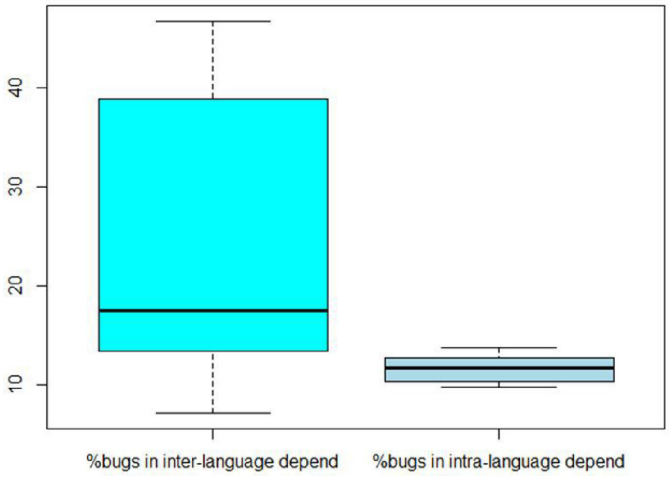
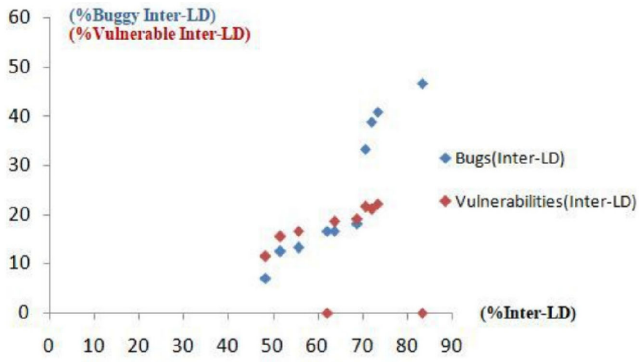
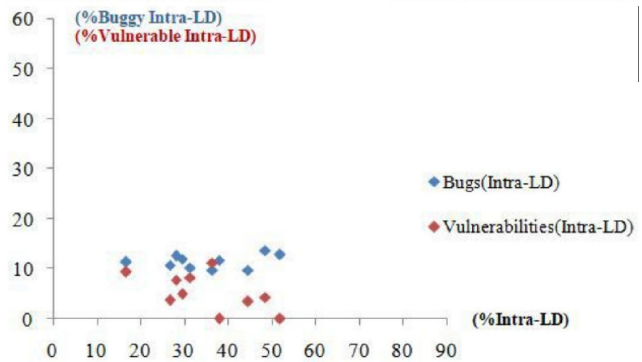


Fig. 10. %Buggy dependencies.



(a)

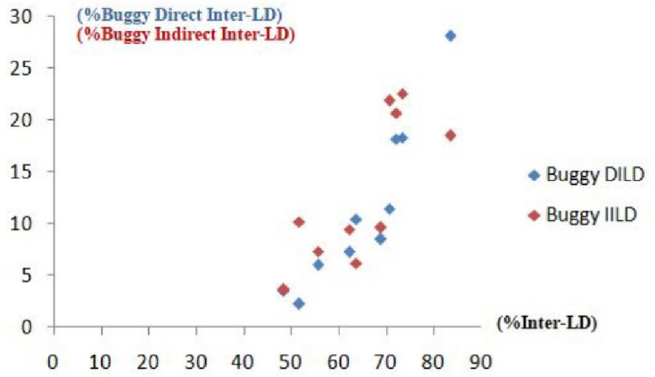


(b)

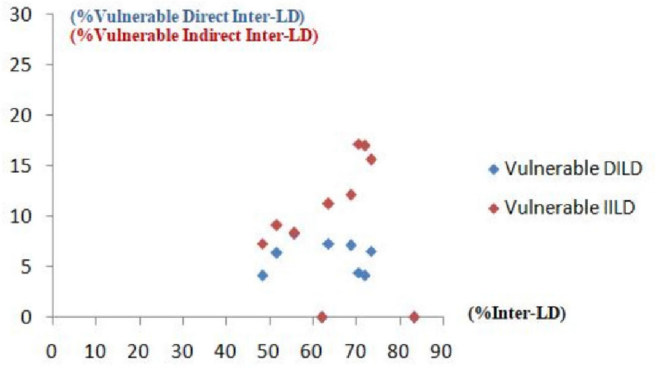
Fig. 11. %Quality and security issues detected in intra- and interlanguage dependencies. (a) Interlanguage dependencies. (b) Intralanguage dependencies.

this risk remains constant for intralanguage dependencies [see Fig. 11(b)].

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



(a)



(b)

Fig. 12. %Quality and security issues detected in (in)direct interlanguage dependencies. (a) Buggy DILD and IILD. (b) Vulnerable DILD and IILD.

percentage of bugs presented in the inter-LD and the percentage of bugs presented in the intra-LD.

The results in Fig. 11 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 Fig. 12(a). 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 interlanguage dependencies (46,66%) is higher nearly to three times compared with quality issues in cochanges involving intralanguage 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 multilanguage 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.

The following are two examples of bugs introduced within interlanguage dependencies.

1) The first example was extracted from *Conscrypt*. It presents a cochange 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 interlanguage 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.
2) The second example concerns a cochange extracted from *Realm*. It involved interlanguage 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 interlanguage dependencies more risky for multilanguage 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 multilanguage systems, security vulnerabilities in those systems became a priority for developers [10]. This research question aims to identify the security impact of the interlanguage dependencies in multilanguage systems. For this, we present the relationship between the inter/intralanguage 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 as follows.

1) *Memory faults*: Such as buffer overflows and other memory corruptions, which impact the security of a software system.
2) *Null-pointer exceptions*: Which can threat the system confidentiality when it is used to reveal debugging information.
3) *Initialization checkers*: Which can threat the system integrity when the software components are used without being properly initialized.
4) *Race conditions*: Related to weak time checking between software tasks and can allow an attacker to obtain unauthorized privileges.

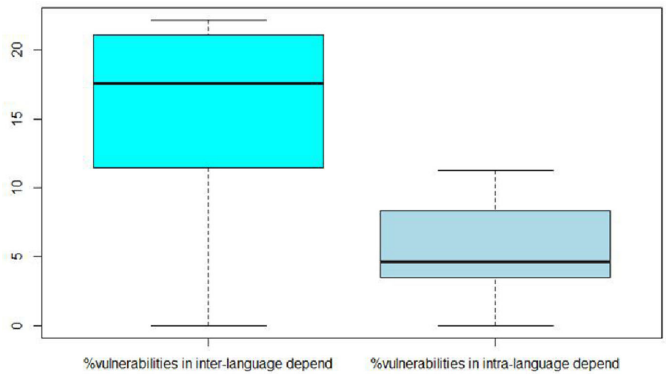


Fig. 13. %Vulnerable dependencies.

5) *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 intralanguage dependencies. The percentage of vulnerabilities in interlanguage dependencies can reach up to nearly 22,18%, and 11,27% in intralanguage dependencies. We present the findings in a scatter-plot, Fig. 11, for a better visual analysis. We observe that, the more we have interlanguage 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 interlanguage dependencies and security vulnerabilities. The box-plot presented in Fig. 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 Fig. 12(b) 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 interlanguage dependencies (22,18%) are twice as common as in intralanguage dependencies (11,27%). This leads to the conclusion that interlanguage dependencies are more risky than intralanguage dependencies and developers should consider the challenge provided with multilanguage systems.

We observe from Table IV that no vulnerabilities were found in interlanguage 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++

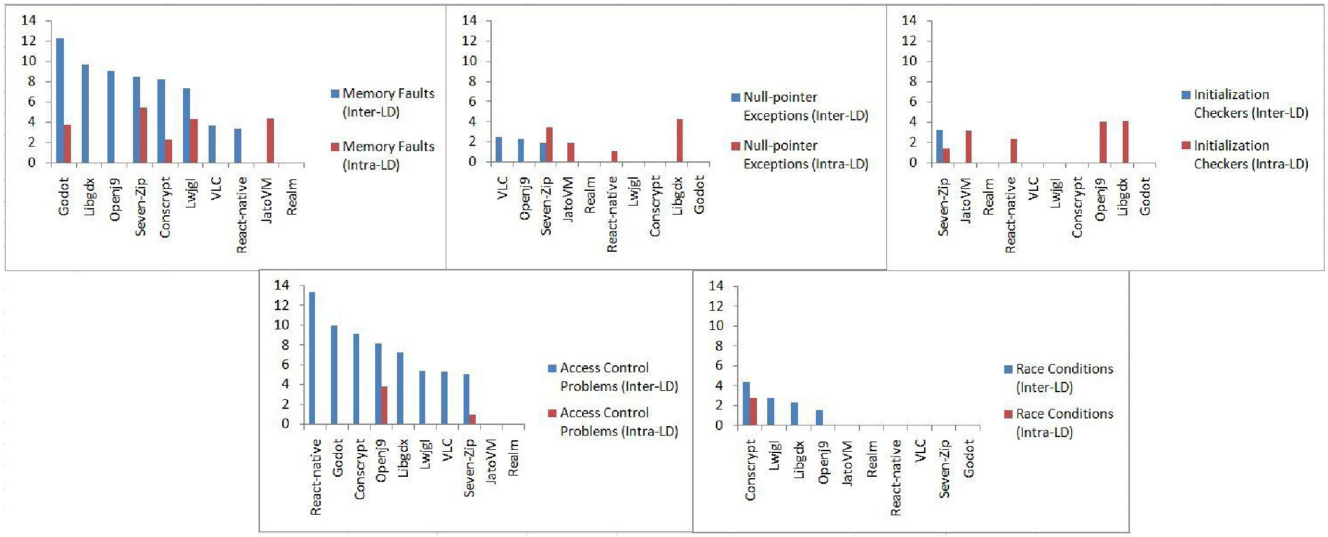


Fig. 14. Distribution of the five security vulnerabilities categories.

[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 multilanguage systems.

JatoVM is the implementation of the Java virtual machine. Vulnerabilities have not been found within interlanguage dependencies, however, they were detected within intralanguage 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 interlanguage 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. Fig. 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, and Race conditions. We can observe from the bar-plot 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.* [24], through their study, supported this finding as Memory faults, e.g., buffer_overflow are the most known vulnerability subject of security issues in multilanguage 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

Threats to internal validity. We relied on the literature to extract the JNI rules and to apply the cochanges 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 the PADL meta model, cochanges method, SZZ algorithm, and vulnerability classification may present a threat to constructing this study as other (more effective) approaches may exist. However, we mentioned that many previous works relied on these approaches and validated them. For this reason, the precision and the recall of these approaches is a concern that we agree to accept.

Regarding H-MLDA and S-MLDA, these approaches present some limitations. Regarding S-MLDA, we based our algorithm on the standard keywords of JNI as defined in Liang's book [13], yet it is possible that new programming rules or new implementation methods have appeared in recent versions of JNI. Our article relied on the first book on JNI that describes the standard and the general basis for JNI development. For H-MLDA, it is possible that we suffer from too fine-grained or too coarse-grained commits [26]. Too fine-grained commits occur when two changes that should have been added together in one commit somehow ended up in different commits, for example because together they fix the same bug but needed

different developers to collaborate, or because a developer forgot some changes in the first commit. Conversely, too coarse-grained commits, also known as tangled commits [26], occur when unrelated changes accidentally were added in the same commit, for example, a bug fix and a small refactoring. This usually happens due to convenience (the refactoring opportunity was found while fixing a bug), or because the developer forgot that both changes were staged for an upcoming commit. The issue with tangled commits is that they will make all changes (and changed modules) inside the commit appear to be related, which could mislead the results, and hence cause noise and bias.

Threats to external validity. Our results may not be representative of all multilanguage projects, since we only studied the case of JNI on ten open-source projects. The choice to study these projects was motivated by the fact that this language combination (Java/C++) is the most used in multilanguage programming. However, it should be noted that the proposed H-MLDA and S-MLDA approaches can be easily extended and applied to other language combinations. In particular, the historical technique (H-MLDA) could be applied as-is to other multilanguage software systems with other combinations of languages such as the Android ecosystem because the technique is based on the concept of cochange. Hence, it just needs to select the file extensions of interest (such as Kotlin and Java) in the analyzed commits. In our study, we limited the analyzed commits to Java and C/C++ files. The static technique (S-MLDA), however, will require an implementation of the specific calling convention rules for any new language combination that is considered. This is because each combination of languages follows a specific methodology to create a bridge that ensures language A being able to call language B and vice versa.

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 multilanguage dependencies. They applied island grammars to detect dependencies in multilanguage software systems.

Nguyen *et al.* [27] studied multilanguage dependencies in web applications where they focused on analyzing 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 cochanges 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 multilanguage system could have and that make the static code analysis a hard task for the developer. They proposed a solution based on 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 and Adams [29] highlighted the challenge of identifying configuration options through a multilayer 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 interlanguage dependencies in multilanguage systems.

Jaafar *et al.* [6] presented a novel approach called *Macocha* to validate two change patterns from analyzing the cochanges: the asynchrony change pattern, corresponding to macro cochanges (MC), that is, of files that cochange within a large time interval (change periods), and the dephased change pattern, corresponding to dephased macro cochanges, i.e., MC that always happens with the same shifts in time. They applied *Macocha* on seven diverse systems. The authors considered only monolanguage 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 cochanging files. They argued that cochanges could be useful to recommend dependent entities potentially relevant for future change. They used the cochange history in CVS to extract cochanging files. The authors did not consider the dependency aspects of the identified cochanges. Moreover, their algorithm could only be applied to one language at a time, e.g., not a multilanguage concept.

VII. Conclusion

Multilanguage 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 article analyzed interlanguage 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 multilanguage systems i) to identify the IILD and DILD and ii) to study their impact on the quality and the security of multilanguage 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 multilanguage developers to first use specific approaches (i.e., S-MLDA) during their multilanguage changes to identify the inter-LD especially the (hidden) indirect ones. Second, they were recommended to take care of the quality aspect while changing a multilanguage software, i.e., specific approaches were needed to help spot quality issues during the change tasks. Finally, we recommend to emphasize more the control on the memory fault and access control problem when dealing with multilanguage systems.

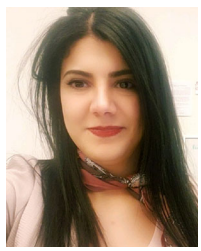
In future work, we plan to 1) generalize this study by analyzing more combinations of programming languages; 2) conduct studies with developers to understand their perspective on multilanguage developments; and 3) perform further qualitative studies to understand the types (and characteristics) of the bugs and vulnerabilities introduced by multilanguage programming in order to understand better why/how these bugs are introduced.

ACKNOWLEDGMENT

The authors would like to thank Y.-G. Guhneuc for his valuable inputs and the Ministre de l'conomie et de l'Innovation - Qubec for its support.

REFERENCES

[1] F. Boughanmi, "Multi-language and heterogeneously-licensed software analysis," in *Proc. 17th Work. Conf. Reverse Eng.*, 2010, pp. 293–296.
[2] M. M. Lehman and L. A. Belady, *Program Evolution: Processes of Software Change*. New York, NY, USA: Academic, 1985.
[3] B. Cossette and R. J. Walker, "Polylingual dependency analysis using island grammars: A cost versus accuracy evaluation," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2007, pp. 214–223.
[4] S. Hassaine, F. Boughanmi, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "A seismology-inspired approach to study change propagation," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 53–62.
[5] A. Shatnawi *et al.*, "Static code analysis of multilanguage software systems," 2019, *arXiv:1906.00815*.
[6] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "Detecting asynchrony and dephase change patterns by mining software repositories," *J. Softw., Evolution Process*, vol. 26, no. 1, pp. 77–106, 2014.
[7] B. Dit *et al.*, "Impactminer: A tool for change impact analysis," in *Proc. Companion Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 540–543.
[8] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proc. 26th Int. Conf. Softw. Eng.*, 2004, pp. 429–445.
[9] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, Sep. 2004.
[10] M. Abidi, M. Grichi, F. Khomh, and Y.-G. Guéhéneuc, "Code smells for multi-language systems," in *Proc. 24th Eur. Conf. Pattern Lang. Programs.*, 2019, p. 13.
[11] H. Brunelire, J. Cabot, G. Dup, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Inf. Softw. Technol.*, vol. 56, no. 8, pp. 1012–1032, 2014.
[12] M. Grichi, M. Abidi, Y.-G. Guéhéneuc, and F. Khomh, "State of practices of java native interface," in *Proc. 29th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, 2019, p. 10.
[13] S. Liang, *Java Native Interface: Programmer's Guide and Reference*. Reading, MA, USA: Addison-Wesley, 1999.
[14] F. Jaafar, Y. Gueheneuc, S. Hamel, and G. Antoniol, "An exploratory study of macro co-changes," in *Proc. 18th Work. Conf. Reverse Eng.*, Oct. 2011, pp. 325–334.
[15] Y. Guhneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 667–684, Sept./Oct. 2008.
[16] S. Mcintosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Sep. 2014, pp. 241–250.
[17] N. Ali, F. Jaafar, and A. E. Hassan, "Leveraging historical co-change information for requirements traceability," in *Proc. 20th Work. Conf. Reverse Eng.*, 2013, pp. 361–370.
[18] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proc. Int. Workshop Mining Softw. Repositories*, vol. 30, no. 4, pp. 1–5, 2005.
[19] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities," in *Proc. Conf. Availability, Rel. Secur.*, 2009, pp. 804–810.
[20] R. L. Scheaffer, W. MendenhallIII, R. L. Ott, and K. G. Gerow, *Elementary Survey Sampling*. Boston, MA, USA: Cengage, 2011.
[21] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric Statistical Methods*, vol. 751. New York, NY, USA: Wiley, 2013.
[22] G. Tan, S. Chakradhar, R. Srivaths, and R. D. Wang, "Safe Java Native Interface," in *Proc. IEEE Int. Symp. Secure Softw. Eng.*, 2006, vol. 97, p. 106.
[23] G. Tan and J. Croft, "An empirical security study of the native code in the JDK," in *Proc. 17th Conf. Secur. Symp.*, 2008, p. 13.
[24] M. Abidi, M. Grichi, and F. Khomh, "Behind the scenes: Developers perception of multi-language practices," in *Proc. 29th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, 2019, p. 7281.
[25] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang, "Safe java native interface," in *Proc. IEEE Int. Symp. Secure Softw. Eng.*, 2006, vol. 97, p. 106.
[26] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 121–130.
[27] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Cross-language program slicing for dynamic web applications," in *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 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 *Proc. IEEE Int. Conf. Softw. Maintenance*, 2001, pp. 42–51.
[29] M. Sayagh and B. Adams, "Multi-layer software configuration: Empirical study on wordpress," in *Proc. 15th Int. Work. Conf. Source Code Anal. Manipulation*, 2015, pp. 31–40.



Manel Grichi (Graduate Student Member, IEEE) received the bachelor's degree (license degree) and the software engineering degree from the University of Tunisia, Tunis, Tunisia, in 2011 and 2014, respectively. She is currently working toward the Ph.D. degree with the Department of Computer and Software Engineering, Polytechnique Montreal, Montral, QC, Canada.

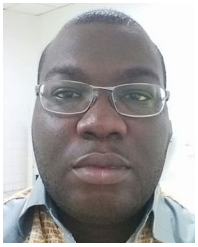
Her current research interests include mining software repositories, data analysis, software quality/security, and multilanguage development.

Mouna Abidi is currently working toward the Ph.D. degree with the SWAT Lab, Department of Computer Science and Software Engineering, Polytechnique Montreal, Montreal, QC, Canada.

She is interested in studying and improving the quality of multilanguage systems. Her research interests include design patterns, antipatterns, and code smells for monolanguage and multilanguage systems and their impact on software quality.

Fehmi Jaafar received the Ph.D. degree from the Department of Computer Science, Montreal University, Montreal, QC, Canada.

He is currently a Researcher with the Computer Research Institute of Montreal and an Affiliate Assistant Professor with the Department of Computer Science and Software Engineering, Concordia University. Previously, he was an Adjunct Professor with Concordia University of Edmonton, and a postdoctoral research fellow with Queens University School of Computing in Kingston, and at Polytechnique Montreal. He established externally funded research programs in collaboration with Defence Canada, Safety Canada, NSERC, MITACS, and industrial partners. His has authored and coauthored in top venues in computer sciences, including the Journal of Empirical Software Engineering and the Journal of Software: Evolution and Process. His research interests include cybersecurity in the Internet of Things, in the analysis and the improvement of the security and quality of software systems, and in the application of machine learning techniques in cybersecurity.



Ellis E. Eghan received the Ph.D. degree from the Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada.

He is currently a Postdoctoral Fellow with the MCIS lab, Polytechnique Montreal. He has currently authored and coauthored five papers in major refereed international journals and conferences. His research interests include release engineering and improving software engineering tasks through the semantic analysis of software build systems.

Bram Adams (Member, IEEE) received the Ph.D. degree from the GH-SEL Lab, Ghent University, Ghent, Belgium.

He is an Associate Professor with Queen's University, Kingston, ON, Canada. His has authored and coauthored premier software engineering venues such as EMSE, TSE, ICSE, FSE, MSR, ASE, and ICSME. His research interests include mining software repositories, software release engineering, and the role of human affect in software engineering.

Dr. Adams has been PC Co-Chair of SCAM 2013, SANER 2015, ICSME 2016, and MSR 2019. In addition to co-organizing the RELENG International Workshop on Release Engineering from 2013 to 2015 (and the 1st IEEE Software Special Issue on Release Engineering), he co-organized the SEMLA, PLATE, ACP4IS, MUD, and MISS workshops, and the MSR Vision 2020 Summer School.