IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 August 2025, accepted 4 September 2025,
date of publication 10 September 2025, date of current version 25 September 2025.

*Digital Object Identifier 10.1109/ACCESS.2025.3608237*

RESEARCH ARTICLE

# Xades: Static Taint Analysis Framework for C#-Based Multilingual Applications

**O. SEHWAN, MINHO KIM, YOUNGHOON BAN, AND HAEHYUN CHO**
School of Software, Soongsil University, Seoul 06978, South Korea

Corresponding author: Haehyun Cho (haehyun@ssu.ac.kr)

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by Korea Government (MSIP) (Generative AI Based Binary Deobfuscation Technology and Its Evaluation Metrics) under Grant RS-2024-00399389.

**ABSTRACT** The C# development ecosystem, centered around the .NET Framework, underpins various platforms for mobile, desktop, and web application development. These platforms support multilingual development, enabling developers to incorporate not only C# but also a variety of subsidiary programming languages. With the rise of multilingual applications, ensuring secure interaction between different programming languages has become a critical aspect of software security. However, prior analysis approaches for Android applications have primarily focused on Java, Python, and C, leaving the risks in C#-based multilingual applications largely unaddressed. We propose a static taint analysis approach tailored for C#-based multilingual applications, along with an implementation for the Xamarin platform, named Xades, to detect sensitive data leaks. Xades offers an overview of how C# code interfaces with other programming languages by constructing a comprehensive call graph that captures cross-language method invocations. To ensure accuracy, Xades analyzes C# code in detail, addressing accessor methods and method chaining to produce a precise call graph. It, then, performs context-, flow-, field-, object- sensitive taint analysis with the call graph. Following the analysis of the C# codebase, Xades produces a taint analysis summary and translates it into Jimple IR, to analyze Xamarin.Android applications through a modified FlowDroid. We evaluated Xades on our benchmarks and real-world apps. As a result, Xades successfully detected all data leaks with interactions between C# and Java in benchmarks. It, also, detected 395 out of 6,087 real-world Android applications leak sensitive data.

**INDEX TERMS** Multi-language analysis, static analysis, taint analysis, C#, Xamarin.

## I. INTRODUCTION

C# remains a robust and flexible programming language for developing applications across a wide variety of platforms [11], [12]. Among various C#-based platforms, Xamarin is a widely-used framework that allows developers to create cross-platform mobile applications with the same C# code base [18]. On Xamarin framework, we can implement applications based on C# with other programming languages such as C, Java on Android, and Objective-C on iOS. Thanks to its convenience, many developers have adopted Xamarin, and we found that the distribution of Xamarin applications in the Android app market increased from 17.23% to 34.59% between 2020 and 2022.

We, therefore, find that the need for software assessment for Android applications developed with Xamarin in light of this increase. In addition, C# malware variants currently pose a threat across multiple platforms, largely due to the cross-platform capabilities and the increasing adoption of C# in various domains [3], [11], [16], [17]. Recently, security experts have discovered that the Xamalicious, an Android malware developed using Xamarin, has infected over 327,000 devices and can be deployed across the other operating systems with minimal changes [8], [13], [14], [15]. Such cross-platform data leaks, especially in multilingual applications, pose serious risks to both end users and service providers, including privacy violations, unauthorized data

The associate editor coordinating the review of this manuscript and approving it for publication was Sedat Akleylek.

access, and potential regulatory non-compliance. Such data leaks can also lead to significant financial and reputational damage, exposing both users and service providers to substantial risk. Consequently, these concerns underscore the urgent need for robust static analysis techniques tailored to the unique challenges of C#-based multilingual applications. To capture malicious actions from such malware on Android, we need not only an analyzer for C# codebase, but also an integrated analysis platform that can handle data and control flows across interactions between multiple languages. However, previous analysis approaches for Android [22], [27], [37], [41] are limited to C and Java, and thus, the risks of the other code remain unaddressed.

In this work, we introduce a static taint analysis approach, named Xades, tailored for detecting sensitive data leaks from C#-based multilingual applications. To this end, we tackle strong technical challenges as follows: (1) We should accurately capture and represent all interactions between C# and other programming languages. (2) C# applications have complex control flows and object-oriented features. With inheritance and interface-based polymorphism, data flow can become difficult to track, especially when methods are overridden or when abstract base classes are involved. (3) C# applications have a large codebase, making efficient static analysis more difficult.

To tackle the challenges, (1) Xades first provides a high-level view to show how C# code interacts with different programming languages by constructing a call graph that includes method invocations to the other programming languages. In addition, to generate a precise call graph, Xades inspects C# code in detail for resolving accessor methods and handling method chainings. (2) It, then, performs context-, flow-, field-, object- sensitive taint analysis with the call graph, examining each statement in detail for inferring the actual data of instances. (3) To optimize the taint analysis process, Xades excludes methods that do not contribute to the data flow between source and sink methods from its analysis scope. Also, Xades does not perform data flow analysis in the internal code of known platform and library method calls. After analyzing C# codebase, Xades generate a summary of taint analysis results and transform them into Jimple IR to analyze Xamarin.Android applications with modified FlowDroid [25]. With this approach, we can expand the analysis scope of Xades to other programming languages with an aid of other analyzers, enabling taint analysis of C#- based multilingual applications.

To evaluate Xades, we categorized data flows between C# and other languages that can lead to sensitive data leakages and we implement benchmark with these data flows in Xamarin. We, also, collected a dataset by identifying Xamarin applications from the Google Play Store. In our evaluations, Xades detected all sensitive data leaks in the benchmark. Also, Xades found data leaks in 395 out of 6,087 apps from the Xamarin apps from PlayStore. Additionally, Xades analyzed C# open-source libraries from NuGet [2] to show the effectiveness. The contributions of this work are as follows:

- Identification of technical challenges in analyzing C#-based multilingual applications.
- Implementation of a novel static analysis tool, Xades, that analyzes C# applications in context-, flow-, field-, object- sensitive manner, identifying data leaks across multiple programming languages in Xamarin.Android applications.
- Identification of the growing use of Xamarin apps in the Android market, motivating the need for analyzing C#-based multilingual applications.
- Evaluation of Xades with the benchmarks and real-world applications, including Xamarin apps in the PlayStore and C# libraries in NuGet.

## II. RELATED WORK

### A. ANALYSIS OF C# APPLICATIONS

Developers have been creating C# apps across diverse domains using .NET-based development platforms. Therefore, researchers have recognized the importance of analyzing these applications and have conducted relevant studies [24], [26], [31], [32], [38]. SharpChecker [32] provides analysis for C# program properties, such as null reference access and resource leaks, using Roslyn. However, SharpChecker is designed for source code analysis without any extraction from compiled applications.

### B. ANALYSIS OF ANDROID APPLICATIONS

The core components of an Android application comprise Java bytecode stored in `classes.dex`, along with metadata about components in `AndroidManifest.xml`, layouts, and more. During runtime, Android apps have a complex lifecycle driven by callback methods triggered by user input. These executions pose challenges for static analysis in generating static modeling. Several studies [25], [29], [33], [42], [43] have proposed approaches to generate modeling of Android applications for analyzing program properties such as sensitive data leakage and malware detection. FlowDroid [25] and Amandroid [42] have introduced modeling and static analysis on Android applications written in a single language (Java or Kotlin). Notably, FlowDroid utilizes a `dummyMainMethod` as an entry point, managing control flow through outgoing edges to potentially executable methods triggered by user actions such as activities and callback methods. FlowDroid then conducts taint analysis from this entry point to detect sensitive data leakages. However, FlowDroid's coverage excludes native libraries used by the NDK, prompting further research to address this limitation.

### C. ANALYSIS OF MULTILINGUAL APPLICATIONS

Analyzing multilingual applications with interactions between multiple programming languages presents challenges due to the difficulty in generating unified modeling.

**TABLE 1. Cross-language interaction support in prior multilingual analysis approaches.**

| Approach | Java–C/C++ | C#–Java | C#–C/C++ |
|---|---|---|---|
| NDroid [44] | ✓ | - | - |
| TaintArt [39] | ✓ | - | - |
| JN-SAF [41] | ✓ | - | - |
| JuCify [37] | ✓ | - | - |
| Soot-dotnet [24] | - | △ | - |
| SCIL [28] | - | △ | - |
| Xades | △ | ✓ | ✓ |

Researchers have proposed approaches [24], [28], [30], [35], [37], [39], [40], [41], [44] to tackle these challenges and effectively analyze such multilingual applications.

PolyCruise [35] converts data flow in multilingual programs comprising Python and native languages (C/C++) into a language-independent symbolic representation. This transformation provides unified modeling for analyzing symbolic dependencies and runtime data flow across different languages. Soot-dotnet [24] focuses on converting C#, represented as CIL code, into Jimple for the Soot framework. However, Soot-dotnet faces limitations in generating unified modeling from multilingual applications with diverse FFI, such as wrappers in Xamarin. Additionally, differences in syntax and type between C# and Java hinder its accuracy and effectiveness.

Several studies [37], [39], [41], [44] have explored the analysis of complete Android applications with native libraries, addressing interactions between Java and native (C/C++) code. JN-SAF [41] introduced unified modeling of Java and native languages and proposed an approach for detecting data leakage through symbolic execution. JuCify [37] translated native functions into Jimple for unified modeling and applied this translated Jimple to FlowDroid for taint analysis. However, these studies primarily focused on Android's NDK, limiting their applicability to analyzing interactions between different languages in applications using alternative platforms such as Xamarin.Android.

In addition, there have been studies [28], [30], [40] that proposed approaches to analyze Xamarin.Android applications, which have gained prominence in the application market. For example, Christensen et al. [28] introduced Simple Common Intermediate Language (SCIL) for C# data flow analysis. SCIL extends CIL with a focus on data flow analysis, employing Single Static Assignment form. While these studies provided unified modeling by analyzing wrappers in Xamarin.Android, they did not address the analysis of internal code within unknown APIs. Consequently, they may not detect data leakages occurring within internal Java methods invoked by C#.

Despite these efforts, as shown in Table 1, prior analysis approaches have predominantly focused on interactions between Java and native code (C/C++), while providing limited or no support for analyzing C#-based multilingual applications. In particular, most existing techniques overlook the interaction between C# and Java or native components, which are essential for accurately analyzing platforms such as Xamarin.Android. This lack of comprehensive cross-language modeling hinders the detection of critical issues, such as data leakage through unknown Java APIs invoked by C# code. To address these limitations, we propose Xades, a unified analysis framework that supports C#–Java and C#–C/C++ interactions in multilingual applications.

## III. BACKGROUND

### A. XAMARIN PLATFORM

Xamarin is a cross-platform software development framework based on the .NET Framework. It enables the development of mobile applications for Android, iOS, watchOS, tvOS, and Tizen using a C# codebase [18].

Xamarin.Android facilitates interactions between C# and Java by using APIs that resolve differences in data types and syntax across languages [19]. These APIs serve as the foundation for two wrappers: (1) Android Callable Wrappers (ACW): It is a JNI bridge used whenever the Android runtime needs to invoke C# code [20]. (2) Managed Callable Wrappers (MCW): It is JNI bridge used whenever C# code needs to call Android API methods [21].

Also, Xamarin.Android applications can interact between C# and C/C++ via .NET compilers, and between Java and C/C++ using the Android NDK. Therefore, for static analysis of Xamarin applications, analyzers must account for each language individually as well as the interactions among all three languages.

Xamarin.Android applications can facilitate interactions between C# and Java through MCW and ACW, between C# and C/C++ via .NET compilers, and between Java and C/C++ using the Android NDK. Therefore, for static analysis of Xamarin applications, analyzers must account for each language individually as well as the interactions among all three languages.

### B. ANALYZING CROSS-LANGUAGE INTERFACES

Various software development platforms support multilingual development, allowing for the reuse of libraries written in different languages. Multilingual applications involve various interactions between different programming languages. Such interactions can lead to software bugs or vulnerabilities; therefore, analyzers for multilingual applications must be capable of covering these interactions.

Static analysis approaches for analyzing interactions between Java and C/C++ in Android applications have been proposed [27], [37], [41]. These approaches perform taint analysis on Android applications with native libraries, focusing on data flows between Java and C/C++ to detect sensitive data leaks. On the other hand, Xamarin.Android applications supports interactions among C#, Java, and C/C++. For vetting those applications, we need an approach that can analyze not only each individual language but

also all the interactions. However, previous approaches for analyzing multilingual applications have focused on specific development platforms and languages (C and Java). Therefore, in this work, we propose a novel approach that analyzes multilingual applications implemented with the Xamarin platform for detecting sensitive data leaks.

## IV. DESIGN

### A. OVERVIEW

C#-based multilingual applications such as Xamarin.Android apps frequently involve method calls between C# and other languages. These cross-language invocations can introduce security vulnerabilities such as sensitive data leaks that are difficult to detect with traditional static analysis techniques. To address this, we propose an extended modeling approach that accurately captures interactions between C# and other languages. Because it is important to identify data leakages not only related to intended behavior in malicious applications but also arising from unintended behavior due to the developer's negligence in benign applications [25].

In the section below, we overview the design of our analyzer, named Xades, to show how we tackle the challenges. In § IV-B, we introduce detail designs. First off, the key challenges for performing static taint analysis of C#-based multilingual applications are:

1) Constructing a call graph that accurately represents call relationships across various types, including typical methods, instance constructors, instance member methods, and accessors. Also, the call graph should capture all interactions between C# and other programming languages.
2) Performing context-, flow-, field-, object- sensitive taint analysis in C#.
3) Handling the large codebase of C# to enable efficient static analysis of applications.

#### 1) GENERATING A CALL GRAPH

To provide an accurate call graph that can provide a high-level view to show how C# code interacts with different programming languages, we generate a call graph representing not only C# methods but also methods in other languages invoked by C#.

$$
\begin{aligned}
&G = (V, E)\\
&n \in V, \quad E \subseteq V \times V\\
&n = (v_1, v_2, v_3, v_4)\\
&v_1 = \texttt{[namespace|package|library].class}\\
&v_2 = \text{method name}\\
&v_3 = \{\text{param}_0 \text{ type}, \ \text{param}_1 \text{ type}, \ \ldots\}\\
&v_4 = \text{return type}\\
&V_n \text{ key} := v_1/v_2\ (v_3)v_4
\end{aligned}
$$

We designed the call graph structure to represent methods from multiple languages, such as C#, Java, and C/C++. The call graph consists of a set of vertices $V$, which represent methods, and edges $E$ that define the invocation relationships between these methods. Each method node $V$ contains detailed information, denoted by attributes $v_1$, $v_2$, $v_3$, and $v_4$. $v_1$ encompasses high-level concepts of the method, including namespaces in C#, packages in Java, library names in C/C++, and class names in object-oriented programming languages. $v_3$ and $v_4$ handle challenges where methods cannot be uniquely identified using only $v_1$ and $v_2$, particularly in cases of method polymorphism. To facilitate identification of nodes, each method node is assigned a unique key, such as $V_n$ key, allowing precise data tracking throughout the graph.

Wrappers for calling functions from other languages in C# are a combination of complex interface code. Since the call graph representing all these function calls can cause confusion in the analysis, we simplify the nodes and edges corresponding to these combinations into a single entity. Furthermore, our approach requires an examination of how the wrappers provided by the development platform are structured within applications. We investigated the wrappers provided by the Xamarin platform, and this will be explained in § IV-B2.

#### 2) TAINT ANALYSIS FOR C#

Taint analysis can detect sensitive data leakages by analyzing data flows from a source method to a sink method [25]. To perform such taint analysis in an effective and sound way, our approach requires flow- and context-sensitive data flow analysis tailored to C#, an object-oriented programming language. The C# language has complex syntax for various operations such as instance creation, modification of instance member variables, and invocation of instance member methods. Consequently, we design an inferred data structure that mirrors C# objects and perform symbolic execution to determine the actual data present in context by sequentially traversing the statements.

When we encounter an object creation statement, we analyze the invoked constructor's code to infer not only the object's fields but also the member variables initialized in the constructor. The inferred elements are then stored in a custom structure designed to reflect C# objects. Additionally, the C# codebase can exhibit control flows that branch and merge due to conditional statements. In certain contexts, a single variable may therefore be inferred to hold multiple values depending on the conditions. To manage this complexity, we store the inferred data of these symbols in a pool, allowing us to verify multiple data points.

#### 3) HANDLING LARGE CODEBASE

The codebase of C# applications based on the .NET Framework is large. In general such applications include developer-written code, platform code necessary for executing the CLR on target devices, and library code reused from NuGet repositories [1]. Consequently, performing exhaustive data flow analysis by visiting every line of code would lead to significant performance overhead. To optimize static analysis

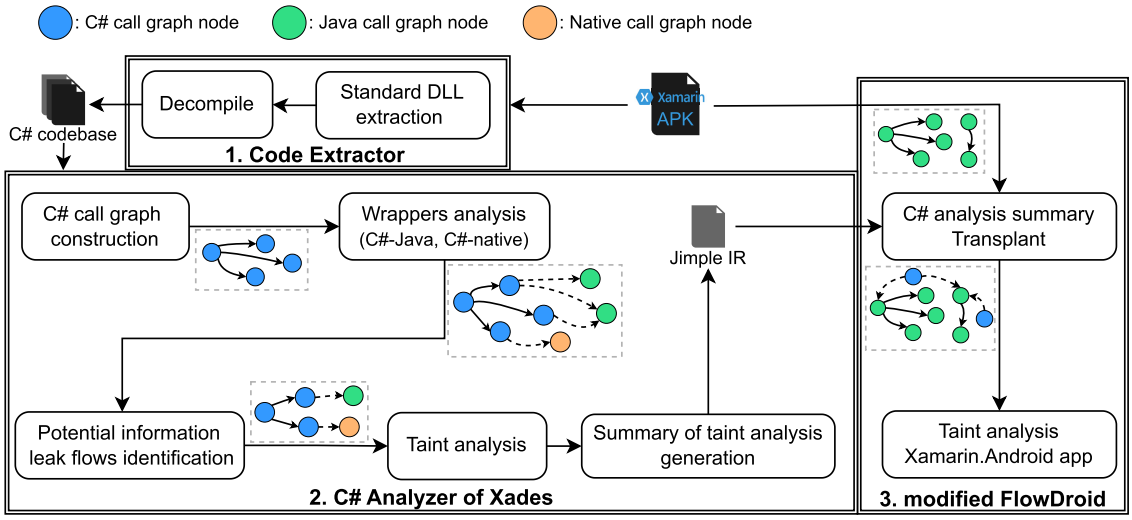



**FIGURE 1.** The process of static taint analysis in Xades.

in our approach, we focus on the following strategies: First, Xades does not perform data flow analysis on the internal code of known platform and library method calls. Next, Xades excludes methods that do not contribute to the data flow between source and sink methods from the analysis scope. Specifically, this optimization process involves searching for the nodes corresponding to source and sink methods within the call graph, and generating a sub call graph that only includes the nodes with paths to these methods.

#### 4) TRANSFORMING ANALYSIS RESULTS

If, as a result of the taint analysis on C# codebase, we observe that data obtained from a source method flows to a method that is implemented in another programming language, Xades considers and highlights those flows as a potential data leakage related to the invocation of other languages' methods. For investigating and confirming that such cases can occur actual sensitive data leaks, we need to expand the analysis results by using static analyzers implemented for the other programming languages. To this end, we convert analysis results of Xades into a format compatible with other analyzers. Specifically, in this work, we translate analysis results to Jimple statements, and thus, Java analyzers such as Flowdroid [25] can directly use that for analyzing Xamarin.Android applications. By doing so, we can expand the analysis boundary of Xades to other programming languages with the help of other analyzers, ensuring thorough detection of cross-language data leaks.

### B. XADES

In this work, we design and implement Xades for static taint analysis of C# applications. Also, by using the Xades, we perform taint analysis for Xamarin.Android apps, which are C#-based multilingual applications incorporating C#, Java, and native code, with FlowDroid [25] customized for this work. Figure 1 shows the static taint analysis process in Xades. Xades comprises three main modules: (1) the code extractor, (2) the C# analyzer, and (3) the modified FlowDroid. First off, the code extractor extracts the C# codebase from the Xamarin.Android application. Next, the C# analyzer processes the C# codebase to generate a call graph, detect potential data leaks, and produce an abstracted summary of these leaks. Then, the modified FlowDroid generates unified analysis results by the summary of C# taint analysis and Java bytecode together—the modified FlowDroid performs taint analysis to detect sensitive data leaks. Xades uses lists of source and sink methods from .NET APIs, which we defined by inspecting all APIs, and Android APIs provided by FlowDroid [25].

#### 1) XADES' CODE EXTRACTOR

As the first step of analysis, Xades extracts C# codebase from Xamarin apps. Xamarin.Android apps consist of native libraries that run the Mono runtime on Android devices, as well as the Common Intermediate Language (CIL) files. The CIL files generated by the Xamarin compiler include the following formats:

1) Standard DLL files: These files have the `MZ` magic bytes. They can be decompiled into C# code by using typical decompilation tools (e.g., dotPeek [4], ILSpy [5]).
2) DLL files compressed by the Xamarin: The Xamarin compiler compresses files via the Lz4 [7] library. They can be identified by the `XALZ` magic bytes, rendering them unable to be decompiled by typical decompilation tools.
3) A file merging compressed DLL files: The `assemblies.blob` is file merged with compressed DLL files, while the `assemblies.manifest` provides information about the merged files.

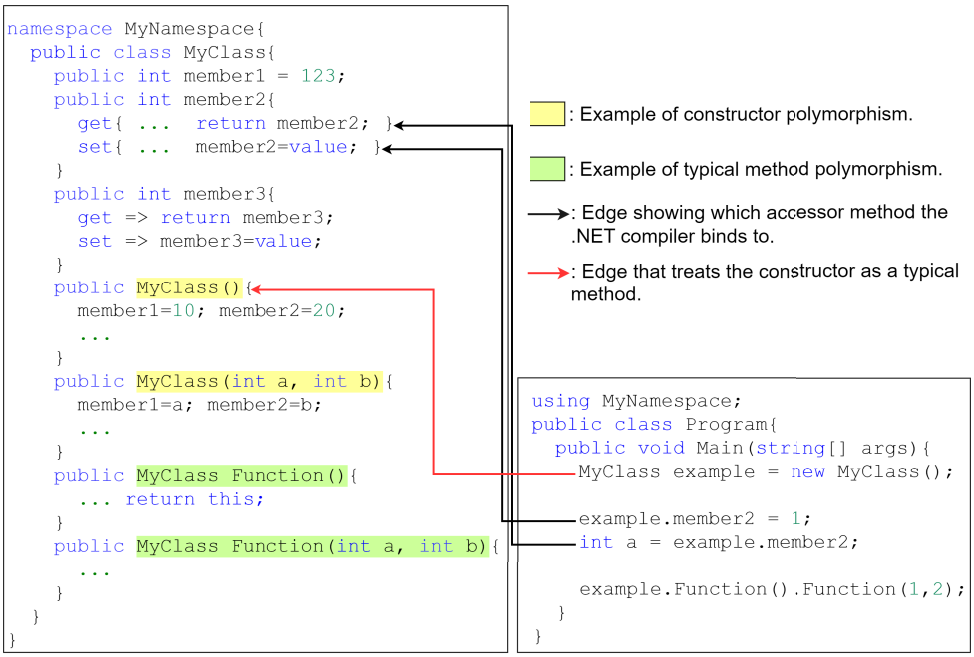



**FIGURE 2.** An Example of C# code with instance constructors, accessor methods of member variables, and method chaining.

Xades' code extractor begins by unzipping the APK file and verifying the structure of the contained CIL files. If the `assemblies.blob` exists, the code extractor parses this file to retrieve the compressed DLL files. The code extractor then utilizes the Lz4 library to decompress these compressed files into standard DLL files. Next, the code extractor decompiles the standard DLL files into the C# codebase with ilspycmd [6].

### 2) XADES' C# ANALYZER

We implemented a C# analyzer to generate a call graph and detect potential data leaks by using Roslyn [10]. Roslyn is the .NET compiler SDK and provides syntax trees and semantic models for the C# language, which are parsed during the .NET compilation process. Our C# analyzer performs static analysis using these syntax trees and semantic models during the re-compilation of the extracted C# codebase.

**Call graph generation.** Xades' C# analyzer initially generates a call graph representing method in C#, Java, and C/C++ though two phases: (1) *C# call graph construction*, and (2) *Wrappers analysis*.

(1) *C# call graph construction:* Figure 2 shows an example of C# code with the various syntax that C# analyzer includes in the C# call graph: typical methods, instance constructors, instance member methods, and accessors (e.g., `get` and `set` for instance member variables). As shown in `MyClass()` and `MyClass.Function()`, we cannot identify methods in object-oriented languages solely by their names. Therefore, the C# analyzer creates nodes that include method parameters and return types, following the call graph structure as depicted in § IV-A1. Furthermore, when the C# analyzer encounters constructor invocations for instance initialization, it generates nodes and edges in the call graph, treating them as typical methods, such as `MyClass()`.

The .NET compiler binds specific accessor methods, represented by black edges in Figure 2, for assignment statements modifying instance member variables at compile time. Therefore, the C# analyzer determines whether an accessor appears on the left-hand or right-hand side of an assignment to resolve accessor methods. If the accessor is on the right-hand side (e.g., `a = example.member2`), it is bound to `a = example.get_member2()`. Conversely, if it is on the left-hand side (e.g., `example.member2 = 1`), it is bound to `example.set_member2(1)`. The C# analyzer adds specific nodes and edges to the call graph to represent these accessor methods.

Additionally, method chaining, where multiple method calls are combined in a single statement. To handle method chaining, the C# analyzer parses statements with chained method calls and adds nodes and edges for each method call. For example, in the statement `example.Function().Function(1,2)`, both invocations of `Function()` are analyzed separately.

(2) *Wrappers analysis:* Xamarin.Android platform facilitates interactions among C#, Java, and C/C++ through its wrappers. In the C# codebase, we can find interactions between C# and Java through the wrappers provided by Xamarin, as well as interactions between C# and C/C++ via the language interoperability supported by the .NET framework. To generate a call graph, the C# analyzer analyze these interactions and then adds virtual nodes and edges

**An example of interface between C# and Java in Xamarin.Android application**

```
namespace Android.Telephony;
[Register("android/telephony/TelephonyManager", DoNotGenerateAcw = true)]
public class TelephonyManager : Java.Lang.Object{
  private static readonly JniPeerMembers _members = new XAPeerMembers("android/telephony/TelephonyManager",
   typeof(TelephonyManager));
  private static Delegate cb_getDeviceId;
  ...
  public unsafe virtual string DeviceId{
  [RequiresPermission("android.permission.READ_PRIVILEGED_PHONE_STATE")]
    get{
      return JNIEnv.GetString(
       _members.InstanceMethods.InvokeVirtualObjectMethod("getDeviceId.()Ljava/lang/String;",this, null).
       Handle, JniHandleOwnership.TransferLocalRef);
    }
  }
```

**An example of interface between C# and native languages in Xamarin.Android application**

```
[DllImport("realm-wrappers", CallingConvention = CallingConvention.Cdecl, EntryPoint = "query_float_greater")]
public static extern void float_greater(QueryHandle queryPtr, IntPtr columnIndex,
  float value, out NativeException ex);
```

**FIGURE 3.** Example codes: (above) C# and Java using MCW; (below) C# and native languages using .NET Framework interoperability.

representing Java and native methods into the C# call graph.

Figure 3 shows interface code in Xamarin apps for interaction among other languages. Xamarin generates wrappers during compilation with combination of interface APIs such as `JNIEnv` and `InvokeVirtualObjectMethod`. We can gather detailed information about Java methods invoked from C# by analyzing the MCWs. Additionally, to simplify these interactions, the C# analyzer only includes nodes and edges that represent the Java methods, excluding the combinations of interface APIs. The `get_DeviceId()` in C# actually invokes the `getDeviceId()` in Java.

Xamarin utilize the language interoperability provided by the .NET Framework, which facilitates by `DLLImport` attributes. The interface between C# and the native languages shows that the C# method `float_greater()` invokes the native method `query_float_greater()` within a `librealm-wrappers.so`. In our call graph structure, the high-level concept of method represented by $v_1$ encapsulates the library and class containing native methods.

The C# analyzer provides a high-level view to show how C# code interacts with different programming languages by analyzing interfaces among C#, Java, and native languages. Additionally, we identify ''unknown APIs'' which refer to methods in untrusted or unknown modules of another programming languages that are invoked by C#. In interactions between C# and other languages, sensitive data can be transferred from C# to such unknown APIs. When Xades performs taint analysis, if it detects sensitive data flows to those APIs, it highlight them as potential data leakages, and then, Xades inspects them with other analyzers such as FlowDroid.

**Static taint analysis.** Xades' C# analyzer constructs a sub call graph in (1) *Potential information leak flows identification* to reduce the analysis scope before proceeding to (2) *Taint analysis*.

(1) *Potential information leak flows identification:* Xamarin.Android apps have a large codebase, including Xamarin platform libraries and various third-party libraries used by developers. To optimize static taint analysis, the C# analyzer constructs sub call graph to restrict its scope to methods with reachable paths to source or sink methods.

Initially, the C# analyzer identifies and adds nodes representing source and sink methods to the sub call graph in the call graph. It then traverses the incoming edges backward from these nodes, adding nodes and edges that have reachable paths to source or sink methods to the sub call graph. Consequently, the root nodes of sub call graph possess reachable paths to both source and sink method. Therefore, the C# analyzer performs taint analysis starting from these root nodes of the sub call graph, treating them as entry points. Furthermore, this approach restricts the scope of analysis and eliminates the need to connect complex lifecycle, such as callbacks and events, to a single entry point.

(2) *Taint analysis:* Xades begins taint analysis by examining the data flow of methods at the root nodes of the sub call graph. The C# analyzer conducts data flow analysis for C# using control-flow graphs, semantic models, and other resources provided by Roslyn. Specifically, the C# analyzer store inferred data in pool, termed to `value pool`, with variable names serving as symbols. To handle various types of inferred data, we define the `value pool` as the `System.Object` type, which serves as the base type for all objects in C#. When the visited statement contains a method invocation expression, the C# analyzer visits the invoked method to analyze its internal code.

However, symbolic execution performed line by line can result in significant performance overhead. Therefore, the C# analyzer checks whether a method belongs to one of the following categories: methods in the Xamarin platform or source and sink methods. When encountering invocation statements for these methods, the C# analyzer infers up to

the invocation expression without delving into the internal code.

Also, the C# analyzer detects invocations of "unknown APIs" as potential data leaks, regardless of whether they involve tainted parameters. Because the internal code of unknown APIs could lead to actual data leaks in other languages.

**Summary of taint analysis results generation.**

We generate abstracted taint analysis results for using it with other analyzers. To this work, we use specific pointers: `BaseInstance` and `FromSource`. `BaseInstance` points to an instantiation statement that precedes a method call, such as `new instance1()` before `instance1.foo();`. `FromSource` points to the invocation of a source method where tainted data was identified during the analysis. Based on these pointers, the C# analyzer creates a summary that represents the context and flow of sensitive data in the C# code.

In addition, the C# analyzer generates the summary with detected potential data leaks to analyze internal code of Java methods invoked by C#. This summary consists of context- and flow- sensitive behavior focusing potential data leakages. The C# analyzer creates summaries of instance constructor calls, represented by `BaseInstance`, before non-static method calls. Additionally, it constructs source method calls that lead to tainted data using `FromSource` before generating a summary of sink method calls. Albeit in this work, we generate the summary to analyze Java method using FlowDroid, the summary can be easily transformed to any intermediate representations for the other analyzers.

**Taint analysis for Xamarin.Android app.** FlowDroid generates static modeling of Java bytecode by parsing Android APK files, but it does not account for behaviors written in C# within Xamarin apps. We integrated Xades with FlowDroid to enhance coverage for detecting sensitive data leaks in interactions between C# and Java. To this end, we modified FowDroid to transplant the summary of C# analysis results into the modeling of Java. To be specific, the modified FlowDroid transforms the abstract summary into Jimple classes, methods, and active bodies. It, also, converts C# data types that cannot be represented in Java to `java.lang.Object`. Additionally, it transforms C# source and sink methods into virtual Java methods: `dummyMainClass_csharp: java.lang.Object csSource()` and `void csSink(java.lang.Object)`. These virtual methods enable the modified FlowDroid to detect data leaks involving the invocation of C# methods.

As we discussed earlier, the C# abstract summary includes only tainted data among parameters of invocations. Consequently, other parameters are represented as `null` rather than inferred with actual values. To finalize the parameter propagation process of FlowDroid, Xades replaces `null` parameters with appropriate dummy values that match their respective data types. After converting C# methods into Jimple, Xades adds edges from the `dummyMainMethod` to C# methods in the FlowDroid's call graph, thus completing the integration.

Finally, Xades' FlowDroid performs taint analysis to detect sensitive data leaks involving both C# and Java. Because the modified FlowDroid finalizes its parameter propagation by assigning dummy values in place of `null` at method calls, Xades focuses on potential data leaks. Furthermore, we used extended source and sink lists recommended by FlowDroid by introducing virtual methods such as `csSource` and `csSink`, representing .NET API source and sink methods. By integrating these virtual methods, Xades detects sensitive data leaks occurring during interactions between C# and Java within Xamarin.Android apps.

**TABLE 2. Number and proportion of Xamarin apps in the real-world Android application market (Google Play Store).**

| Year | 2020 | 2021 | 2022 | 2023 | 2024 |
|---|---|---|---|---|---|
| # total apps | 231,531 | 132,474 | 42,844 | 26,751 | 8,366 |
| # Xamarin apps | 39,902 (17.23%) | 31,797 (30.12%) | 14,821 (34.59%) | 9,238 (34.53%) | 2,737 (32.72%) |

## V. EVALUATION

Our evaluation addresses the following research questions:

- **RQ1:** How widely are C#-based multilingual applications used in the Android application market?
- **RQ2:** Can Xades detect data leaks in complex interactions between C# and other languages?
- **RQ3:** Can Xades detect data leaks in real-world Xamarin applications?
  - **RQ3.a:** How much interaction exists between C# and other languages in these applications?
  - **RQ3.b:** How efficient is Xades in terms of its costs?
- **RQ4:** Can Xades be used for analyzing real-world C# apps not Xamarin?

To answer these questions, we conducted static analysis of C#-based multilingual applications using our implementation, Xades, on a PC running Windows 10 with an Intel i9-12900 CPU and 64GB RAM.

### A. RQ1: C#-BASED MULTILINGUAL APPLICATIONS IN ANDROID APPLICATION MARKETS

We collected real-world Xamarin apps from the AndroZoo [23] repository. To identify Xamarin apps, we verified the presence of crucial native libraries, such as `libxamarin-app.so` and `libmonodroid.so`, within the downloaded APK files. These libraries are essential for running the Xamarin platform on Android devices. Collected Xamarin.Android applications from the PlayStore are detailed in Table 2. The adoption of the Xamarin platform by developers steadily increased during the observed period. These results indicate a growing adoption and a considerable market share.

**RQ1.** Our findings show that Xamarin, a C#-based multilingual platform, is experiencing growth in the

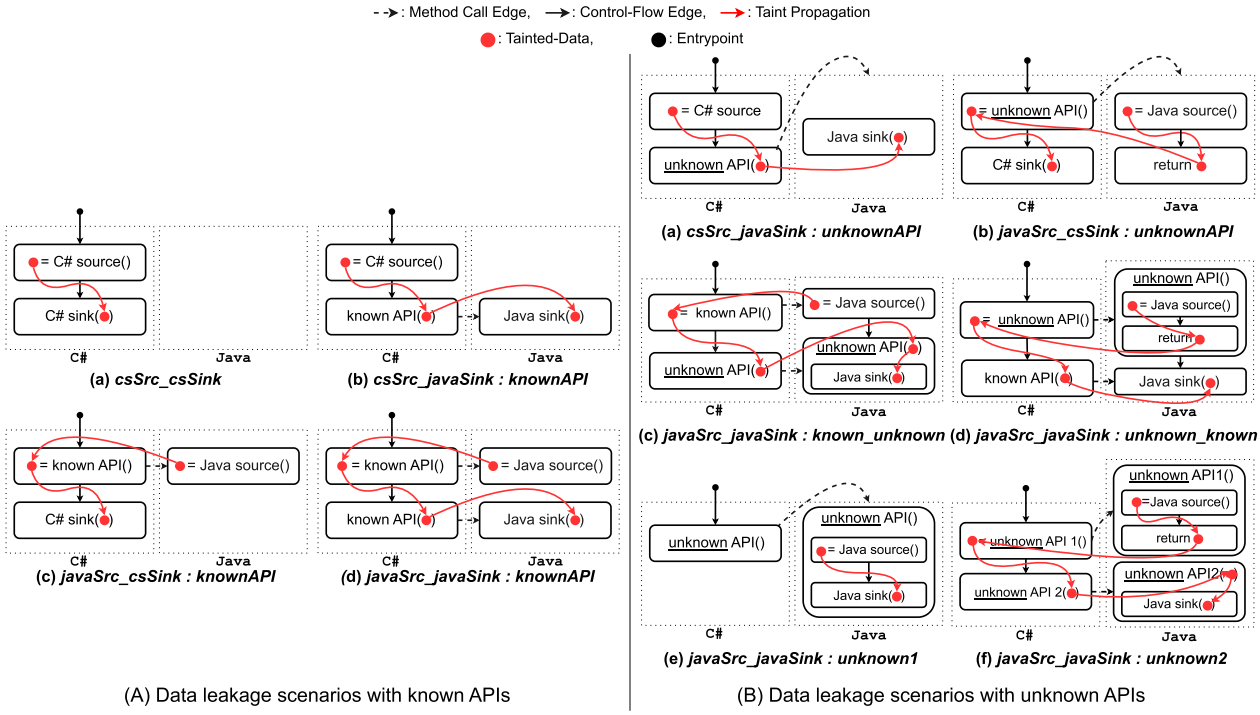



**FIGURE 4.** Data leakage scenarios with known APIs(A) or unknown APIs(B) in Xamarin.Android applications.

Android market. This result highlights the critical need for conducting security assessments on C#-based multilingual applications.

### B. RQ2: DATA LEAK SCENARIOS WITH INTERACTIONS BETWEEN C# AND JAVA IN XAMARIN APPS

In § IV-B2, we described that Xades' C# analyzer identifies "unknown APIs" during the construction of extended call graph. In this research question, we define 10 data leak scenarios involving interactions between C# and Java and construct a benchmark application to test these scenarios. Figure 4 illustrates the data and control flows in the benchmark application containing C# and Java methods. Specifically, red lines indicate the flow of sensitive data from the return values of source methods to the parameters of sink methods. Figure 4 (A) presents data leakage scenarios involving known APIs, allowing us to omit detailed analysis of the internal behavior of Java methods. In contrast, Figure 4 (B) illustrates cases involving unknown APIs, where analyzing their internal behavior becomes necessary for identifying data leaks.

We then evaluate whether Xades can detect sensitive data leaks in these benchmark scenarios. Known APIs are Java methods that belong to platform packages such as `android.` and `java.`, or are included in FlowDroid's source and sink list. Unknown APIs are Java methods within unknown packages invoked by C#, and thus, Xades analyzes internal code of these Java methods with modified FlowDroid. We manually verified the benchmark behavior and confirmed that all expected data leaks were correctly identified by Xades, ensuring the validity of the evaluation.

**RQ2.** We constructed benchmarks with complex data flows between C# and Java according to 10 data leak scenarios in Figure 4. Xades detected all data leaks in our benchmarks. Unfortunately, state-of-the-art academic Android analysis tools, such as JuCify and JN-SAF do not support C# programs, we could not directly compare analysis results with them.

### C. RQ3: XADES IN REAL-WORLD

We evaluated Xades for real-world Xamarin applications. We provide a detailed evaluation results that Xades found and its modules' performances: *Code Extractor*, *C# Analyzer*, and *modified FlowDroid*. To this end, Xades performed taint analysis on 6,341 randomly selected Xamarin applications from the PlayStore in 2022, with timeouts set to 15 minutes for the C# Analyzer and 45 minutes for the modified FlowDroid.

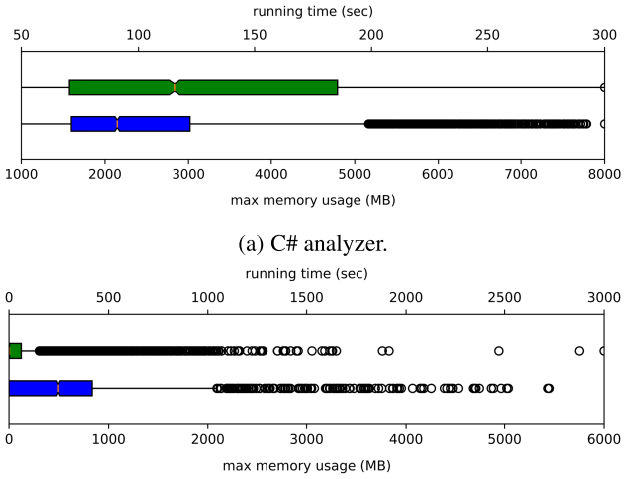
Table 3 presents the results of Xades' static analysis and detailed outcomes from its modules. Xades finalized taint analysis on 5,650 (i.e., 89.10%) real-world apps. Notably, the failures encountered with the *Code Extractor* were due to the Xamarin compiler producing C# code in formats different from those described in § IV-B1. Among 6,341 applications, Xades found that *395 applications* leak sensitive data. These results demonstrate that Xades is a promising framework for detecting sensitive data leaks between C# and other languages in real-world Xamarin apps.

**TABLE 3. Results of Xades' static taint analysis on real-world apps and detailed outcomes from its three modules.**

| | Success | Fail |
|---|---|---|
| Code Extractor | 5,924 (93.42%) | 417 (6.58%) |
| C# Analyzer | 5,865 (92.49%) | 476 (7.51%) |
| *Modified* FlowDroid | 6,086 (95.98%) | 255 (4.02%) |
| Xades (Overall) | **5,650 (89.10%)** | 691 (10.90%) |
| # of apps with data leaks out of 5,650 apps | | **395** |

**TABLE 4. Average number of nodes and edges in an call graph generated by Xades from a real-world app.**

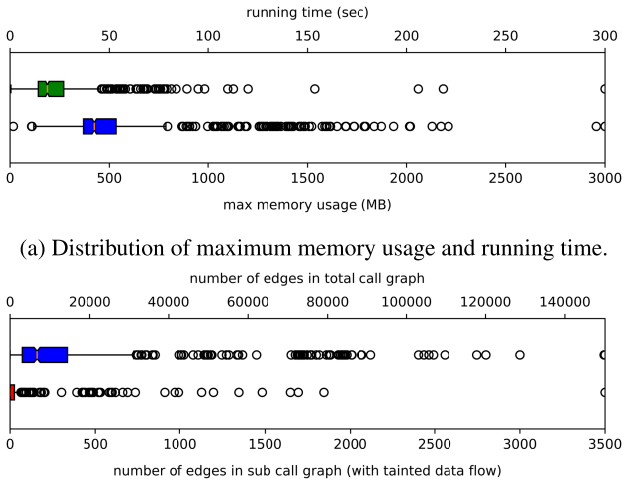
| | C# | Java | Native(C/C++) |
|---|---|---|---|
| # of nodes | 171,321.58 | 14,338.44 (7.69%) | 734.15 (0.39%) |
| # of edges | 533,020.79 | 23,992.76 (4.30%) | 749.22 (0.13%) |
| # of edges to *Unknown API* | | 2,020.87 (8.42%) | 356.57 (47.59%) |
| # of application having edges to *Unknown APIs* | | 5,542 **(87.40%)** | 4,888 **(77.09%)** |




(a) C# analyzer.

(b) Modified FlowDroid.

**FIGURE 5. Distribution of maximum memory usage and running time of the Xades' C# analyzer and modified FlowDroid for Xamarin.Android applications.**




(a) Distribution of maximum memory usage and running time.

(b) Distributed number of edges in call graphs and sub call graphs.

**FIGURE 6. Distribution of maximum memory usage and running time of the C# analyzer for libraries in the NuGet repository(a) with distributed number of edges in call graphs and sub call graphs(b).**

#### 1) RQ3.A: HOW MANY INTERACTIONS EXISTS BETWEEN C# AND OTHER LANGUAGES IN THESE APPLICATIONS?

Table 4 presents the average number of nodes and edges in a call graph generated by Xades' C# analyzer. Notably, the average number of edges going to unknown APIs written in Java and native languages is 2,020.87 (8.42%) and 356.57 (47.59%), respectively. Furthermore, 87.33% and 77.03% of applications exhibit interactions between C#, Java, and native languages through the invocation of unknown APIs.

#### 2) RQ3.B: HOW EFFICIENT IS XADES?

To evaluate the performance of Xades, we measured its max memory usage and running time for analyzing Xamarin.Android applications. Specifically we measured performances of Xades' *C# analyzer* and modified FlowDroid.

Figure 5a shows the distribution of max memory usage and running time of static taint analysis on real-world apps by Xades' C# analyzer. The average of C# analyzer's max memory usage is 2,686 MB and running times is 161.13 seconds. Figure 5b shows the max memory usage (mean = 572.63 MB) and running time (mean = 97.14 seconds) of static taint analysis by Xades' modified FlowDroid.

> **RQ3.** We observed significant numbers of Xamarin.Android applications and interactions between C# and the other languages in real-world Android applications, which highlights the need of effective C# analyzers that can expand the boundary of current analysis scopes. The performance evaluation results of Xades demonstrate the efficiency in performing static taint analysis on C#-based multilingual applications for detecting sensitive data leaks.

### D. CAN XADES BE USED FOR ANALYZING REAL-WORLD C# APPS NOT XAMARIN?

To evaluate the compatibility and performance of Xades, we analyzed 557 feature-specific projects from 6 popular library solutions in NuGet [2] by using Xades. Figure 6 shows the distribution of max memory usage and running time for analyzing these libraries. Xades utilized 579.74 MB of memory on average and average running time for analyze a library is 27.72 seconds. Figure 6, also, depicts the distribution of the number of edges in the call graph and the sub call graph generated during the optimization process for our tainted data flow as described in § IV-B2.

> **RQ4.** Xades can effectively and efficiently analyzes real-world C# apps.

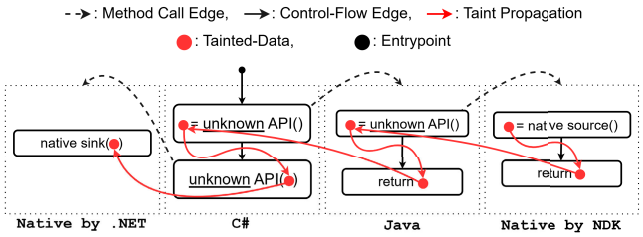



**FIGURE 7.** Data flows with interactions between C#, Java, and native languages in Xamarin.Android.

## VI. DISCUSSION

As multilingual development becomes increasingly common, data may flow between C#, Java, and native code components. Without proper safeguards, such flows can expose sensitive user information, potentially resulting in privacy violations, regulatory non-compliance, and reputational damage, particularly in widely distributed mobile applications.

To address these risks, Xades constructs a unified call graph that models inter-language method invocations across C#, Java, and native code. It then applies context-, flow-, field-, and object-sensitive taint analysis to track sensitive data, even across wrapper-based interoperation layers. To ensure scalability for real-world applications, Xades limits analysis to call graph paths relevant to source-to-sink propagation and skips internal analysis of well-known framework and library methods. For cross-language integration, Xades translates C# taint summaries into Jimple IR, enabling existing Java-based tools such as FlowDroid to analyze both languages within a unified framework.

In our evaluation, Xades effectively detected sensitive data leaks in real-world Xamarin.Android applications and multilingual interactions in large-scale C# libraries. This demonstrates its potential to enhance security analysis for C#-based multilingual applications.

**Limitations.** In our evaluation, Xades demonstrated promising performances in performing taint analysis for C#-based multilingual applications. However, Xades has several limitations: Although Xades initiates the extraction of the C# codebase from Xamarin.Android applications, we encountered 417 failures. The failure was due to the codebase not being structured in the format specified in § IV-B1. The Common Intermediate Language (CIL) within C# applications is compiled into native machine code using a Just-In-Time (JIT) compiler at runtime. Additionally, the .NET Framework supports Native Ahead-Of-Time (AOT) compilation [9], which converts C# code into native assembly to improves runtime performance by bypassing JIT compilation. As a result, some Xamarin.Android apps include C# native libraries (.so files) instead of `assemblies.blob` or CIL files, complicating code extraction. We plan to improve the robustness of our extraction process to better support various compilation formats and code structures, including AOT-compiled native binaries.

Xades inherits limitations from existing analysis tools integrated with the C# analyzer. FlowDroid [25] has known limitations such as application modeling for reflective calls [34], multi-threading [36], and dynamic loading [45]. In our evaluation, we conducted separate evaluations of C# analyzer and modified FlowDroid in Xades. Unfortunately, we faced several issues where the modified FlowDroid failed to complete taint analysis due to the timeout. To address this, we plan to explore alternative Java analyzers or enhancements to FlowDroid to improve reliability and coverage in real-world apps.

Xades also inherits a fundamental limitation of taint analysis based on predefined source and sink specifications. For Java methods, we adopted the function list defined by FlowDroid, while for C# methods, we constructed the list based on our own analysis. This approach implies that Xades may fail to detect certain sensitive data leak scenarios that fall outside the scope of the defined sources and sinks. To mitigate this limitation, we intend to systematically manage and refine the source and sink lists for C#, Java, and native methods, thereby enhancing the precision of our taint analysis.

For analyzing C# applications using other programming languages, Xades requires integration with appropriate analysis tools alongside the C# analyzer. In this work, we adopted FlowDroid to analyze data flows between C# and Java. Additionally, Xamarin apps can include native method calls from Java and C#, as shown in Figure 7. These native methods can access interfaces provided by the Android NDK and the .NET framework, enabling cross-language interactions. Figure 7 illustrates a sensitive data leakage scenario involving such native interactions. Therefore, we plan to integrate with a native language analyzer such as JuCify [37] so that the analysis scope can cover native code.

**Threats to Validity.** To verify the accuracy of Xades, we implemented benchmark applications with complex data flows, and then checked whether Xades effectively detects data leaks. We, also, assessed components of extended call graph generated by C# analyzer of Xades in § V-C1, focusing Java and native languages. While the configurations of the extended call graph generated by Xades were not manually verified, we assumed that Xades produces an accurate extended call graph based on the evaluation results from real-world applications. In addition, in our evaluation, Xades found that 395 real-world applications can leak sensitive data. Because detail inspection of all the applications requires huge manual efforts, we could not manually verify them. However we manually inspected randomly selected 50 applications, validating actual information leaks. Therefore, we hypothesized that Xades produced accurate results based on our evaluation results of the benchmarks and the examination of the extended call graph components and 50 real-world applications.

## VII. CONCLUSION

In this work, we proposed a static taint analysis framework, Xades, to analyze C#-based multilingual applications, which centers on analysis of C# codebase. Xades tackles strong technical challenges in analyzing programs implemented

with C# and other programming languages as follows. First, our C# analyzer provides a high-level view to show how C# code interacts with different programming languages by constructing a call graph that includes method invocations to the other programming languages, resolving accessor methods and handling method chaining. The C# analyzer then performs context-, flow-, field-, object- sensitive taint analysis using the extended model. Also, it generates a summary of analysis results into Jimple IR to analyze Xamarin.Android applications with the modified FlowDroid.

In our evaluation, we demonstrated the effectiveness and efficiency of Xades by analyzing both the benchmarks and real-world applications. We believe that our approach and Xades can serve as a stepping stone for static analysis in C#-based multilingual applications.

## REFERENCES


[1] *An Introduction to NuGet*. [Online]. Available: https://learn.microsoft.com/en-us/nuget/what-is-nuget

[2] *C# Libraries in Nuget Repository*. [Online]. Available: https://www.nuget.org/packages

[3] *Development Platforms Based on .NET Framework*. [Online]. Available: https://dotnet.microsoft.com/en-us/

[4] *DotPeek*. [Online]. Available: https://www.jetbrains.com/decompiler/

[5] *ILSpy*. [Online]. Available: https://github.com/icsharpcode/ILSpy

[6] *Ilspycmd*. [Online]. Available: https://www.nuget.org/packages/ilspycmd/

[7] *LZ4 Library Github*. [Online]. Available: https://github.com/lz4/lz4

[8] *McAfee Detects Xamarin-Based Android Malware*. [Online]. Available: https://mexicobusiness.news/cybersecurity/news/mcafee-detects-xamarin-based-android-malware

[9] *Native AOT*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/deploying/native-aot/

[10] *.NET Compiler Platform SDK*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/

[11] *.NET Framework*. [Online]. Available: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework

[12] *.NET Framework*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

[13] *New Android Backdoor Malware Xamalicious Discovered by McAfee Mobile Research Team*. [Online]. Available: https://nquiringminds.com/cybernews/new-android-backdoor-malware-xamalicious-discovered-by-mcafee-mobile-research-team/

[14] (2023). *New Sneaky Xamalicious Android Malware Hits Over 327,000 Devices*. [Online]. Available: https://thehackernews.com/2023/12/new-sneaky-xamalicious-android-malware.html

[15] *Stealth Backdoor Android/Xamalicious Actively Infecting Devices*. [Online]. Available: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/stealth-backdoor-android-xamalicious-actively-infecting-devices/

[16] *Unity*. [Online]. Available: https://learn.unity.com/

[17] *Windows Forms .NET*. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-8.0

[18] *Xamarin*. [Online]. Available: https://dotnet.microsoft.com/en-us/apps/xamarin

[19] *Xamarin API Design*. [Online]. Available: https://learn.microsoft.com/en-us/xamarin/android/internals/api-design

[20] *Xamarin.Android ACW(Android Callable Wrapper)*. [Online]. Available: https://learn.microsoft.com/en-us/xamarin/android/platform/java-integration/android-callable-wrappers

[21] *Xamarin.Android Architecture*. [Online]. Available: https://learn.microsoft.com/en-us/xamarin/android/internals/architecture/

[22] S. Alam, Z. Qu, R. Riley, Y. Chen, and V. Rastogi, "DroidNative: Automating and optimizing detection of Android native code malware variants," *Comput. Secur.*, vol. 65, pp. 230–246, Mar. 2017.

[23] K. Allix, T. F. Bissyandé, J. Klein, and Y. L. Traon, "AndroZoo: Collecting millions of Android apps for the research community," in *Proc. IEEE/ACM 13th Work. Conf. Mining Softw. Repositories (MSR)*, New York, NY, USA, May 2016, pp. 468–471.

[24] S. Arzt, T. Kussmaul, and E. Bodden, "Towards cross-platform cross-language analysis with soot," in *Proc. 5th ACM SIGPLAN Int. Workshop State Art Program Anal.*, New York, NY, USA, Jun. 2016, pp. 1–6.

[25] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, Jun. 2014.

[26] P. Capek, E. Kral, and R. Senkerik, "Towards an empirical analysis of .NET framework and C# language features' adoption," in *Proc. Int. Conf. Comput. Sci. Comput. Intell. (CSCI)*, Dec. 2015, pp. 865–866.

[27] M. Choi, Y. Im, S. Ko, Y. Kwon, Y. Jeon, and H. Cho, "DryJIN: Detecting information leaks in Android applications," in *Proc. ICT Syst. Secur. Privacy Protection*, 2024, pp. 76–90.

[28] M. C. L. Christensen, S. E. Larsen, S. A. H. Bjergmark, and T. P. Nielsen, "Finding data leaks in xamarin apps by performing taint analysis on cil code," Dept. of Computer Science, Aalborg University, Tech. Rep., Jun. 2018.

[29] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Services*, New York, NY, USA, Jun. 2012, pp. 281–294.

[30] P. Grzmil, M. Skublewska-Paszkowska, E. Łukasik, and J. Smołka, "Performance analysis of native and cross-platform mobile applications," *Informat. Control Meas. Economy Environ. Protection*, vol. 7, no. 2, pp. 50–53, Jun. 2017.

[31] V. Koshelev, V. N. Ignatyev, and A. Borzilov, "C# static analysis framework," *Proc. Inst. Syst. Program. RAS*, vol. 28, no. 1, pp. 21–40, 2016.

[32] V. K. Koshelev, V. N. Ignatiev, A. I. Borzilov, and A. A. Belevantsev, "SharpChecker: Static analysis tool for C# programs," *Program. Comput. Softw.*, vol. 43, no. 4, pp. 268–276, Jul. 2017.

[33] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting inter-component privacy leaks in Android apps," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 1, May 2015, pp. 280–291.

[34] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "DroidRA: Taming reflection to support whole-program analysis of Android apps," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2016, pp. 318–329.

[35] W. Li, J. Ming, X. Luo, and H. Cai, "PolyCruise: A cross-language dynamic information flow analysis," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur.)*, Aug. 2022, pp. 2513–2530.

[36] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for Android applications," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 316–325, Jun. 2014.

[37] J. Samhi, J. Gao, N. Daoudi, P. Graux, H. Hoyez, X. Sun, K. Allix, T. F. Bissyandé, and J. Klein, "JuCify: A step towards Android code unification for enhanced static analysis," in *Proc. IEEE/ACM 44th Int. Conf. Softw. Eng. (ICSE)*, Jordan, May 2022, pp. 1232–1244.

[38] R. Shaukat, A. Shahoor, and A. Urooj, "Probing into code analysis tools: A comparison of C# supporting static code analyzers," in *Proc. 15th Int. Bhurban Conf. Appl. Sci. Technol. (IBCAST)*, Jan. 2018, pp. 455–464.

[39] M. Sun, T. Wei, and J. C. S. Lui, "TaintART: A practical multi-level information-flow tracking system for Android RunTime," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Oct. 2016, pp. 331–342.

[40] K. Vishal and A. S. Kushwaha, "Mobile application development research based on xamarin platform," in *Proc. 4th Int. Conf. Comput. Sci. (ICCS)*, Aug. 2018, pp. 115–118.

[41] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: Precise and efficient NDK/JNI-aware inter-language static analysis framework for security vetting of Android applications with native code," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, Oct. 2018, pp. 1137–1150.

[42] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," *ACM Trans. Priv. Secur.*, vol. 21, no. 3, Apr. 2018.

[43] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android malware detection through manifest and API calls tracing," in *Proc. 7th Asia Joint Conf. Inf. Secur.*, Aug. 2012, pp. 62–69.

[44] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. S. Chan, "NDroid: Toward tracking information flows across multiple Android contexts," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 3, pp. 814–828, Mar. 2019.
[45] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving Android malware and dynamic loading technique," *IEEE Trans. Inf. Forensics Security*, vol. 12, no. 7, pp. 1529–1544, Jul. 2017.

**O. SEHWAN** received the B.S. and M.S. degrees from the School of Software, Soongsil University, in 2022 and 2024, respectively, where he is currently pursuing the Ph.D. degree. His research interests include program analysis and system security.

**MINHO KIM** received the B.S. degree in electronic engineering and the M.S. degree in computer science and engineering from Soongsil University, in 2020, where he is currently pursuing the Ph.D. degree with the School of Software. He is a Research Staff with the Cyber Security Research Center. His research interests include binary analysis, software engineering, reverse engineering, and systems security.

**YOUNGHOON BAN** received the B.S. degree in computer engineering from Chungwoon University, Incheon, South Korea, in 2020, and the M.S. degree from the Graduate School of Software Convergence, Soongsil University, in 2022. He is currently pursuing the Ph.D. degree in software with Soongsil University. He is a Ph.D. Researcher with the Cybersecurity Research Center. His primary research interests include evasion attacks, adversarial example generation for PE malware, and deep learning.

**HAEHYUN CHO** received the B.S. and M.S. degrees in computer science from Soongsil University, Seoul, South Korea, in 2013 and 2015, respectively, and the Ph.D. degree in computer science from the School of Computing, Informatics and Decision Systems Engineering, Arizona State University, and especially concentrating on information assurance. He is currently an Assistant Professor with the School of Software and the Co-Director of the Cyber Security Research Center, Soongsil University. His primary research interests include systems security, which is to address and discover security concerns stemmed from insecure designs and implementations. He is also passionate about analyzing, finding, and resolving security issues in a wide range of topics.