

# OCScraper: Automated Analysis of the Fingerprintability of the iOS API

Gerald Palfinger<sup>1,2</sup> 

<sup>1</sup>A-SIT Secure Information Technology Center Austria, Seidlgasse 22 / Top 9, 1030 Vienna, Austria

<sup>2</sup>Institute of Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria


**Keywords:** Fingerprinting, Smartphones, Apple iOS, Automatic Detection.

**Abstract:** Tracking has allowed application providers to offer the vast majority of their applications for free as it allows them to target advertising. However, tracking has proven to be an invasion of user privacy. To counter this, operating system vendors have removed access to unique identifiers in their APIs. Nevertheless, applications can still combine other non-unique data from the device to create a unique fingerprint. Until now, it has not been well understood what kind of information is available to do so on iOS. This paper addresses this gap by introducing the OCScraper framework, a tool for automatically discovering fingerprintable information sources on iOS devices. OCScraper does this by systematically crawling the API of the operating system. In the process, it creates objects on which methods are called and properties are queried. In our evaluation, we show that OCScraper can successfully invoke a large number of methods and retrieve the majority of parameters. We discover hundreds of robust information sources that provide distinct bits of information which can be used to create a cross-application fingerprint.

## 1 INTRODUCTION

Extensive tracking of users across different mobile applications is common on both Android and iOS (Kollnig et al., 2022a). This tracking is used by applications and their advertisement providers to learn more about their users and serve better targeted advertisements. With over 90% of applications being offered for free, the leading application monetization method worldwide are such in-application advertisements<sup>1</sup>. While this allows application vendors to offer their applications for free to the users, the tracking used to serve relevant advertisements is highly detrimental to the users' privacy (Shklovski et al., 2014). Furthermore, it may also violate the users' data protection rights, such as those mandated by the General Data Protection Regulation (GDPR) of the EU. In order to combat this, Apple has taken measures in recent versions of iOS with the intention of limiting the tracking conducted by third-party applications and their tracking libraries. Developers now have to request user consent using the App Tracking Trans-

parency framework<sup>2</sup>. Additionally, they have to declare what types of information they use to facilitate tracking. However, while these changes technically prevent the collection of the unique advertising identifier if no consent has been given, a recent study by Kollnig et al. (2022b) has shown that a large number of applications still collect or even start collecting other data from the operating system which can be used to create a fingerprint of the device. With the restriction on unique identifiers the technique of fingerprinting devices and users thus is becoming more relevant. For instance, in browsers where there are no unique advertising identifiers that can be used to identify users, fingerprinting is already prevalent (Iqbal et al., 2021). In essence, browser fingerprinting aims to collect as much unique data about the environment as possible to distinguish users. Kurtz et al. (2016) have shown that it is possible to apply this concept to iOS devices by using hand-picked information sources from the API to re-identify users. However, to date there exists no systematic approach to detect such information sources on iOS. To fill this gap, we present OCScraper, a framework for automatically detecting fingerprintable information sources on iOS.

<sup>a</sup>  <https://orcid.org/0000-0001-6633-858X>

<sup>1</sup> <https://www.statista.com/topics/983/mobile-app-monetization/>

<sup>2</sup> <https://developer.apple.com/documentation/apptestingtransparency>

By applying it in different case studies, we have identified hundreds of sources which provide information suitable for fingerprinting. In addition, our approach was designed to show that the sources found are robust across device restarts and reinstallations, and can be used for cross-application tracking.

## 1.1 Approach

In order to collect data from the operating system API and in turn identify robust, fingerprintable information sources suitable for tracking across application vendors, we take the following steps:

1. We create a smartphone application to systematically collect data that are available to applications via the API. Making use of reflection, the application instantiates all classes on the device. For each of the created objects, it invokes all instance methods of the class and stores the return value. Furthermore, it retrieves every property of the instantiated classes. Finally, using the class descriptor, it calls each class method.
2. Using the smartphone application, we collect the data from multiple devices. To simulate two different application vendors collecting the data, we run the smartphone application twice, using different developer profiles and bundle identifiers. Afterwards, the framework prepares the gathered data for cross-device analysis by cleaning and formatting it.
3. Finally, the data are analysed to detect fingerprintable information sources in the API. First, data of the same method or property which differ between the two runs on the same device are discarded, as these are not robust across different application vendors. Afterwards, the remaining data are compared across devices. Data sources which differ between the devices are considered fingerprintable.

## 1.2 Contributions

The contributions of this paper are as follows:

1. We have created a framework called OCScraper which systematically iterates through the iOS API and automatically creates objects, reads their properties, invokes instance and class methods and stores the received return values.
2. We evaluate the framework in terms of its coverage of the operating systems API. We show that the framework covers a large number of the methods and properties.

3. Using OCScraper, we dump the information from the API on different devices. By applying our automatic analysis framework, we evaluate which information sources can be used to fingerprint a device. Afterwards, although it is not necessary to use the discovered information sources for fingerprinting, the identified methods and properties are categorised to illustrate what kind of information sources are available.

## 2 BACKGROUND

In the following section, we will discuss the differences between the two main programming languages available on the iOS operating system that are relevant to our approach. Subsequently, we will discuss related work.

### 2.1 Swift vs. Objective-C

Swift, the newer option for developing applications on iOS, has very limited support for reflection. This support is provided by the Mirror API<sup>3</sup>. While it is possible to introspect values stored in objects, structs, and similar types, the API does not allow to create such objects or invoke methods on them. Therefore, Swift does not provide the tools necessary to create a comprehensive framework for querying data from the operating system. Objective-C, on the other hand, has a rich reflection API, which allows to query information about the API itself, dynamically create class objects, and invoke methods on those objects at runtime. Most of the reflection functionality is provided by the Objective-C runtime and its root class `NSObject`<sup>4</sup>. Due to this feature richness, we opted to use Objective-C to create our framework. Fortunately, both languages are interoperable, meaning it is possible to call Swift code from Objective-C and vice versa. To support this, the `@objc` qualifier has to be added to the Swift class. When this qualifier is present, the compiler will automatically generate the required headers.

### 2.2 Related Work

Mayer (2009) showed that it is possible to identify users by leveraging the “quirkiness” of browsers. This principle, now known as browser fingerprinting, was investigated by Eckersley (2010), who conducted a large-scale study showing that browser fingerprinting

<sup>3</sup><https://www.swift.org/blog/how-mirror-works/>

<sup>4</sup><https://developer.apple.com/documentation/objectivec>

is feasible. Since then, various features of modern browsers were exploited to track users. For instance, it was shown that extensions (Starov and Nikiforakis, 2017), or fonts (Fifield and Egelman, 2015) provide fingerprintable information. For a broader picture of the plethora of browser fingerprinting methods, we refer to a survey by Laperdrix et al. (2020).

Unlike browsers, smartphone fingerprinting has received less scrutiny, largely because unique identifiers were more readily available until recent versions of iOS and Android. In particular, most of the existing work has either relied on manually selecting a small number of information sources to fingerprint devices, or has focused on the Android operating system. Kurtz et al. (2016) manually selected 29 information sources from which they collected data. Their experiments showed that by combining them it was possible to re-identify devices with high accuracy. Similar to this study, Wu et al. (2016) found that uniquely identifying a device using a combination of 38 non-unique identifiers is also possible on Android. To re-identify users they used and compared different algorithms.

In contrast to the previous two studies, where a set of information sources was chosen by the authors, Torres and Jonker (2018) looked at the types of information used by tracking libraries on Android. To find these libraries, they used the sources in (Wu et al., 2016) and extended it by extracting the sources used by two known fingerprinting libraries. Based on these sources, they used static analysis to find six similar libraries.

As a more systematic approach, Palfinger and Prünster (2020) designed a framework to collect data from the Android API. The framework obtains data from fields, methods, and the Android-specific content providers. They showed that a large number of the detected sources provide fingerprintable information. However, the framework is only applicable to the Android operating system. As a result, the question of which information sources are available for fingerprinting on iOS remained unanswered. In this paper we seek to answer this question by designing an automated framework adapted to the specificities of iOS. The following chapter describes the methodology.

### 3 METHODOLOGY

OCScraper systematically traverses the iOS API to retrieve information from the operating system. The framework is divided into three main components, the backend, the smartphone application, and the analy-

sis component. Essentially, the smartphone application retrieves information about the API via reflection, automatically calling methods and reading properties. However, due to the low-level nature of Objective-C, failed invocations can lead to the termination of the smartphone application. To alleviate this issue, we designed a backend application which restarts the smartphone application if necessary. In addition, it also retrieves some specific information in advance that is not available during runtime. Finally, the backend transfers the collected data from the smartphone application for analysis. Once the collection process is complete, the analysis component sanitises and structures the collected data and analyses it across devices to identify fingerprintable information sources.

#### 3.1 Backend

**Parser.** Even though Objective-C provides a very powerful reflection API, as outlined earlier, it lacks one capability to successfully invoke some of the methods which require parameters. While it is possible to get a list of the available methods of a class, their names and, unlike Android, even the names of parameters, it is not possible to get the exact type of objects. Therefore, the framework cannot detect what type of object it needs to create in order to successfully invoke a method. To solve this problem, we gather the required information from the header files of the SDK. We created a small parser that collects the types of all method parameters. In particular, it traverses all the header files. When it encounters an interface or a protocol, it collects all the defined instance and class methods. For each of the methods it encounters, it parses all of the parameters, including their name, position, and type. This information is finally stored in a structured list. The list is then used by the smartphone application to create an object of the correct type. The parsing only needs to be done once for each new SDK release to account for new methods in the API.

**Control Application.** The data collection uses a tethered approach, meaning that the collection process is started and controlled by the control application on the desktop. This is necessary because, although Objective-C nowadays has automatic reference counting (ARC) for objects, it is not a fully managed language like Java or Kotlin on Android. For example, while some methods of the API throw an exception which can be caught by the smartphone application, others just terminate the application. Furthermore, instantiating private classes or calling private methods may cause the application to be terminated

by the operating system. Therefore, class instantiations, method invocations, or property retrievals made by the smartphone application could create a state from which it cannot recover itself. Thus, the control application on the desktop monitors the execution state of the smartphone application.

To control the device, the control application uses the open-source libimobiledevice library<sup>5</sup>. Before starting the collection process, the control application first installs the latest version of the smartphone application on the device. Afterwards, it starts monitoring the output of the system log of the attached device. Once this setup is complete, the control application launches the smartphone application. Using the library, the control application can detect when the application has been closed by the operating system. If this happens, the control application checks which class instantiation, retrieval of a property, or method invocation was the cause. It extracts this information from the monitored system log by searching for the last instantiated class, invoked method, or queried property. It then adds the offending class, method, or property to a blocklist. This updated blocklist is retrieved by the smartphone application the next time it is launched. Finally, the control component restarts the smartphone application and continues its monitoring of the output via the system log.

In addition to force closures, the control application detects whether or not the smartphone application is still reading properties and invoking methods correctly. If this is not the case for a predefined period of time, the control application restarts the smartphone application. This can happen when the smartphone application invokes a method which waits for something. For example, this method could be waiting for user input, network packets, or simply trying to acquire a lock. Therefore, before the application is restarted, the control application adds the last invoked method to the blocklist. This prevents the offending method from being called again and allows the smartphone application to proceed with the next method.

This process is repeated until all classes of the API have been instantiated and all their properties retrieved and methods called. Thereafter, the results of the invocations are collected from the monitored system log.

### 3.2 Smartphone Application

The data collection is performed using an application on the smartphone. It collects the data provided by the operating system API, which is available without requiring any user interaction or permission. In

<sup>5</sup><https://libimobiledevice.org/>

particular, the data collection application runs without any special entitlements<sup>6</sup> or permissions. For this purpose, methods are called and the return values are collected. The contents of the properties of the class are also retrieved and stored.

**Initialisation.** Once the application has been started by the control application, it fetches and parses the blocklists for classes, methods, and properties. In addition, it retrieves the name and index of the class at which to continue recording. The first time it runs, the blocklists are empty and it begins at the first class. After this initialisation procedure, it starts the actual recording process. First, it gets all the available classes. For each of the classes derived from `NSObject`, an object is allocated and initialised. This object is then used to read all the properties and then invoke all the instance methods on it.

**Property Retrieval.** To retrieve the properties of the object, a list of all properties is first retrieved. For each property descriptor in the list, the name of the property is fetched. To read the property by its name, the method `valueForKey` is called on the class object. The return value of this method is the value of the property. This value is then stored for the analysis. To store it in a readable form, the type of the property is required. This type is determined using `property_getAttributes` and then converted into the form required for the output. The value is finally written to the system log.

**Method Invocation.** Once all the properties have been retrieved, the methods of the class are called. For this purpose, similar to the properties, the list of methods of the class is retrieved. This list contains method descriptors. These allow to subsequently retrieve the selector of the method which is in turn used to get the method signature object. The method signature is then used to create an `NSInvocation` object. Furthermore, if the method takes parameters, these are created first. To create these objects, the type of the parameter must first be determined. This is done by retrieving the type encoding of each parameter from the previously created method signature. For instance, the type encoding for the primitive type integer is `i`, for an integer array with three integers `[3i]`, and for an Objective-C class object `@`. While most arguments can be created using this description, we need more information to successfully create an object type. For this purpose we use the information previously parsed

<sup>6</sup><https://developer.apple.com/documentation/bundleresources/entitlements>



by the parser component. Thus, when the application encounters a class object, it queries the pre-parsed information using the method signature and the parameter index. As the pre-parsed information includes the object type, the smartphone application can proceed to create the correct object instance for the argument. Each of the created arguments is finally added to the `NSInvocation`.

Before the `NSInvocation` is used to invoke the method, the target must be set. Depending on the type of method, a different target has to be specified. In Objective-C, there are two types of methods — instance methods and class methods. Instance methods, as their name suggests, require an instance object of the class to be invoked on. In contrast, class methods can be invoked on the class descriptor directly and therefore do not require an instance object. Thus, depending on the type of the method, we set either the instance object or the class itself as the target of the invocation. Finally, the `NSInvocation` is used to invoke the method. Afterwards, the return value is written to the system log. As with the properties, the type of the return value is required. The type is retrieved using the `method_getReturnType` method and then converted to the required form. This process is repeated for each method. Once all of the methods of the current class have been invoked, the framework continues with the next class.

The smartphone application does not invoke methods that return nothing, i.e. have a void return type. It also skips certain methods which are detrimental to the collection process. These include methods that deallocate the current object. In addition, it also skips methods which would cause undefined behaviour by re-allocating the current class. Finally, it also skips methods that are part of the reflection API, or methods inhibiting the collection process, for example, the method `lock`.

### 3.3 Data Analysis

**Preprocessing.** While the data is collected by the smartphone application, all results are written to the system log. The control application stores this output unchanged. In the stored log, all output from the smartphone application is retained, including, for example, output from called methods. Therefore, the collected data has to be cleaned before the analysis. This is done by our analysis application. It collects the return values from the system log and discards superfluous output. Additionally, it converts the results into a predefined format. These steps allow the results to be compared automatically.

**Data Sanitisation.** Similar to the approach of Palfinger and Prünster (2020), the data collection process is performed twice on each smartphone. In between, the smartphone application is removed and the device is restarted. After removal of the application, it is reinstalled with a different signing certificate and bundle identifier. The second collection phase is then started. This elaborate process is performed to simulate the collection of data by applications from different developers. This eliminates those sources of information that are different between two distinct applications and therefore not suitable for creating a cross-application fingerprint. For instance, this eliminates false positives such as the application's installation directory, which contains a random value that is different for each application. Additionally, developer-specific identifiers, such as the property `UIDevice.identifierForVendor`<sup>7</sup> are also excluded using this approach. Finally, this measure also eliminates all temporary values that could change due to an application restart or a device restart. All remaining values, i.e. those that do not differ between the two runs on the same device, are then stored. These are collected for all devices examined and compared in the next step, the analysis.

**Analysis.** The collected, cleaned and ordered values of a device are compared with all other examined devices using the analysis framework from Palfinger and Prünster (2020). Methods that could only be invoked on one device, or properties which could only be retrieved on one device, are removed as no comparison value is available. All remaining values, i.e. those that could be retrieved on at least two devices, are then automatically compared. All values that are identical on all analysed devices are removed. All values that differ on at least one device are retained and their source is flagged as potentially fingerprintable. These are then manually reviewed and categorised for illustrative purposes the next chapter.

## 4 EVALUATION

In the following section, we show how many properties can be retrieved and how many methods can be successfully invoked using our approach. Afterwards, we illustrate our evaluation design. Finally, we show the results we obtained using OCScraper in our case studies.

<sup>7</sup><https://developer.apple.com/documentation/uikit/uidevice/1620059-identifierforvendor>

## 4.1 API Coverage Analysis

Since our framework is fully automated, the number of successfully invoked methods is important to evaluate its effectiveness. To investigate the coverage we used an iPhone 11 running iOS 16.3.1 (latest at time of writing).

**Methods.** In total, the device has 14,381 classes which can be detected via reflection. These classes contain 263,104 instance and class methods. Some of these methods had to be skipped. In particular, 9,363 were omitted as these methods would deallocate the current class object, making it impossible to invoke any further methods on that object. 124 methods were not invoked because they would lock a synchronisation primitive or sleep the current thread, which would halt the collection process. Another 365 methods were ignored as they would just cast the current object to a different type. Additionally, 13,841 methods were not called because they would re-allocate or re-initialise the current class, which would potentially lead to memory problems. 150 methods were skipped since they are part of the reflection API. 1,796 occurrences of the method `copyWithZone` were omitted as they are deprecated in modern Objective-C using automatic reference counting. Finally, another 79,351 methods were not called due to them providing no return value. This leaves a total of 158,114 methods that are potentially relevant and could return fingerprintable information. This therefore represents the base number of methods that OCScraper will attempt to call.

Of the 158,114 relevant methods, 437 could not be invoked because the corresponding class does not derive from `NSObject`. Therefore, these classes lack important reflection functionality. 6,982 methods were omitted because the associated class object could not be created, and have therefore been blocklisted. Reasons for this include missing parameters or superclasses that are not intended to be instantiated directly. A further 12,463 methods could not be invoked because an exception was thrown while the corresponding object was created. Additionally, 3,396 methods were skipped as the created class object was null. A total of 6,473 methods were skipped because they caused problems and were hence blocklisted by the controller, for example due to missing entitlements or memory issues. In addition, exceptions were thrown by 4,218 of the invoked methods. Finally, 14,919 methods failed to be invoked because arguments were not created successfully. This can occur, for example, if the class name was not known because the class was not part of the SDK header files. This results

in a total of 109,226 methods being invoked successfully, representing approximately 69% of the relevant methods.

**Properties.** A total of 74,166 class and instance properties were found using reflection. 7,168 of the instance properties could not be retrieved because the corresponding class object could not be created. Similar to the method invocations, this could be due to exceptions during object creation, because the class is blocklisted, because the instanced object is null, or if the class does not support reflection. Another 2,568 properties were omitted because an exception was thrown during retrieval of the property. Further 856 properties were blocklisted. Finally, for 2,489 properties, reading the value of the property was not possible. Thus, in total, 61,085 of the 74,166 properties were successfully retrieved, which corresponds more than 82% of the discovered properties.

## 4.2 Evaluation Design

To perform the experiments, our automation framework was executed on different iOS devices. The test devices were running iOS 16. The smartphone application collected data from properties and methods that did not require additional user consent when they were invoked or accessed. As no entitlements were requested by the application, protected methods would either throw an exception or terminate the smartphone application (which would then blocklist the method). If a method still triggered a user consent prompt (such as accessing certain local network details), the request was denied.

To demonstrate the effectiveness of our framework, we conducted two different case studies. In the first study, we compared the results of devices of the same model. In particular, this allows us to detect differences resulting from user adjustments, such as changes to the device settings. In the second case study, we collected data from different device models, allowing us to identify differences between these models. To avoid reporting the same source twice, the methods and properties which have been reported in the first case study were not included in the second.

## 4.3 Findings

**Same-Model Study.** When analysing the results across devices of the same model, our framework found a total of 642 sources of information that differed between them. Specifically, these sources include 368 methods and 274 properties. The majority of the information sources are the result of user

Table 1: Information sources identified analysing devices of the same model. (M = Methods, P = Properties).

Category	# M	# P
Network Information (Unstable)	4	-
Boot Time (Unstable)	2	-
UI Size & Screen Resolution	114	139
Language & Locale	83	28
Keyboard Settings	54	33
Various Accessibility Settings	39	22
Date & Time Format	16	21
UI Colour & Style	7	4
Model Number	5	5
User Persona & other Identifier	6	4
Currency Settings	5	5
Number Format	4	3
Carrier Name & MMS Setting	5	2
Input Method	6	-
Removed System Apps	6	-
Various Settings	4	-
Device Enclosure Colour	1	3
App Information	-	3
Uncategorised	2	1
First Week Day	2	-
Personalised Advertising allowed	1	-
UI Effects	1	-
System Sound	1	-
User-defined Device Name	-	1
<b>Sum</b>	<b>368</b>	<b>274</b>

customisation of the device. The categorisation of the results is shown in Table 1. The results were divided into unstable (i.e. changing in the medium term) and stable fingerprintable information sources. The unstable information sources include network details, such as the gateway IP and domain name, and the boot time. This information is marked as unstable because it changes when connecting to a different network. Nevertheless, they can still provide some details, for example, about frequently used networks. Furthermore, the boot time is also counted as an unstable source, as it changes each time the device is restarted.

More relevant to the creation of a fingerprint are stable sources of information, which make up the vast majority of detected sources. These include the display size of elements selected by the user, the screen dimensions, and the size of the toolbar. In addition, information about the colour space and various other settings concerning the user interface can be retrieved. These include the style of the status bar and the use of blurring. The selected language and the system locale can also be accessed via various methods. The programming interface also permits fetching the time zone. In addition, information about the currency

Table 2: Information sources identified analysing devices of a different model. (M = Methods, P = Properties).

Category	# M	# P
OS Version	42	24
Model Information	27	20
UI Size & Screen Dimensions	6	25
UI Style	12	4
Keyboard Settings	12	4
Hardware Differences	9	6
Screen Resolution	2	7
Uncategorised	5	4
Model-Specific UIDs	4	1
Screen Corner Radius	1	3
Versions	1	3
Physical Memory Size	2	2
Biometrics Supported	2	2
Home Button Type	1	2
Audio Support	3	-
Audio Latency	2	-
Device Size	2	-
<b>Sum</b>	<b>133</b>	<b>107</b>

used and its format settings can also be accessed. Likewise, various format settings, such as date and time formats and number formats, can be retrieved. Many of the user-defined accessibility settings are also accessible to applications. These include, for example, the setting to minimise movements, certain exclusion areas in the user interface, the preference for scaled content, the disabling of sliders or the use of the device's dictation function. Additionally, the input method as well as the input application including its bundle identifier can be found. Depending on the input method used, the supported input languages also differ. The size of fonts used in the user interface are also retrievable. Furthermore, using some possibly private APIs, the framework found unique identifiers, such as the users' persona identifier. Additionally, it was possible to query whether or not the user allows personal advertising. The API also holds information about the network operator and whether the MMS service is enabled. Information about removed system applications included in iOS could also be found in the results. In addition, the device name defined by the user can be determined via a property. Finally, although we used the same version of iPhones in this case study, we found some differences in the model number and information about the colour of the device itself.

**Cross-Model Study.** In the cross-model study, our framework identified a total of 240 sources of information that allow us to infer differences between these device models. These sources consist of 133

methods and 107 properties. They allow for the detection of variations between different device models. Our categorisation of the detected sources is depicted in Table 2.

The largest number of sources report information pertaining the version of the operating system, including the version string and build numbers. This category is followed by sources that provide information about the device model. Similar to the previous study, we also found methods and properties that report differences in the size of user interface elements and the screen dimensions. Additionally, we detected sources which report variations in the style of the user interface. Furthermore, our framework found varying information about the keyboard. Variations in the hardware used, such as SoC generation, hardware model numbers, and hardware support for certain features, can be identified between the device models. The API also provides information about the different screen resolutions and sizes of the devices. Various other differences between the models can be queried, such as the type of biometrics supported, physical memory sizes, screen corner radius, or the type of home button. In addition, model-specific identifiers can be retrieved. Finally, differences in support for some audio features and different specified output latencies have been identified.

## 5 DISCUSSION

In our work, we identified a large number of information sources that can be used to fingerprint a device. As our search was automated, we were able to find a larger number of sources than have been previously published. We anticipate that the list of properties and methods that provide this information can be leveraged to improve the detection of fingerprinting behaviour in applications. In particular, we envisage that the detected sources can be utilised to perform static and dynamic code analysis. The vast number of sources should allow for better detection of such behaviour, especially when applications try to hide their intentions by using less obvious sources of information to construct a fingerprint.

Our framework is based on Objective-C as Swift has only very limited support for reflection. While most of the APIs are available for both languages, some of the most recent frameworks such as CryptoKit or SwiftUI are only available to Swift-based applications. As Swift does not have extensive enough support for reflection, a different approach has to be found to analyse these APIs. However, as new APIs are increasingly designed with privacy in mind (Ole-

jnik et al., 2017), we decided to first focus on the existing APIs.

Some of the sources found may be considered private and thus cannot be used in applications distributed through the official store. However, in contrast to Android<sup>8</sup>, this means that there are no technical measures in place on the smartphone to prevent applications from using these sources. Thus, the security of the users is only subject to proper vetting of the submitted applications by Apple and its tight control over the ecosystem. In particular, any use of these methods, either directly or via reflection, must be detected by their static and dynamic analyses. Furthermore, this tight control over the distribution channel may be challenged in the future, as new EU legislation<sup>9</sup> requires the provisioning of application stores to be opened up to third parties. In addition, applications that are not subject to the application store guidelines, such as Apple's own applications (Kollnig et al., 2022b) or corporate applications, have access to these sources.

## 6 CONCLUSIONS

This paper introduced OCScriber, a framework for the automatic discovery of fingerprintable information sources on the iOS operating system through the methods and properties of its API. It is able to automatically create objects of classes, instantiate parameters, invoke methods, and retrieve properties. Our evaluation has shown that it can successfully call a large proportion of the methods and query a majority of the properties. By analysing the collected data across different devices, it finds information sources that differ across them. Using the framework, we have identified hundreds of methods and properties that can be used to create a fingerprint which is robust between application and device restarts and is applicable across application vendors. The results improve the understanding of what kind of information is still available to applications on iOS that can be leveraged to create a fingerprint. Furthermore, we believe that the identified sources can be used in static and dynamic code analysis to enhance the detection of fingerprinting behaviour in applications.

<sup>8</sup><https://developer.android.com/guide/app-compatibility/restrictions-non-sdk-interfaces>

<sup>9</sup>[https://competition-policy.ec.europa.eu/dma\\_en](https://competition-policy.ec.europa.eu/dma_en)



## REFERENCES

- Eckersley, P. (2010). How Unique Is Your Web Browser? In *Privacy Enhancing Technologies – PET 2010*, volume 6205 of *LNCS*, pages 1–18. Springer.
- Fifield, D. and Egelman, S. (2015). Fingerprinting Web Users Through Font Metrics. In *Financial Cryptography – FC 2015*, volume 8975 of *LNCS*, pages 107–124. Springer.
- Iqbal, U., Englehardt, S., and Shafiq, Z. (2021). Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *IEEE Symposium on Security and Privacy – S&P 2021*, pages 1143–1161. IEEE.
- Kollnig, K., Shuba, A., Binns, R., Kleek, M. V., and Shadbolt, N. (2022a). Are iPhones Really Better for Privacy? A Comparative Study of iOS and Android Apps. *Proc. Priv. Enhancing Technol.*, 2022:6–24.
- Kollnig, K., Shuba, A., Kleek, M. V., Binns, R., and Shadbolt, N. (2022b). Goodbye Tracking? Impact of iOS App Tracking Transparency and Privacy Labels. In *Conference on Fairness, Accountability, and Transparency – FAccT 2022*, pages 508–520. ACM.
- Kurtz, A., Gascon, H., Becker, T., Rieck, K., and Freiling, F. C. (2016). Fingerprinting Mobile Devices Using Personalized Configurations. *PoPETs*, 2016:4–19.
- Laperdrix, P., Bielova, N., Baudry, B., and Avoine, G. (2020). Browser Fingerprinting: A Survey. *ACM Trans. Web*, 14:8:1–8:33.
- Mayer, J. R. (2009). Any person... a pamphleteer”: Internet anonymity in the age of web 2.0. *Undergraduate Senior Thesis, Princeton University*, 85.
- Olejnik, L., Englehardt, S., and Narayanan, A. (2017). Battery Status Not Included: Assessing Privacy in Web Standards. In *Workshop on Privacy Engineering – IWPE@S&P*, volume 1873 of *CEUR Workshop Proceedings*, pages 17–24. CEUR-WS.org.
- Palfinger, G. and Prünster, B. (2020). AndroPRINT: analysing the fingerprintability of the Android API. In *Availability, Reliability and Security – ARES 2020*, pages 94:1–94:10. ACM.
- Shklovski, I., Mainwaring, S. D., Skúladóttir, H. H., and Borgthorsson, H. (2014). Leakiness and creepiness in app space: perceptions of privacy and mobile app use. In *Conference on Human Factors in Computing Systems – CHI 2014*, pages 2347–2356. ACM.
- Starov, O. and Nikiforakis, N. (2017). XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *IEEE Symposium on Security and Privacy – S&P 2017*, pages 941–956. IEEE Computer Society.
- Torres, C. F. and Jonker, H. (2018). Investigating Fingerprinters and Fingerprinting-Alike Behaviour of Android Applications. In *European Symposium on Research in Computer Security – ESORICS 2018*, volume 11099 of *LNCS*, pages 60–80. Springer.
- Wu, W., Wu, J., Wang, Y., Ling, Z., and Yang, M. (2016). Efficient Fingerprinting-Based Android Device Identification With Zero-Permission Identifiers. *IEEE Access*, 4:8073–8083.