Measuring While Playing Fair: An Empirical Analysis of Language and Framework Usage in the iOS App Store

Florian Magin[*]
Fraunhofer SIT | ATHENE
Darmstadt, Germany
florian.magin@sit.fraunhofer.de

Fabian Scherf[*]
Fraunhofer SIT | ATHENE
Darmstadt, Germany
fabian.william.scherf@sit.fraunhofer.de

Martin Renze
Fraunhofer SIT | ATHENE
Darmstadt, Germany
martin.renze@sit.fraunhofer.de

Cléo Fischer
Fraunhofer SIT | ATHENE
Darmstadt, Germany
cleo.fischer@sit.fraunhofer.de

Gwendal Patat
Fraunhofer SIT | ATHENE
Darmstadt, Germany
gwendal.patat@sit.fraunhofer.de

Abstract

Reverse engineering research has mainly focused on binaries compiled from C and C++, however, in the iOS ecosystem, neither of these languages are the focus of application developers. Apple provides their own languages with Objective-C and Swift as the official choices, while third party cross-platform frameworks, like Microsoft's .NET MAUI, Jetpack Compose, Flutter or even React Native promise unified development across iOS and Android. To investigate the relevance of languages for R&D efforts in software understanding, we conduct a historical analysis spanning 84,432 distinct iOS applications over the past five years.

Unlike previous approaches, we sidestep the technical and legal challenges of the FairPlay DRM system used to encrypt iOS apps and demonstrate that FairPlay does not cover various useful metadata, some of which can be used to detect the presence of programming languages in individual binaries and applications. Our key findings show that, as expected, Swift is now included in almost every popular application, however without phasing out Objective-C usage. Additionally, newer cross-platform languages like Flutter and Kotlin have seen a steady increase in use, while .NET has stagnated since 2020. All of these applications still include and interact with Objective-C, demonstrating that cross-language analysis is now an unavoidable challenge in the modern iOS analysis landscape.

CCS Concepts

• **Software and its engineering → Software architectures**; • **Security and privacy** → *Software and application security*; **Software reverse engineering**.

Keywords

iOS, App Store, Framework, Language

[*]Both authors contributed equally to this research.

ACM Reference Format:
Florian Magin, Fabian Scherf, Martin Renze, Cléo Fischer, and Gwendal Patat. 2025. Measuring While Playing Fair: An Empirical Analysis of Language and Framework Usage in the iOS App Store. In *Proceedings of the 2025 Workshop on Software Understanding and Reverse Engineering (SURE '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3733822.3764671

1 Introduction

The closed nature of the iOS ecosystem creates significant challenges for researchers. Application binaries downloaded from the App Store are encrypted using the proprietary Digital Rights Management (DRM) system from Apple, FairPlay [16], and decrypted during execution. Hence, gaining access to the unencrypted apps requires not only access to proprietary Apple hardware, but also having it jailbroken. These constraints complicate the study of iOS apps and limit our understanding of their ecosystem evolution. Previous works on automated detection of programming language and framework identification on iOS were therefore limited to publicly available datasets like the Cocoapods dependency manager, or smaller real-world samples from the iOS App Store [3, 4].

We introduce a new methodology to extract information from protected iOS apps, without requiring decryption or Apple hardware, by leveraging the unencrypted parts of an application's iOS App Store Package (IPA) file. We then derive and categorize indicators from this information, to identify programming language use at different levels of granularity.

We applied our indicators to a large-scale analysis of real-world apps from the App Store, covering a dataset of 84,432 distinct apps collected over a five-years period from January 2020 to January 2025. These apps were derived from monthly snapshots of the two thousand most popular free apps in a major European country. We observed significant shifts in the iOS development ecosystem. While Objective-C is used in every app of all our monthly samples, languages like Swift or C++ have now also reached almost full saturation. We further observed that modern iOS apps use a mix of multiple languages, and identified the most frequently occurring language combinations. Additionally, newer frameworks and languages such as Flutter, React Native and Kotlin are emerging. With a presence of Flutter growing from less than 1% in 2020 to approximately 12% in 2025. This highlights an increase in language diversity of apps within the App Store.

Our contributions can be summarized as follows:

- We demonstrate that FairPlay DRM only covers a limited amount of information, and that the unprotected information can be used to identify meaningful data from binaries.
- We provide a new tool working on encrypted apps to extract such information available in a computationally efficient manner. [10]
- We performed a large-scale measurement on real-world apps from the App Store, covering 84,432 distinct apps from the last five years that stem from monthly snapshots of the two thousand most popular apps in a major European country.
- We provide the first dataset of such measurements for iOS to support comparison and further research. [10]

2 Background

The iOS ecosystem is characterized by proprietary file formats and languages. Previous research on language use in the iOS environment has predominantly focused on dependencies publicly available through the CocoaPods dependency manager or open-source applications [4], that might introduce a bias against commercial applications often relying on proprietary libraries and frameworks.

2.1 IPA Files

On iOS, applications are downloaded through the App Store as IPA files. Acting as the Apple proprietary counterpart of APK for Android, IPA files are ZIP or LZFSE archives containing the application code, resources, and metadata. The IPA internal structure adheres to a well-established hierarchy. A *Payload* directory contains the app bundle, integrating all compiled executables, assets, and resource files. The `Frameworks` folder contains frameworks that the application can use. Each framework is a packaged collection of code and resources that provides specific functionalities to the app, allowing modular development and code reuse.

2.2 Mach Object (Mach-O) Files

The actual executable files and loadable libraries use the Mach-O file format and can be found inside the application bundle of IPA archives. The file header provides metadata such as the file type (executable, dynamic library, or object file), target CPU architecture, and the number of load commands embedded within the file. Load commands instruct the loader how to load segment mappings, dynamic libraries, and where to link symbols. Segments delineate contiguous memory regions characterized by specific access permissions, while sections provide granular subdivisions within segments.

Following segment mapping, the loader processes the command `LC_ENCRYPTION_INFO_64` which contains the offset and size of the encrypted memory region. The mechanism relies on the Apple FairPlay DRM system, ensuring that protected content remains executable solely on authorized devices. Decryption occurs post-segment mapping but pre-execution, leveraging device-specific keys managed by Apple secure infrastructure. This means that the data is always encrypted at rest, and no official mechanism exists to decrypt a Mach-O file to yield a valid, but unencrypted, Mach-O file.

3 Dataset and Data Extraction

In this section, we present our raw dataset and introduce our findings on not DRM protected areas of apps in the iOS App Store. The analysis is based on monthly historical data starting from January 2020 representing a total of 84,432 distinct applications.

3.1 App Store Dataset Collection & Analysis

Assembling historical iOS app snapshots presents unique difficulties: Apple does not provide public archives of (past) App Store rankings, and geo-restrictions limit data collection. Our dataset originates from another project, which continuously collects iOS applications from the App Store. Due to the purpose of that project, our dataset excludes Apple's game categories to reflect business apps of a major European country. Our dataset therefore represents a rare continuous record of 2,000 free iOS applications per month, extending back to January 2020. The apps were chosen by enumerating the most popular applications across the iOS App Store categories [6]. Apple determines which are the most popular free apps.

This dataset consisted of 11 terabytes of compressed IPA files, comprising a total of 84,432 unique entries. From here, we extracted the metadata described in Section 4 as JSON files to perform our in-depth analysis and store our results. The final measurements were performed on a Dell PowerEdge R6625 with two AMD EPYC 9374F 32-Core Processors.

3.2 Mach-O Binaries and Encrypted Sections

The `LC_ENCRYPTION_INFO` load command that implements the DRM encryption only protects parts of a Mach-O binaries by specifying a memory range that will be decrypted. This memory range is simply specified via a start offset and a length, which contains no inherent information about what kind of data is covered by it. To investigate this we used our overall set of Mach-O binaries to determine the most frequently occurring sections and analyze if the sections are covered by the encryption range. With that information, we found that sections are always either fully encrypted or not encrypted, but never partially encrypted.

The `__text` section in the `__TEXT` segment was found in all 54,499 Mach-O binaries. It contains the compiled machine code of the Mach-O binary and its content is covered by the encryption in all cases. Other sections that are typically encrypted are for example the Objective-C specific sections `__objc_classname` and `__objc_methname`, containing class and method names. However, not all potentially usable information is encrypted: We observed the `__oslogstring` section in the `__TEXT` segment occurring 3,764 times, but never encrypted.

The FairPlay DRM system only encrypts specific internal sections of Mach-O files, such as the `.text` section, which contains compiled code. Files and several Mach-O sections containing metadata remain unprotected besides the overarching integrity check. Overall, DRM protects only a small fraction of the data contained in an iOS app.

4 Identification of Residual Data

In this section, we present our methodology to identify languages and Third Party Library (TPL) within applications. We first present the available metadata that can be extracted from IPA files, without decrypting sections protected by the FairPlay DRM, before discussing our chosen language indicators.

4.1 Available Metadata

IPA files are secured by code signing to ensure integrity but do not all employ encryption for confidentiality. Notably, all files within an IPA, except for the Mach-O binaries, are unaffected by the FairPlay DRM. Within Mach-O files, the Load Commands remain unencrypted by design, providing access to valuable metadata.

4.1.1 File Meta-Data. The file name, size and CRC checksum are available without extraction, which allows basic checks for file names, file suffixes and duplicates without needing to extract the archive file. ZIP files also support partial extraction, which allows us to only extract the files we need into memory on demand. This keeps disk usage low, and saves compute time by skipping unnecessary decompression of irrelevant files.

4.1.2 File Contents. The content of files can provide information about specific languages. We can extract unique identifiers for certain languages such as configuration files, unique resources, or directly the program code (e.g. code contained in `.js` files). As an example, apps that are developed with the cross-platform app development framework Ionic [2] typically contain a `config.xml` that contains the string `cordova`. Another interesting file is `Info.plist`, which provides us with insights about the app permissions, by identifying all purpose strings, which typically end with `UsageDescription`.

4.1.3 Mach-O Entitlement Information. Entitlement information is stored separately for each Mach-O file within the code signature. Precisely, it resides in the `LC_CODE_SIGNATURE` load command and can be extracted by decoding the contained data structure.

4.1.4 Mach-O Section Names and Sizes. As recalled in Section 2.2, the `LC_SEGMENT_64` Load Command specifies one segment from the file that should be loaded into memory. A segment consists of multiple sections, each with their own name, size and address at which they will be loaded. This provides us with the information about the section names, which can be used for language fingerprinting. The section sizes for specific known sections can then also be used to infer metadata without having to decrypt them.

4.1.5 Mach-O Encryption Information. The `LC_ENCRYPTION_INFO_64` Load Command specifies the area of memory that needs to be decrypted by the kernel. We extract this information, which we later match with the section information to determine which sections are commonly encrypted or left readable, as this is not publicly documented.

4.1.6 Mach-O Symbols. The `LC_SYMTAB` and `LC_DYSYMTAB` Load Commands provide the unencrypted symbol table for linking purpose. This allows us to extract all symbol names, and determine whether it is an import or export, and to which library the symbol is linked. Doing so, we can identify inter-component interaction, as well as language interactions, within the app.

4.2 Language Indicator Categories

We now use this data to detect language usage. We distinguish between multiple levels of hierarchical indicators: IPA Level, Mach-O Level and Symbol Level. Each indicator can only show the presence of the language on its own level and levels above. For example, a file in the IPA might imply that the iOS App overall uses a certain language somewhere, but with no information which part of the code uses it. A Mach-O Level indicator shows the specific file, and thus also that the app uses the language, but cannot infer which part of the code inside this Mach-O relates to this language. At the bottom are function symbol indicators which allow assigning a language to a specific function, and thus a specific code region.

Indicators can also be boolean or scalar. For example, some section indicators can only be used to show the presence of the language, while others allow deriving more information e.g. the number of Objective-C classes defined inside the binary.

4.2.1 Name Mangling Indicator (Symbol Level). Symbol indicators assign a language to a specific symbol, leveraging the name mangling process used by many languages. Our measurement of this splits the symbols into three categories: Exports, Imports and Internals. While the presence of internal symbols cannot be relied on, the imports and exports symbols must be present if the app makes use of dynamic linking.

4.2.2 Library Indicator (Mach-O Level). Many languages have runtime libraries or standard libraries that serve as clear indicators whether a binary makes use of this language in some capacity. Examples for this include the Objective-C runtime library `/usr/lib/libobjc.A.dylib` or the C++ standard library `/usr/lib/libc++.1.dylib`.

4.2.3 Symbol Indicator (Mach-O Level). Mach-O files that use certain languages will often have symbols that imply that the file uses this specific language. These are similar but distinct from library indicators, and happen if, e.g., the runtime functions of the language are statically compiled into the binary. In general, we split symbols into three categories. Imported Symbols, Exported Symbols and internal symbols. Symbols which are imported, or that can be exported are specifically marked by the metadata needed to facilitate dynamic linking. Internal symbols are then simply all symbols which are neither exports nor imports. Mach-O binaries are special in contrast to executable such as PE or ELF because their imports typically explicitly specify which library they should be imported from. This allows us to group the imports of one Mach-O file into subgroups for each imported library. This implicitly constructs a dependency graph between the files inside one IPA.

4.2.4 Section Names and Sizes (Mach-O Level). Compilers use predictable segment names to store language-specific metadata. While the exact metadata is often not fully available because the sections are encrypted, it is still sometimes possible to use side-channel measurements to infer information. For example, if a certain section is known to contain a list of pointers to the encrypted data for each class define inside a Mach-O file, we can simply derive the number of classes via dividing the length of the section by the pointer size.

4.2.5 File Extension Indicator (IPA Level). Some languages ship files with clear extensions which allows straight-forward inference of

Table 1: Indicator Availability per Language. (✓) indicates the use of symbol prefixes instead of true mangling.

Language	Mangling	Symbols	Libraries	Files	Sections
Objective-C	(✓)		✓		✓
Swift	✓		✓		✓
C++	✓		✓		
C#		✓		✓	✓
Kotlin	✓	✓			
Flutter		✓			
JavaScript		✓	✓	✓	

language presence in an IPA file. Common examples for this include JavaScript and TypeScript which use the extensions `.js` and `.ts`. An uncommon example is the presence of `.dll` files in apps using C#. These files are PE files, the executable format normally used by the Windows operating system, which are loaded by the C# runtime.

4.3 Language Identification

In this subsection, we outline the various indicators available per programming languages in the iOS ecosystem for identification. Note that we do not try to detect the C language, as it is too low level to leave identifiable artifacts in the available metadata. A summary of available indicators can be found in Table 1.

► **Objective-C.** Objective-C has been a foundational language for iOS development since the start of the platform [1]. It integrates Object-Oriented Programming (OOP) with C, allowing developers to utilize an extensive runtime and a specific library ecosystem. To identify the presence of Objective-C, we use the following indicators.

- ▹ **Mangling:** Objective-C does not use conventional mangling, however, when available, all function symbols are either prefixed with `+[` or `-[`. Other common Objective-C symbols are prefixed by `_OBJC` or `__OBJC`, such as class or protocol symbols starting with `__OBJC_CLASS` and `__OBJC_PROTOCOL`, respectively.
- ▹ **Libraries:** As we said, the library ecosystem of Objective-C is really distinctive, as we can detect the presence of the runtime library `/usr/lib/libobjc.A.dylib`.
- ▹ **Sections:** Various sections start with the prefix `__objc_`. Most importantly, the section `__objc_classlist` is a list of pointers to all classes defined in the Mach-O file, while `__objc_classrefs` is a list of pointers to all classes referenced in the Mach-O (includes classes imported from other components). This allows estimated differentiation between how much a Mach-O file uses Objective-C and how much Objective-C code it contributes.

► **Swift.** Introduced by Apple in 2014, Swift is the primary programming language for iOS development, actively maintained and supported by Apple, making it the default choice for iOS app development [8]. It was designed to improve safety, performance, and developer productivity, offering features such as type safety, memory management, and modern syntax. As identifications for the Swift language, we can leverage the following metadata.

- ▹ **Mangling:** Swift uses the mangling prefix `_$s`.
- ▹ **Libraries:** `/usr/lib/swift/libswiftCore.dylib` is the runtime library, though we argue that any library residing in `/usr/lib/swift` is an indicator for Swift.
- ▹ **Sections:** Various sections starting with `__swift`. Of special importance is the section `__swift5_types`, which contains a list of relative pointers to all classes, structs, and enums defined in the Mach-O file and allows quantification.

► **C++.** C++ is utilized in iOS development for performance-critical components and integration with existing cross-platform C++ codebases. Alongside native C++, iOS developers can incorporate C++ modules with Objective-C code using Objective-C++, making it suitable for apps that require efficient computations or access to established C++ libraries. For identificators, C++ offers us the following.

- **Mangling:** Clang uses the prefix `__Z` for C++ symbols. Other mangling schemes used by other compilers are known as well.
- **Libraries:** The specific library `libc++.1.dylib`, can be used to spot C++ occurences.

► **C#.** The C# language is used in iOS development primarily through the Xamarin [11] and .NET MAUI frameworks, which enable cross-platform app development. Xamarin and .NET MAUI compile C# code into native iOS binaries, providing access to iOS APIs while allowing developers to share code across platforms. C# uses the mono runtime to execute Ahead-of-Time (AOT) compiled code. The IPA typically still contains `.dll` files that are used by the mono runtime for features like reflection or serialization. With Xamarin apps, we often see a `Xamarin.` prefix within these `.dll` files. With .NET MAUI, the prefix of the files is `Microsoft.MAUI`. In principle, also other .NET languages like F# and VB.NET can be used in combination with the above frameworks which produce the same patterns, that will be detected by these indicators. For brevity, we only refer to this .NET class as C#. This information can be used for indicators as follows.

- ▹ **Files:** The presence of `.dll` files inside the IPA archive is an indicator of C# code.
- ▹ **Sections:** The `il2cpp` section can be present in Mach-O binaries when C# code is converted into C++ code. This is done with the IL2CPP backend [14] which can be used, e.g., in the Unity game framework.

► **Kotlin.** Predominantly used for Android development, Kotlin has been extended to iOS through Kotlin Multiplatform [7]. This enables developers to share common code between Android and iOS, with platform-specific components developed with Kotlin Native.

- ▹ **Mangling:** Functions and classes can be identified with mangling symbols starting with `_kfun:` and `_kclass:` respectively.
- ▹ **Symbols:** The `ComposeAppKotlin` symbol can identify binaries resulting from Kotlin.

► **Flutter.** Flutter is an open-source framework developed by Google for building cross-platform apps [5]. For iOS, Flutter uses the Dart programming language.

- ▹ **Symbols:** Flutter AOT snapshots contain specific symbols like `_kDartIsolateSnapshotData`.

Table 2: Language Occurrence in All 54,499 Mach-O Binaries and in All 2,052 Apps from January 2025.

Language	No. of Binaries		No. of Apps	
Objective-C	49,221	(90.3%)	2,052	(100.0%)
JavaScript	N/A		2,029	(98.9%)
C++	30,326	(55.6%)	2,016	(98.2%)
Swift	29,628	(54.4%)	1,995	(97.2%)
Flutter	5,280	(9.7%)	247	(12.0%)
Kotlin	252	(0.5%)	182	(8.9%)
C#	34	(0.1%)	58	(2.8%)

- ▹ **Libraries:** Mach-O files using Flutter import the Flutter runtime library which is simply called `Flutter`.

► **JavaScript.** JavaScript is used in iOS apps either as an addition, for example displaying WebViews like policy pages, or as a cross-platform solution through frameworks like React Native. We summarize JavaScript, TypeScript and JavaScript bytecode under the term JavaScript for the sake of our comparison with other languages.

- ▹ **Files:** Specific JavaScript related files can be seen such as `.js`, and `.ts` files.
- ▹ **Libraries:** Specific libraries like JavaScriptCore or Hermes can be used for JavaScript identification.
- ▹ **Symbols:** The WebView functionality within the UIKit library is utilized only when the `UIWebView` symbol is present.

5 Empirical Analysis

5.1 Language Distribution

We now take a look at how programming languages are used across iOS apps, both in terms of binaries and entire applications, analyzing the most recent snapshot in our dataset from January 2025.

5.1.1 Language Identification on Mach-O Binary Level. We use the indicators, described in Section 4.3 to identify the programming languages present in a Mach-O binary. Among the whole set of 54,499 binaries, we observed that Objective-C is the dominant language, occurring in $\approx$ 90.3% of all binaries. The second most frequent language is C++ (55.6%), closely followed by Swift (54.4%). We were able to detect Flutter less frequently (9.7%). Kotlin and C# occur rarely compared to the overall number of binaries, but they are still relevant on the app level (see Section 5.1.2). The distribution is pictured in Table 2.

Furthermore, we observed that the most frequent language combination occurring (43.4%) is a combination of C++, Objective-C and Swift. Pure Objective-C Mach-O binaries occur second most frequently, with a total occurrence of 26.9%. Interestingly, in the third most common position (7.2%), we have binaries where none of our language indicators matches for any language. A manual inspection revealed that indeed many of them have a `__text` section in the `__TEXT` segment of size 0. The purpose of this is not clear to us. The remaining ones were all pure-C libraries like OpenSSL or ffmpeg, for which we have no detection markers, as explained in Section 4.3.

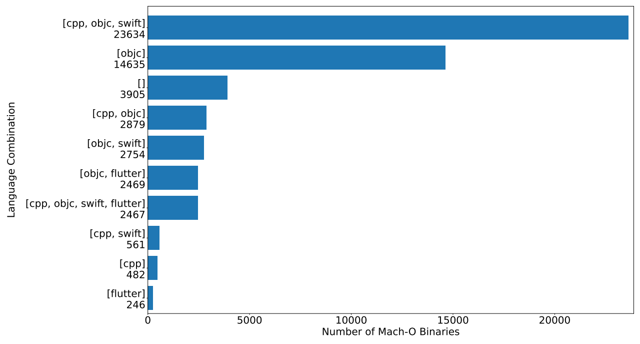

Figure 1: Most Common Language Combinations Detected in Mach-O Binaries

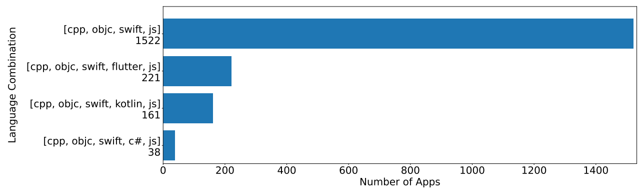

Figure 2: Most Common Language Combinations Detected in Apps

Figure 1 shows the most common combinations of languages within Mach-O binaries that occurred more than 100 times within our data set.

5.1.2 Language Identification on App Level. We observed that the dominant languages occurring on app level are Objective-C, occurring in all 2,052 apps, as well as JavaScript, C++, and Swift, each occurring in more than 97% of the app sample. The languages Flutter, Kotlin, and C# occur significantly less with an occurrence between 2% and 12%. Table 2 shows the occurrence of a single language among all 2052 apps.

We observe this trend also concerning language combinations as shown in Figure 2. The most frequent language combination on app level is a combination of C++, Objective-C, Swift, and JavaScript with an occurrence of 74.2%. The less frequent combinations are composed of Objective-C, Swift and JavaScript with either Flutter (10.8%), Kotlin (7.8%) or C# (1.9%). Other combinations have less than 1% occurrence.

5.2 Historical Data

5.2.1 Mach-O Binaries in Apps and their Language Distribution. We investigated the distribution of the number of Mach-O binaries per app and observed an increase of the median from 9 in January 2020 to 15 in January 2025. The interquartile range broadened from [1, 23] to [6, 38] during that time, showing an increase in complexity concerning the number of binaries in IPA files.

We investigated the occurring languages of Mach-O binaries over time. The most frequently occurring language, that is used in almost all Mach-O binaries is Objective-C. There is a slight downward trend

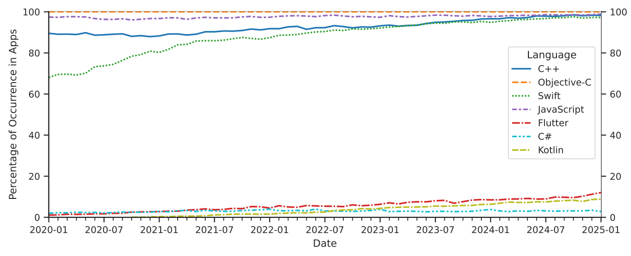

Figure 3: Monthly Percentage of Applications Containing the Corresponding Languages.

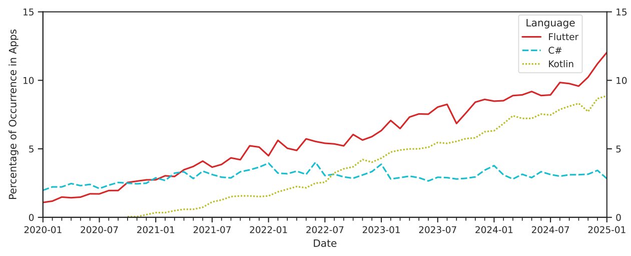

Figure 4: Monthly Distribution of the Less Common Languages, Occurring in the App Sample.

from 99.4% in January 2020 to 90.3% in January 2025, indicating that the number of Mach-O binaries with none of our Objective-C related indicators is increasing. Swift and C++ have a comparable percentage ranging between 40% and 60%. We observed a clear upward trend for Flutter from 0.6% to 9.7%.

5.2.2 Language Distribution on App Level. We discuss the observed languages in apps between 2020 and 2025, as shown in Figure 3. The dominant languages Objective-C and JavaScript are relatively stable in their usage. Objective-C is used in every app of all samples since January 2020. JavaScript is the second most frequently occurring language during the whole time span behaving relatively constant, only slowly increasing from 97.4% to 98.9%. C++ increases from 89.5% to 98.2%. We observed the most significant increase for Swift, increasing from 68.0% to 97.2%.

We observed an increase of the less established languages Flutter and Kotlin. Flutter increased from ≈ 1.1% to 12.0% occurrence. Kotlin first occurred in October 2020 and increased to 8.9% in January 2025. The percentage of apps using C# fluctuated between ≈ 2.0% to 4.0% over the observed time span. Figure 4 shows the detailed measurements of the less dominant languages in that time interval.

5.2.3 Distribution of JavaScript based Cross-Platform App Development Frameworks. Over the period from January 2020 to January 2025, there is a clear upward trend in the usage of JavaScript files in apps, increasing from ≈75% to over 90%. The presence of React Native also shows significant growth, doubling from 6% to over 12% in the same time frame. Ionic usage remains relatively stable with minor fluctuations, generally staying between 3% and 4%. NativeScript maintains minimal and consistent usage throughout the years, remaining below 0.5%. The combined usage of these frameworks does not account for all the apps with JS files. This suggests that a significant portion of JS files in apps are utilized outside these frameworks, see Figure 5.

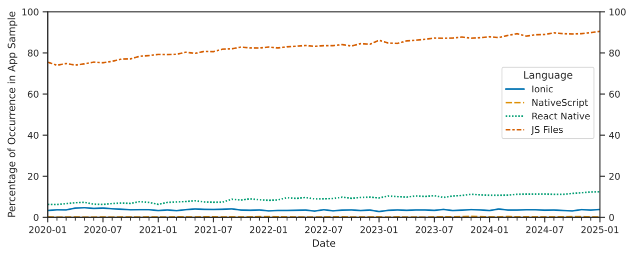

Figure 5: Percentage of All Apps Using JS Cross-Platform Frameworks or JS Files Over a Five-Year Period.

6 Related Work

Previous studies have used large-scale crawling systems, such as PlayDrone [15] or DroidSearch [13], to analyze free Android apps on Google Play.

In the iOS ecosystem, several studies have focused on analyzing mobile apps and their development practices. For third-party apps, CRiOS [12] analyzed more than 40,000 iOS apps, identifying extensive use of third-party libraries and security concerns with SSL/TLS endpoints, but it pre-dates the adoption of Swift. LibKit [3] proposed a tool for detecting the name and version of third-party libraries in iOS apps, leveraging CocoaPods to create fingerprints of library versions. We directly improve upon previous work, examining programming language trends across more than 25,000 libraries [4] found on CocoaPods. Their study could only compare Swift and Objective-C, for which CocoaPods is the official package manager. Our work can detect usage of other languages such as Flutter or Kotlin in real-world apps.

Kollnig et al. [9] conducted a large-scale comparative study of privacy practices in iOS and Android apps, analyzing 24,000 apps across both ecosystems. While their work focused on privacy concerns, it also underscores the challenges of analyzing iOS apps due to their encrypted nature and the absence of publicly available tools for analysis.

7 Conclusion

As iOS apps show a clear trend towards more complexity in terms of shipped binaries and language usage, there is a now a clearly demonstrable need for static analysis tools to support inter-binary, cross-language analyses. While it is clear that cross-language analysis is important, we determined and quantified the languages in the iOS ecosystem the research should focus on. Besides the established and still ubiquitous language Objective-C, the focus should also lie on its language interaction with Swift, C++ and JavaScript, as well as Flutter as new emerging language. However, it is reasonable to assume that certain cross-language interactions will occur more frequently than others. Quantifying these language interactions represents a desirable future work direction, which requires a further analysis of our processed dataset.

Our work also showed that protected IPA files contain rich information that can be used for practical purposes. While this is important for further analyses, as it allows side-stepping the tedious decryption process for many research questions, this is also important information for app developers, as the details of what exactly is protected through Fair Play's encryption are not documented by Apple and may not match their expectations.

We will publish the analysis tool and the dataset of measurement. This allows other research groups, which are possibly not deeply familiar with the iOS ecosystem nor have specialized hardware setups, to contribute to the state of knowledge for a still opaque ecosystem.

Acknowledgments

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] [n. d.]. *About Objective-C.* https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

[2] Ionic Framework [n. d.]. *Ionic Framework - The Cross-Platform App Development Leader.* Ionic Framework. https://ionicframework.com/

[3] Daniel Domínguez-Álvarez, Alejandro de la Cruz, Alessandra Gorla, and Juan Caballero. 2023. LibKit: Detecting Third-Party Libraries in iOS Apps. In *ESEC/SIGSOFT FSE.* ACM, 1407–1418.

[4] Daniel Domínguez-Álvarez, Alessandra Gorla, and Juan Caballero. 2022. On the Usage of Programming Languages in the iOS Ecosystem. In *SCAM.* IEEE, 176–180.

[5] Google. 2017. Flutter. https://flutter.dev/

[6] Apple Inc. [n. d.]. *Categories and Discoverability - App Store.* Apple Developer. https://developer.apple.com/app-store/categories/

[7] JetBrains. 2017. Kotlin Multiplatform. https://kotlinlang.org/docs/multiplatform.html

[8] JetBrains. 2021. Swift and Objective-C. https://www.jetbrains.com/lp/devecosystem-2021/swift-objc/

[9] Konrad Kollnig, Anastasia Shuba, Reuben Binns, Max Van Kleek, and Nigel Shadbolt. 2022. Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps. *Proc. Priv. Enhancing Technol.* 2022, 2 (2022), 6–24.

[10] Florian Magin, Fabian Scherf, Martin Renze, Cléo Fischer, and Gwendal Patat. 2025. iOS App Measurement Dataset and Code for "Measuring while Playing Fair" Paper. doi:10.6084/m9.figshare.28795532

[11] Microsoft. 2016. Xamarin. https://dotnet.microsoft.com/en-us/apps/xamarin

[12] Damilola Orikogbo, Matthias Büchler, and Manuel Egele. 2016. CRiOS: Toward Large-Scale iOS Application Analysis. In *SPSM@CCS.* ACM, 33–42.

[13] Siegfried Rasthofer, Steven Arzt, Max Kolhagen, Brian Pfretzschner, Stephan Huber, Eric Bodden, and Philipp Richter. 2015. Droidsearch: A tool for scaling android app triage to real-world app stores. In *2015 Science and Information Conference (SAI).* IEEE, 247–256.

[14] Unity Technologies. [n. d.]. *Unity - Manual: IL2CPP Overview.* https://docs.unity3d.com/6000.0/Documentation/Manual/scripting-backends-il2cpp.html

[15] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *SIGMETRICS.* ACM, 221–233.

[16] Tielei Wang, Yeongjin Jang, Yizheng Chen, Simon P. Chung, Billy Lau, and Wenke Lee. 2014. On the Feasibility of Large-Scale Infections of iOS Devices. In *USENIX Security Symposium.* USENIX Association, 79–93.