iHunter: Hunting Privacy Violations at Scale in the Software Supply Chain on iOS

Dexin Liu[1,2*], Yue Xiao[3*], Chaoqi Zhang[3], Kaitao Xie[2], Xiaolong Bai[2†], Shikun Zhang[1], Luyi Xing[3†]

[1]*National Engineering Research Center for Software Engineering, Peking University,*
{dxliu,zhangsk}@pku.edu.cn,
[2]*Alibaba Group, {easylyou.xkt,bxl1989}@gmail.com,*
[3]*Indiana University Bloomington, {xiaoyue,cz42,luyixing}@iu.edu*

Abstract

Privacy violations and compliance issues in mobile apps are serious concerns for users, developers, and regulators. With many off-the-shelf tools on Android, prior works extensively studied various privacy issues for Android apps. Privacy risks and compliance issues can be equally expected in iOS apps, but have been little studied. In particular, a prominent recent privacy concern was due to diverse third-party libraries widely integrated into mobile apps whose privacy practices are non-transparent. Such a critical supply chain problem, however, was never systematically studied for iOS apps, at least partially due to the lack of the necessary tools.

This paper presents the first large-scale study, based on our new taint analysis system named *iHunter*, to analyze privacy violations in the iOS software supply chain. *iHunter* performs static taint analysis on iOS SDKs to extract taint traces representing privacy data collection and leakage practices. It is characterized by an innovative iOS-oriented symbolic execution that tackles dynamic features of Objective-C and Swift and an NLP-powered generator for taint sources and taint rules. *iHunter* identified non-compliance in 2,585 SDKs (accounting for 40.4%) out of 6,401 iOS SDKs, signifying a substantial presence of SDKs that fail to adhere to compliance standards. We further found a high proportion (47.2% in 32,478) of popular iOS apps using these SDKs, with practical non-compliance risks violating Apple policies and major privacy laws. These results shed light on the pervasiveness and severity of privacy violations in iOS apps' supply chain. *iHunter* is thoroughly evaluated for its high effectiveness and efficiency. We are responsibly reporting the results to relevant stakeholders.

*The first two authors contributed equally to this work and are ordered alphabetically. Dexin Liu's contribution was made during his research internship at Alibaba Group. Yue Xiao was a Ph.D. student at Indiana University.

†Xiaolong Bai and Luyi Xing are corresponding authors.

1 Introduction

Privacy violations and compliance issues in mobile applications pose serious concerns for users, developers, and regulatory bodies. Prominent issues include those related to data collection transparency and fairness, the right to be forgotten, or various other privacy rights defined by legislation. Comprehensive analyses of these issues have been predominantly Android-focused, utilizing various available tools as demonstrated in prior studies [39, 40, 45, 67, 73, 81, 88]. Conversely, privacy research in iOS applications remains underexplored, despite its stature as a leading mobile platform and Apple's efforts to enforce privacy safeguards, such as mandatory privacy policies [2] and App Store privacy labels [6]. This research gap stems from Apple's closed ecosystem and the absence of off-the-shelf analysis tools. Existing literature has exposed considerable privacy risks within iOS applications [37, 48, 50, 63, 83, 84, 90], including recent findings on widespread non-adherence to privacy labels [84] and reports of extensive app suspensions by Apple for privacy violations in 2021 [4]. Notably, these studies primarily target app-level privacy violations, with less attention given to the influential role of third-party SDKs in perpetuating these issues—a critical aspect that merits deeper investigation in the realm of privacy research.

Modern programs, like mobile applications, heavily rely on third-party software development kits (SDKs) to ease development, reduce costs, and keep up with trends. While enjoying the convenience they bring, apps also have to bear the privacy risk incurred by the SDKs. [17, 47, 70, 74, 77, 80] have demonstrated that SDKs can introduce extra privacy problems to apps. Previous work [46, 47, 57, 67, 70, 71, 74, 77, 78, 79, 80] focused on analyzing and defending against privacy hazards in Android third-party SDKs, including a new kind of privacy violation called Cross Library Data Harvesting (XLDH in short) [80] among Android SDKs. However, no one has studied the problem in iOS SDKs.

Challenges. Several obstacles make it difficult to analyze privacy violations in iOS SDKs:

• **Lack of dataflow analysis tools.** Compared to apps, SDKs are unsuitable for dynamic analysis because they have no entries and cannot be dynamically executed like apps. Existing tools for iOS binaries, such as PiOS [50] and iRiS [48], are limited in their static analysis capabilities, particularly in terms of sophisticated data flow analysis like taint tracking, which is crucial for examining the transmission of privacy data. On the other hand, traditional static dataflow analysis tools [14, 33, 56, 58, 72] for binaries on other platforms cannot handle the semantics of Apple's unique Objective-C and Swift languages. As a result, a static dataflow analysis tool for iOS binaries, which can properly handle Objective-C and Swift semantics, is required to analyze iOS SDKs.

• **Unclear mapping between iOS system APIs and privacy-related data.** System APIs are fundamental sources of user privacy on devices. Yet, a comprehensive and precise list of iOS system APIs that yield privacy-sensitive data, as governed by privacy laws or regulatory frameworks, remains unavailable.

Methodology. We developed an automated static analysis system, *iHunter*, dedicated to performing taint analysis on iOS binaries with flow-sensitive, context-sensitive, and field-sensitive granularity by tackling obstacles imposed by Apple's unique languages. As the foundation of *iHunter*, an innovative iOS-oriented extension to symbolic execution is proposed to handle the dynamic features of Objective-C and Swift and perform taint propagation. Prior works [48, 49, 50, 82] manually defined a limited set of system APIs that return privacy-related data (e.g., device identifier, location, contacts, etc), in which coverage and accuracy cannot be guaranteed. Moreover, the invocation of system APIs, like string operations and array/dictionary manipulation, heavily affect taint propagation, which commonly requires a time-costing cross-binary analysis into the system libraries. To tackle these two challenges and ensure high coverage and efficiency, *iHunter* proposes a fully automatic method to extract sources of privacy data and rules of taint propagation from Apple's high-level system APIs without the need to perform cross-binary analysis. In particular, our system adopts natural language processing (NLP) on Apple's developer documents to automatically extract sources and taint rules from the description of APIs. With taint sources, sinks, and rules, *iHunter* generates taint traces that represent the privacy collection behaviors of SDKs. Our thorough evaluation shows that *iHunter* can extract taint traces effectively and efficiently.

Large-scale analysis and findings. Our approach involves comparing the taint traces, as identified by *iHunter*, against the privacy policies of iOS SDKs procured from their respective websites to scrutinize the adequacy of disclosure regarding their data practices. *iHunter* identified non-compliance in 2,585 SDKs (accounting for 40.4%) out of 6,401 iOS SDKs, signifying a substantial presence of SDKs that fail to adhere to compliance standards. Specifically, we categorized non-compliant SDKs as follows: 2,436 were marked as non-compliance type I, exhibiting data processing activities without any privacy policies; 58 were classified as non-compliance type II, having privacy policies that either neglect to mention or incorrectly represent their data processing activities; and 45 were recognized as non-compliance type III, these SDKs were engaged in data harvesting from other SDKs, violating the Terms of Use of the victim SDKs. We further examined the prevalence of these SDKs in real apps and assessed how these non-compliant SDKs affect the privacy compliance of the apps. We found that 15,336 (47.2%) apps have integrated at least one of the problematic SDKs. Due to non-transparent data practices and improper disclosures of the SDKs, those real apps come with practical risks of privacy noncompliance based on the Apple app store policy. Specifically, Apple app store [6] requires that app vendors should conspicuously disclose (using a privacy policy) all data practices (e.g., collection, sharing) occurring in the app, including those performed by third-party libraries. We characterize the findings with case studies, including the occurrence of XLDH [80], an emerging type of privacy violation previously reported on Android. This has been found in highly popular SDKs such as AppsFlyer [7]. The results of our first large-scale study indicate the prevalence and severity of privacy violations in the iOS apps' supply chain.

Contributions. We summarize key contributions as follows.

• We performed a systematic study on privacy violations in the iOS SDKs. Our results shed light on the prevalence, and seriousness of privacy violations in iOS SDKs.

• We introduced *iHunter*, a comprehensive system designed to systematically scrutinize privacy compliance among iOS SDKs and detect potential violations.

• We conducted a large-scale study of 6,401 iOS SDKs and discovered a high portion (40.4%) of privacy violation behaviors. Moreover, we provide detailed case studies of violations of privacy policies and cross-library data harvesting SDKs affecting real-world iOS apps. This sheds light on the prevalence and severity of privacy violations in the iOS apps' supply chain.

• We plan to release the source code of *iHunter* [36] to facilitate future research on iOS and related datasets in this study.

2 Background

2.1 Programming Languages in iOS

Objective-C [19], the fundamental programming language of iOS apps, offers features such as dynamic typing, messaging, and reflection, enhancing its usability and adaptability. Yet, its unique dynamic features and runtime mechanisms introduce complexities in taint tracking, especially concerning runtime method calls and dynamic types. Objective-C invokes methods by passing function names (selectors) and objects to a standard `objc_msgSend` function. This function dynamically

determines the method implementation at runtime, enhancing program flexibility. However, this dynamism, managed by the Objective-C runtime, complicates taint tracking and data flow analysis, as runtime behavior modifications can alter data flows. Existing literature [48, 50, 82, 84] has explored these challenges in the context of Objective-C binary analysis.

Swift is Apple's new programming language designed for iOS development. It is primarily a statically typed language, known for its emphasis on safety, performance, and user-friendliness. Swift primarily uses static dispatch for function calls, yet it can employ dynamic dispatch by specifically marking a function as 'dynamic.' This change allows the system to decide at runtime which overridden method to execute, rather than making this determination at compile time. In addition, the Swift compiler utilizes name-mangling [60] to generate unique identifiers for each function to be called. Name-mangling is a technique used to produce a distinctive identifier for an object based on its language-level name and type, which can be leveraged by the linker to reference the object. When performing a static taint analysis on Swift binaries, the dynamic dispatch mechanism and the name-mangling scheme used for function names can present significant obstacles in determining which functions in the Swift SDK are being called. This necessitates an in-depth analysis of the runtime behavior of Swift binaries to ascertain the object types and resolve function calls dynamically.

2.2 Taint Analysis and Challenges

Taint analysis can be defined as a graph reachability problem on the program's data flow graph (DFG in short). Given a binary, our goal is to detect the paths from source nodes N_{so}, where privacy data is collected, to sink nodes N_{si}, where privacy data is leaked, within the data flow graph G_d. Commonly, static taint analysis can be summaries with steps as follows:

① Construct the data flow graph $G_d = (N_d, E_d)$ of the whole program. N_d is the node set consisting of registers and memory locations. E_d is the edge set that contains the data flow edges $e_d = (n_1, n_2)$, where data value in n_2 depends on n_1.

② Define and identify source/sink nodes in N_d, and use them to construct N_{so} as the set of initial nodes and N_{si} as the set of terminated nodes.

③ Extract all paths from nodes in N_{so} to nodes in N_{si} within the directed graph G_d, which can be formulated as a graph reachability problem $R = (D_d, N_{so}, N_{si})$.

Challenges. Performing a thorough static taint analysis on iOS app binaries poses several challenges.

• *Dynamic features of unique languages.* As we introduced in § 2.1, the default method invocation mechanism in iOS development is a dynamic dispatch. This mechanism is standard in binaries written in Objective-C and Swift, the two primary programming languages used in iOS development. Resolving the actual targets of dynamic calls serves as the foundation for constructing inter-procedural DFG and enables further taint analysis. To this end, static analysis needs to abstract the high-level semantics of dynamic dispatching functions (e.g., `objc_msgSend` and `dispatch_async`), and analyze along control flow to find out clues deciding the final targets of such dynamic calls. Backward slicing [48, 50], forward propagation [52, 82], and their combination [49] are the most widely adopted approaches to solving this problem in prior works. However, with the rapid and massive growth of the binary size, we need a more efficient method to both solve dynamic calls and perform taint analysis. In *iHunter*, we develop an iOS-oriented symbolic execution engine to tackle the dynamic features of languages, build an accurate DFG, and eventually facilitate taint analysis. This engine, together with other tactics, will be described in § 4.3.

• *Broadly utilized system APIs.* Symbolic execution and related techniques can only help us handle instruction-level taint analysis issues. However, the data flow could be truncated when binaries call external functions implemented outside of the binary, due to the lack of high-level semantics of the external functions. Cross-binary function calls are mainly caused by the invocation of system APIs since iOS SDK vendors seldom compile code into multiple binaries to avoid high integration costs for developers. If we do not handle these API calls properly, the data flow will not be complete. An intuitive method would be performing cross-binary analysis on iOS framework binaries to complement the apps' data flow. However, all iOS system APIs are compiled into a single enormous binary (about 3GB) called `dyld_shared_cache`, which is nearly impossible for a thorough static data flow analysis. In fact, it is not feasible to fully load the binary using static analysis tools (e.g., IDA Pro, Ghidra, Hopper) on commodity hardware (with 32 GB memory and 8-core CPU). To effectively and efficiently address the challenges, we develop an NLP-powered taint rule generation technique for iOS system APIs, to extract and abstract taint propagation semantics from iOS API documentation. Our methodology will be presented in § 4.2.

3 Compliance Criteria for Third-Party SDKs

This section elaborates on the criteria used to consider SDKs as non-compliant, based on their adherence, or lack thereof, to prevalent privacy laws such as the General Data Protection Regulation (GDPR) [35] and the California Consumer Privacy Act (CCPA) [34], or Terms of Service (ToS), as enumerated in Table 5. While adhering to GDPR and related regulations, we consider personal data based on state-of-the-art privacy-data ontologies [39, 40], including categories like "device information", "tracking information", "contact information", "location information", etc. We define data collection practices of SDKs as any data flow that accumulates personal data through sensitive source APIs and then transfers this data

to network or storage APIs. SDKs that conduct data collection practices, including storing, collecting, or transmitting, personal data and meet any of the following conditions are considered non-compliant:

• ***Non-Compliance Type I: Absence of a Privacy Policy.*** SDKs, when processing personal data through actions like collection, sharing, storage, or access under their own means and purpose, act as data controllers. As articulated in Article 12 of the GDPR: "The controller shall take appropriate measures to provide any information referred to in Articles 13 and 14 and any communication under Articles 15 to 22 and 34 relating to processing to the data subject in a concise, transparent, intelligible and easily accessible form, using clear and plain language." Failing to provide a privacy policy detailing this information is a violation of GDPR (i.e., Article 12) and CCPA (i.e., Section 1798.100(b)). In our study, SDKs that engage in data processing activities without providing a privacy policy are categorized as Non-Compliance Type I.

• ***Non-Compliance Type II: Inappropriate Disclosure in Privacy Policies.*** Compliance with privacy laws requires not just providing a privacy policy, but also need to provide a correct and comprehensive declaration of data processing practices in the privacy policy. For example, GDPR (Article 5(1)(a)) requires "Personal data shall be processed lawfully, fairly and in a transparent manner ('lawfulness, fairness, and transparency')." Any inconsistencies between actual practices in code behaviors and those stated in the privacy policy are deemed a violation. We employed the flow-to-policy consistency model as detailed in [40]. Our analysis focuses on two key inconsistencies: omit disclosure and incorrect disclosure, both of which are regarded as violations in previous works [81, 87, 92]: (1) Omit Disclosure arises when data is collected, but there are no statements in the privacy policy that disclose or mention this collection. (2) Incorrect Disclosure is present when data is indeed collected, but the privacy policy conversely states that such a data collection does not occur. We classify the SDKs as Non-Compliance Type II if any above violations are detected.

• ***Non-Compliance Type III: Transgression of ToS Provisions.*** For an SDK that accesses user data from other third-party SDKs, we check whether it violates the ToS policy of the latter. The ToS policy of an SDK may impose different restrictions when another SDK tries to access its specific data, in particular "no third-party access", "requiring user consent", and "complying with regulations" [80]. A violation of the SDK ToS policies is considered Non-Compliance Type III.

4 Design and Implementation of *iHunter*

4.1 Overview

Architecture. Our approach relies on extracting data flow statically from SDK binaries and assessing compliance with their respective privacy disclosures to identify non-compliant SDKs. In particular, the design of *iHunter* includes three major components: Semantic Knowledge Extraction, Taint Analysis, and Compliance check, as outlined in Figure 1.

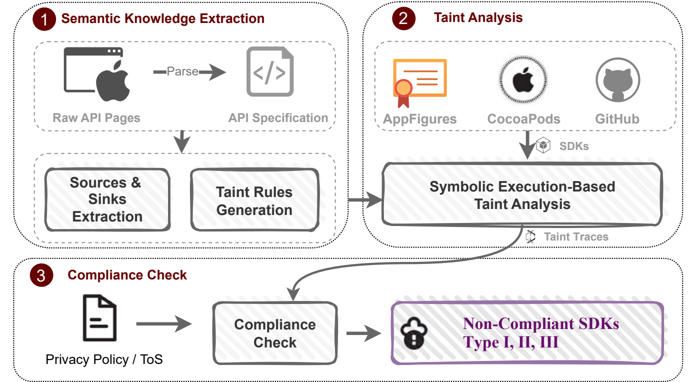

Figure 1: Overview of *iHunter*

Component ❶: Semantic Knowledge Extraction § 4.2 Our approach employs NLP techniques to extract semantic knowledge from API documentation, yielding two key elements: (1) taint sources, identified as sensitive APIs that return privacy data, and (2) taint rules applicable among system APIs. Those semantic knowledge are crucial for constructing a data flow graph in subsequent taint analysis. Note that this process is a one-time preparation step, necessitating execution only once for the entire system.
Component ❷: Taint Analysis § 4.3 Armed with semantic knowledge outlined in § 4.2, *iHunter* employed under-constrained symbolic execution to construct data flow graph (DFG) and further extract taint traces from sources to sinks within this DFG. Each identified taint trace represents a unique data flow, indicating specific data processing activity, such as accessing, storing, or transmitting privacy data.
Component ❸: Compliance Check § 4.4 After extracting the taint traces, we further compare them with the corresponding privacy disclosures. To achieve this, we employed a set of consistencies model in [40, 80] to detect different types of non-compliance SDKs.

4.2 Semantic Knowledge Extraction

Taint sources, sinks, and rules are crucial for constructing the data flow graph during taint analysis. Previous research depended on manually-curated taint sources (10 in [50] and 150 in [48]), which were neither comprehensive nor scalable, especially considering the existence of nearly 60,000 documented public APIs. Traditional methods, focusing on instruction-level analysis, often miss system-level API taint rules, leading to incomplete call graphs and data flow graphs. Our methodology addresses these gaps by automatically formulating taint

sources and taint rules that span system APIs.

Collect API information. We utilized a crawler to extract semantic data from 60,372 pages of Apple developer documents [29], yielding detailed information on 60,153 public API specifications. This data, encompassing aspects like API names, descriptions, parameters, and return value, enriches our subsequent NLP analysis

Source extraction. To detect data flows involving private data , we initially establish a set of sensitive system APIs to serve as taint sources. Our methodology classifies an API as privacy-sensitive if its return value corresponds to privacy terms within a privacy data ontology as mentioned in [39, 40], which constitutes a graph data structure of privacy terms in legal documentation (e.g., privacy policy/terms of use). To this end, we semantically align the data objects in the API specification to privacy data ontology, facilitated by vector computations using an embedding technique. Specifically, we enhance the skip-gram-based word2vec model in [80], incorporating 60,372 Apple iOS documentation. We classify a system API as a sensitive API if the similarity score between its return value and the privacy terms within the privacy data ontology exceeds 0.885. This similarity threshold was empirically determined by manual examination of a random sample of 500 APIs from the Apple documentation (see evaluation details in § 5.2).

Sink collection. We categorized the potential privacy violation behaviors into two major types: (1) *privacy leakage*, which involves transmitting private data out of devices and storing private data in the file system, and (2) *privacy collection* which contains other activities about related to privacy while not being privacy leakage, e.g., logging by the "NSLog" function. The sink APIs associated with these behaviors are typically defined in the official *Foundation* framework and C-type functions, such as "connect" and "send" in POSIX socket programming. Hence, we conduct a manual investigation of the *Foundaiton* framework and various popular third-party network SDKs. Through this process, we collect 372 APIs from the framework and POSIX, along with 26 APIs from third-party libraries. These included 198 network APIs that handle data transmission out of the device, 138 storage APIs that deal with data stored on the device, and 62 other data-collecting APIs.

Taint rule generation. Taint rules serve as a fundamental component in data taint analysis, providing guidance for tracking the flow of sensitive data as it propagates from system APIs to network or file system APIs. Apple provides detailed specifications for each system API in its document website for developers [29], encompassing elements such as function description, signatures, parameters, and return types. That information can be utilized to infer the trajectory of taint propagation between the parameters and the returned data. For example, consider the API `void NSDecimalCopy(NSDecimal *destination, const NSDecimal *source)`, whose function description is "*Copy the value in source to destination*". From the description, we can infer that the taint rule is from *NSDecimal *destination* to *const NSDecimal *source*. Apple's software ecosystem comprises an extensive 259 frameworks, including over 60,000 APIs. Manual inference of taint rules from such a significant quantity of APIs is undoubtedly a laborious and time-consuming undertaking. To generate taint rules automatically, in our study, we implement a series of Natural Language Processing (NLP) techniques to retrieve semantic information of API in the Apple API documentation.

Table 1: Adpositions associated with propagation direction

Direction	Adpositions List Example
To	in, into, onto, towards
From	with, by, for, at, by, via, through

Taint rules between parameters. API descriptions often use patterns to depict the relationship between parameters and their propagation. For example, the description "Adds data to a compression context" uses the adposition "`to`", marking "*data*" as the taint source and "*compression context*" as the taint sink. Conversely, the adposition "`from`" reverses these roles: in "read the source graph from a specified file", the taint source is "a specified file", and the taint sink is "the source graph". We employed the SpaCy dependency parser [23] for automated extraction of taint rules, identifying the `VERB` and `Adposition` to locate the direct object and adposition object respectively. The direction of propagation was determined by constructing two lists of adpositions signifying distinct directions of propagation, derived from a manual examination of the 650 API documentation, as shown in Table 1. This keyword list guided our determination of whether the direct object or the adposition object functioned as the taint source.

Taint rules from parameters to return value. Regarding propagation from parameters to the return value, our base assumption is that any input parameter propagates to the final return value, given that the API's return value is non-void. This is because iOS SDK APIs often process and transform input parameters, thereby incorporating them into the return value. An exception to this rule arises when dealing with instance methods that do not return a value (void return type), but induce a field change in the instance object. This effect is often a result of taint sources (parameters) influencing the object's fields. In such scenarios, the instance object is treated as the taint sink, and the parameters are considered taint sources.

4.3 Taint Analysis

To tackle the challenges associated with analyzing Mach-O binaries on iOS, particularly their dynamic dispatching feature, *iHunter* employs under-constrained symbolic execution. This allows *iHunter* to accurately resolve the targets of dynamic dispatching functions and construct Data Flow Graphs (DFGs), which serve as the basis for taint analysis. The taint

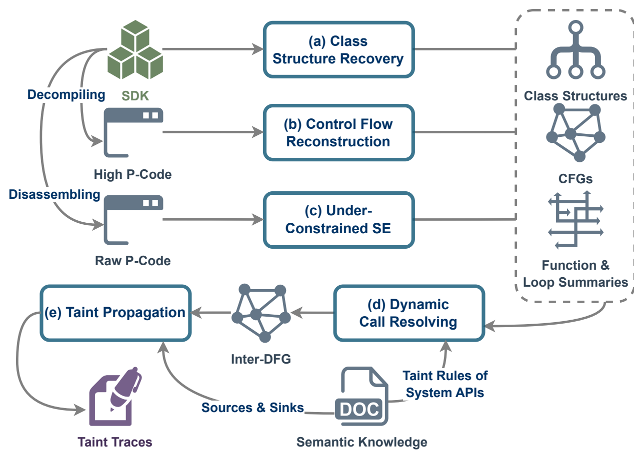

Figure 2: Taint analysis process in *iHunter*

analysis framework in *iHunter* comprises five essential steps, as depicted in Figure 2. These steps are as follows:

(a) Class Structure Recovery. This step is essential for static field-sensitive analysis on binaries compiled from object-oriented languages like Objective-C and Swift. *iHunter* adopts an approach similar to PiOS [50] to parse the header and `__OBJC` segment in the binaries and recover information of classes defined in the binary. The information includes (1) the class inheritance tree, (2) the name, prototype, and address of class methods (3) the name and type of class properties. Additionally, We gather information on dynamically linked libraries in this step to facilitate searching taint sources, i.e., only sensitive APIs defined in linked libraries will be searched before the final taint propagation step.

(b) Control Flow Reconstruction. In *iHunter*, intra-procedural Control Flow Graphs (intra-CFGs) are constructed using basic blocks produced by lifting tools such as *Ghidra*. These basic blocks are then split at each call site to ensure accurate analysis. Loops will present challenges to symbolic execution and type inference in the subsequent analysis. To overcome these challenges, we employ a stack-based topological sorting algorithm during the control flow reconstruction step to identify nested loops and extract essential loop information, including entry points, exit points, and loop levels, which serves as the foundation for generating loop summaries later. Furthermore, in Swift binaries, function names are mangled, which presents a challenge when trying to match these function calls to the APIs in the iOS system SDKs. To overcome this issue, we have developed a demangling tool that can convert these mangled names back into their original function names. The demangling tool achieves this by parsing the mangled names using a finite-state automaton and recovering the encoded type names. The extracted type information is then utilized by *iHunter* to perform an accurate taint analysis in Swift.

(c) Under-Constrained Symbolic Execution. We have developed an under-constrained symbolic execution engine specifically designed for generating Data Flow Graphs (DFGs) in iOS Mach-O binaries. This engine operates under the under-constrained setting, inspired by the approaches in [69, 72], to prevent state exposition during the SE process. The engine is designed to handle various types of variables, including those stored in the stack, registers, or stored in the heap as global variables. In addition, it also collects comprehensive type information to assist in dynamic call resolving during subsequent analysis steps. The SE engine consists of four main stages, including:

Stage ❶: Symbol Initialization. *iHunter* symbolizes all function parameters, stack variables, and heap-allocated variables. This allows us to obtain a more accurate representation of the program's data dependencies and understand the full range of data values used throughout the program.

Stage ❷: Symbolic Execution on Raw p-code In *iHunter*, the symbolic execution process is conducted based on *Ghidra*'s *raw p-code* Intermediate Representation (IR) [13]. *Raw p-code* represents a low-level IR that preserves the original semantics of instructions. To tackle the challenge of context-sensitive taint analysis, we introduce a data structure called the context-switch table to record the call trace of each function encountered during the symbolic execution process. By maintaining this call trace information, *iHunter* can accurately track the flow of tainted data across function boundaries and maintain context sensitivity. Additionally, in *iHunter*, the iOS-oriented symbolic execution engine goes beyond calculating symbolic or concrete values for registers and memory locations. It also propagates type information associated with these values, enabling accurate resolution of the targets of `msgSend` calls. This approach enhances the efficiency of *iHunter* by constructing DFG and collecting information for type resolution in a single pass.

Stage ❸: Loop Summary and Function Summary. Loops and function calls can significantly impact the efficiency of symbolic execution, leading to state explosion and recursive resolving challenges in data flow analysis. To tackle this, *iHunter* generates function and loop summaries by employing symbolic execution within the function or loop during the initial encounter, and subsequently reuses these summaries when encountering them again. Specifically, these summaries include key information such as data flow transmission and symbolic variable type information. When generating function summaries, *iHunter* symbolizes function arguments and explores each path within the function to identify data flows represented as symbolic expressions and variable types. This process accounts for specific side-effects like API calls dependent on particular inputs, discerning specific paths within the function. Similarly, loop summaries are generated by leveraging symbolic execution during the first iteration of the loop, analyzing and tracking symbolic variables within it. Subsequently, during re-entry into the loop (2nd iteration), the symbolic execution finalizes results by identifying any

discrepancies compared to the initial iteration.

Stage ❹: Block Function Processing. Objective-C supports function closure which is compiled into function pointers in designated structures (`NSConcreteGlobalBlock` and `NSConcreteStackBlock`). These structures are frequently used in dispatch functions or as callback parameters of system APIs. *iHunter* traces the data flow associated with the closures, identifies the structures during symbolic execution, and generates a function summary in a specialized format to document the data flow and the type propagation rules for each closure. When the closure structure is transmitted to a dispatcher function, *iHunter* links its context information with the call site and adjusts the symbolic data flow in the generated summaries.

Through symbolic execution, we generate summaries for each function, which can be considered as intra-procedural Data Flow Graphs (intra-DFGs), and form the basis for the subsequent context-sensitivity, flow-sensitivity, and field-sensitivity inter-procedural taint analysis [66].

(d) Resolving Dynamic Calls. *iHunter* resolves the targets of `msgSend` dynamic calls by resolving the type of objects and selector names passed into `msgSend` during this step. As the dynamic dispatching calls are resolved, *iHunter* constructs the data flow in the whole program as the inter-DFG. To this end, *iHunter* employs a two-pass approach. During the first pass, *iHunter* leverage the context-sensitive intra-DFGs generated by SE and taint rules of system APIs extracted from documents. By a rapid backward slicing approach leveraging these data flow summaries, *iHunter* identifies all message-sending sites and endeavors to infer the class type of the message sender, which demonstrates efficiency compared to prior approaches [48, 50, 52]. In the second pass, *iHunter* focuses on refining the resolved dynamic call targets. A work-list algorithm is employed that iteratively updates and refines the dynamic call targets. The refining terminates until both the control flow and data flow stabilize, reaching a fix-point, meaning that the resolving has been carried out to a quite complete level. For other indirect calls in Swift, which can be C++ like function pointers, *iHunter* utilizes dataflow in summaries and tracks addresses stored in memory, as well as the memory block referenced by function pointers. It establishes matching between function pointers and addresses to function to determine the targets of C++ style indirect calls in Swift code.

(e) Taint Propagation. In this final analysis step, *iHunter* extracts all taint traces from the inter-DFG. Considering the propagation efficiency, *iHunter* first simplifies the DFG into a taint graph, which exclusively includes data flows connecting to a source or a sink. Subsequently, it traverses this taint graph using a flooding algorithm to determine reachability from sources to sinks, thus generating the taint trace results.

4.4 Compliance Check

This section details the methodology *iHunter* uses to identify the three types of privacy violations referenced in §§ 3.

Non-Compliance Type I. SDKs processing personal data without an available privacy policy are labeled as Non-Compliance Type I. To determine the privacy policy of an SDK, we leverage the polisis [55] tool, which fetches privacy policies using an SDK's official website URL. Retrieving these official websites varies based on the SDK's source platform: AppFigures, CocoaPods, or GitHub. For SDKs from AppFigures, the platform directly provides the official website URL for each top-ranked SDK listed. CocoaPods-hosted SDKs come with specifications. We analyze these specifications to derive the SDKs' homepage URLs. GitHub, on the other hand, poses a challenge due to the absence of direct links to official websites. To circumvent this issue and prevent overlooking any official websites, we use a semi-automatic approach. First, we extract all URLs located in the SDK's README file. Subsequently, we manually discern the URL that represents the SDK's official website. If no URL is discovered, the SDK is considered as not providing a privacy policy. Given the low frequency of URLs in README files (see details in §§ 6), conducting a manual examination is feasible.

Non-Compliance Type II. SDKs displaying inconsistencies between their code's data collection practices and disclosures in their privacy policy are categorized as Non-Compliance Type II. Specifically, we employed PoliCheck [40] to extract data-action tuples from their policy statements in the privacy policy. Then we utilized the flow-to-policy model of PoliCheck [40] to compare the dataflows that were detected from code with data-action tuples in privacy policy to detect any omit and incorrect disclosure.

Non-Compliance Type III. SDKs that illicitly obtain data from other "victim" SDKs, thereby breaching the Terms of Use (ToS) of these victim SDKs, are designated as Non-Compliance Type III. For any suspected unauthorized data retrieval from a victim SDK, it's imperative to ascertain if the caller SDK breaches the victim SDK's ToS. Using the approach outlined in [80], we extract the conditions under which individual data items are restricted by the ToS and then check whether caller SDK satisfies those requirements. There are two main conditions: (1) *No Access Allowed*: Under this condition, any access to the data by caller SDKs from victim SDKs is a violation. (2) *Regulation Compliance Required*: Violations arise if SDKs inadequately or incorrectly disclose data practices in their privacy policies that contradict regulatory mandates. To validate compliance, similarly, we analyze the caller SDK's privacy policy, ensuring accurate data collection practice disclosures.

4.5 Implementation

The *iHunter* system is implemented based on the Ghidra [14] reverse engineering platform as an analysis plugin, specifically implemented under Ghidra v10.3.1. The entire system is written in Java, Python, and Swift, with a total of about 29,000 lines of code. For the static taint analysis part, *iHunter* leverages two types of IR in Ghidra: *raw p-code* generated from disassembling for symbolic execution and *high p-code* generated by the Sleigh [22] lifter/decompiler for control flow reconstruction. For multiple binaries within a single SDK, *iHunter* treats all of them as components of the SDK and conducts independent analysis on each binary.

5 Evaluation of *iHunter*

In this section, we perform comprehensive evaluations of both the overall system of *iHunter* and the individual components. We aim to assess the effectiveness of each component as well as their combined impact and performance on the system. All evaluations of *iHunter* were conducted on three MacBooks with 16/32 GB memory. Our implementation ensures that the results remain consistent across various MacOS devices.

Evaluation dataset. We established a ground truth dataset named D_g to evaluate the effectiveness of *iHunter*. First, we randomly selected 50, 40, and 40 SDKs from AppFigures, Cocoapods, and GitHub respectively, ensuring the generality of our results. We further employed IDA Pro [56] and Ghidra [14] to reverse-engineer those SDKs, manually tracing the path of privacy data from source APIs to sink APIs (e.g., network transmission or file system storage). In total, we identified 325 taint traces that process privacy data.

5.1 Evaluation of the Entire *iHunter* System

Overall effectiveness of *iHunter*. *iHunter* analyzes SDKs along with their relevant privacy policies or Terms of Service (ToS), identifying and categorizing SDKs into three distinct non-compliance types. To evaluate the overall effectiveness of *iHunter*, we first built a ground-truth SDK dataset (D_g) and conducted a manual check to identify taint traces of privacy data. Subsequently, we assessed the non-compliance of these data practices by (❶) manually searching the SDK's privacy policy, flagging its absence as Non-Compliance Type I; (❷) if a privacy policy was present, manually verifying whether these data practices are appropriately disclosed; and (❸) if the data is harvested from another victim SDK, evaluating whether such practices violate the victim SDK's ToS.

Ultimately, 59 (45.4% in 130) SDKs were deemed non-compliant, with classifications of 49 (83.1%) as Non-Compliance Type I, 10 (16.9%) as Type II, and 4 (6.8%) as Type III, while 3 (5.1%) as both Type I & III and 1 (1.7%) as both Type II & III. On the ground-truth dataset, *iHunter* identified 209 taint traces and classified 179, 25, and 5 as Non-Compliance Type I, II, and III, respectively.

Table 2: Overall effectiveness of *iHunter*. *"#SDKs" refers to the number of non-compliant SDKs detected by iHunter.*

Non-Compliance Type	#SDKs	Precision	Recall	F1
Type I	49	83.7%	74.5%	78.8%
Type II	10	60.0%	66.7%	63.2%
Type III	4	75.0%	100%	85.7%
All types	**59**	**81.4%**	**75.0%**	**78.1%**

• *Results. iHunter* achieve an average F1 score of 78.1%, marked by 81.4% precision and 75.0% recall. As detailed in Table 2, *iHunter* demonstrated a precision of 83.7% and a recall of 74.5% for Non-Compliance Type I, a precision of 60.0% and a recall of 66.7% for Type II, and a precision of 75.0% and a recall of 100% for Type III. *iHunter* exhibits superior detection accuracy for Non-Compliance Type I and III compared to Type II. This discrepancy can be attributed to the incomplete extraction of private data declarations using existing NLP tools (PoliCheck [40] and Polisis [55]), resulting in relatively low precision and recall. To mitigate the potential instability associated with using prior tools, we manually checked and confirmed the results for Non-Compliant SDKs Type II detected by *iHunter* and their corresponding privacy policies, which helps ensure the accuracy and reliability of our measurements (§ 6.1).

• *False positives and negatives analysis.* In addition to the accuracy of the privacy policy extraction tool affecting the results, the rest false positives were mainly caused by inaccurate data flow propagation, such as data transmission in Objective-C collection classes (e.g., `NSArray`, `NSDictionary`). This confusion in data flow introduced wrong crossings between instances and pointers in the containers, which resulted in the generation of non-existing taint traces. On the other hand, false negatives were mostly caused by the limitations in resolving dynamic calls of templated classes and containers, which affected *iHunter*'s ability to identify source API calls.

Performance overhead. The average time consumed to analyze a binary in our datasets with the average size (2.2 MB) is 7.5 min, consisting of 4 minutes of loading and pre-analysis time by *Ghidra* and 3.5 minutes of analysis performed by *iHunter*. To analyze 6,401 SDKs in our three measurement datasets, it took almost two weeks to complete the analysis on our three testing devices. This analysis is efficient considering the measured time starts from when the binary is being loaded into *Ghidra* (which was commonly considered to be time-consuming) to when *iHunter* generates all taint traces. Such high performance is the consequence of a combination of hand-picked techniques like on-demand type inference, one-pass symbolic execution, loop summary, function summary, etc. In our evaluation of *iHunter*, we observed that

the maximum memory usage during the analysis reached 19GB. This was mainly due to the decompilation process performed by the *Sleigh* lifter [22], which requires significant memory resources to execute. Since all of the analyzing stages in *iHunter* are parallelizable, the analyzing process can be significantly accelerated by running *iHunter* on a server parallelly.

5.2 Effectiveness of Individual Components

Effectiveness of source API extraction. To evaluate the performance of source API extraction, we took a random sample of 500 APIs out of the total APIs incorporated from Apple iOS documentation. Through careful examination of the API specifications, we assigned each API a label of being either privacy-sensitive or non-privacy-sensitive. Utilizing our enhanced skip-gram-based word2vec model and a similarity threshold of 0.825 (which was empirically determined), we classified APIs as sensitive or not. This threshold was set to capture the similarity between the return value of an API and the privacy terms within the privacy data ontology. Our methodology proved to be highly effective since it successfully distinguished privacy-sensitive APIs with a precision rate of 88.5% and a recall rate of 92.4% on the manually annotated dataset. In general, our model flagged 2,340 sensitive APIs from the total pool of APIs. Following independent manual verification by two researchers, 1,951 APIs were confirmed as true positives and subsequently incorporated into *iHunter*'s analysis.

Effectiveness of taint rule generation. To gauge the effectiveness of our approach for taint rule extraction, we randomly sampled 500 APIs from a total of over 60,000 APIs that span across Apple's 259 frameworks. These APIs underwent manual analysis to establish ① propagation rules between parameters and ② rules between parameters and return values, serving as the ground truth. Under evaluation, *iHunter* achieved precision/recall rates of 85.9%/79.1% on ① and 89.6%/94.0% on ② respectively. Cumulatively, our methodology identified 1,395 propagation rules between parameters and 21,077 propagation rules between parameters and return values. This assessment further underscores the accuracy and precision of our automated approach in identifying taint propagation rules.

Effectiveness of data flow analysis. To assess the capability of *iHunter* in conducting static data flow analysis on iOS binaries, an evaluation was conducted to evaluate its accuracy and effectiveness in resolving dynamic calls. Specifically, we conducted comparative experiments using IDA Pro v8.3.2 as a baseline. We used IDA Pro and *iHunter* to parse the 41 SDKs in the subset of the ground truth SDK dataset D_g and analyzed the results of resolving dynamic calls using both tools. Of the calls dispatched by message, there were a total of 524,839 call sites for the `objc_msgSend` function or the `dispatch_async` function. Through the two-phase approach described in § 4.3, *iHunter* was able to resolve 480,091 dynamic calls **(91.5%)**, including both object types and selector names. This shows an improvement compared to the static analysis component in PiOS [50] (82%) and iRiS [48] (85%). Compared to the SoTA commercial reverse engineering tool, IDA Pro [56] was able to resolve 393,899 (75.1%) message-sending call sites. In addition, 352,784 (95.2%) of the call sites that could be resolved by both IDA Pro and *iHunter* produced the same resolving results.

As for the dynamic calls that *iHunter* can't resolve, several limitations contribute to this situation: **(a)** Tracking pointers stored in container instances like `NSArray` or `NSDictionary` can be challenging, and inferring their types is not straightforward. **(b)** There are numerous APIs exposed to external apps that are not called within the SDK. It becomes much harder to resolve message sending in these functions when the calls come from a parameter of these functions.

Additionally, we found that there were 20 SDKs written in Swift or partly written in Swift. The recall rate for resolving indirect calls in Swift code is 74.4%, notably lower than in Objective-C programs (91.5%). In Swift, some complex mangled function names and indirect calls through function pointers may not be fully resolved through symbolic execution, leading to this discrepancy. Utilizing *iHunter*'s ability, we were able to identify 7,819 Swift calls of dynamic dispatch to Apple's iOS SDKs. Until this point, other iOS binary analysis tools still lack this feature and Swift support.

Effectiveness of taint propagation. To separately evaluate the accuracy of taint propagation performed on the DFG, we constructed a dataset consisting of 325 manually identified taint traces in D_g. During the evaluation process, the source points and sink points of taint propagation were fixed to the termination points of these manually identified taint traces. Under the evaluation setting described, *iHunter* achieved a successful tracking rate of 85.8%, meaning it accurately traced the propagation of tainted data in 279 out of the total taint traces. The precision of the analysis was measured at 77.7%, so the F1 score was calculated to be 82.4%, affirming the effectiveness of our data flow analysis and taint propagation process.

6 iOS SDK Non-Compliance in the Wild

Running *iHunter* on 6,401 iOS SDKs, we show that the non-compliance of iOS SDKs is prevalent. We conducted a measurement study to understand the landscape of the non-compliant iOS SDKs and assess their real-world privacy implications for end-users and compliance risks of iOS apps that integrated the SDKs.

Measurement dataset. We made a broad collection and sampling from SDK repositories and built several datasets to study the situation of non-compliance privacy collection in the iOS apps' supply chain. As shown in Table 3, our collection comprises (1) 140 SDKs from $D_{AppFigures}$, 2,038 SDKs from D_{GitHub}, and 4,356 SDKs from $D_{CocoaPods}$, specifically

Table 3: Summary of datasets and corpora

Name	Size	Source	Timestamp	Usage
C_{api}	60,372 well-formed document pages	Apple Developer Documentation [29]	202305	Detection
$D_{AppFigures}$	140 SDKs	top SDKs in *AppFigures* [3]	202305	Detection
D_{GitHub}	2,038 SDKs	top-starred in *Github*	202306	Detection
$D_{CocoaPods}$	4,356 SDKs	sampled from *CocoaPods* [10]	202306	Detection
D_{sp}	244 SDKs' privacy policies	websites and repositories	202306	Detection
D_{ToS}	52 SDKs' Terms of Service	websites and repositories	202306	Detection
D_a	32,478 iOS apps	App Store top-popular list	2020-2022	Measurement
D_{ap}	1,358 apps' privacy policies	privacy links in App Store	202307	Measurement
D_{al}	1,040 apps' privacy labels	privacy labels in App Store	202307	Measurement
D_g	130 SDKs	randomly selected SDKs	202303	Evaluation

earmarked for detecting privacy violation behaviors, (2) privacy policies associated with 244 SDKs*, and (3) terms of service (ToS) for 52 designated victim SDKs. Further, to assess their real-world impact within the app ecosystem, we constructed a dataset of 32,478 apps collected from the App Store, denoted as D_{app}. These apps were collected from the App Store in July 2022 and were selected based on their inclusion in the top popular list or the top new list of the App Store or the *AppFigures* website. Among 32,478 apps, 15,336 (47.2%) of them integrated non-compliant SDKs. To assess how non-compliant SDKs lead to apps' privacy violations, we collected 1,358 privacy policies and 1,040 privacy labels from App Store pages.

6.1 Landscape

iHunter detected 2,585 (40.4%) non-compliant SDKs among a total of 6,401 iOS SDKs, revealing the prevalence of non-compliance. In this section, we perform an in-depth analysis of those non-compliant SDKs, examining their types of non-compliant, distribution sources, and categories.

6.1.1 SDKs of Different Non-Compliance Types

Using the compliance criteria defined in §§ 3, *iHunter* identified 2,436 SDKs with Non-Compliance Type I, 58 with Type II, and 45 with Type III†.

Non-Compliance Type I. In accordance with privacy laws, SDKs that process personal data without a privacy policy are considered Non-Compliance Type I. In our measurement, we extract the official websites of these SDKs and utilize polisis [55] to retrieve their privacy policies. We obtained 41, 110, and 93 privacy policies of SDKs with data collection practices (i.e., the SDKs get data through sensitive APIs, then send the data to the Internet through sink APIs (e.g., network APIs). in $D_{AppFigures}$, D_{GitHub}, and $D_{CocoaPods}$ respectively. Consequently, we identified 27 non-compliant SDKs

*We obtained privacy policies for 2,680 SDKs found to engage in privacy collection activities. However, only 244 were found, while the remaining 2,436 SDKs lacked privacy policies.

†Some SDKs may come with multiple non-compliance types.

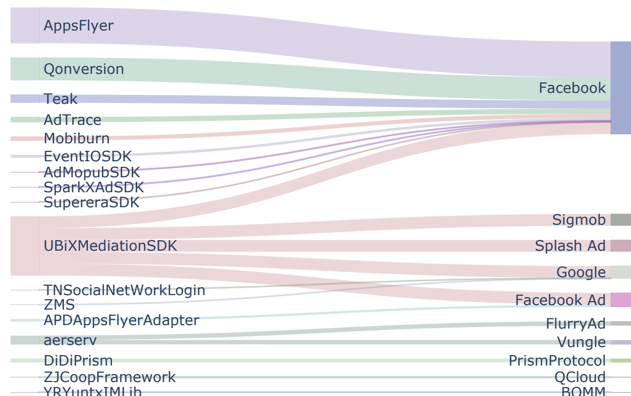

Figure 3: ToS Non-Compliant Flows: Non-Compliant SDKs Type III (left) fetching data from victimized SDKs (right)

in $D_{AppFigures}$, 454 non-compliant SDKs in D_{GitHub}, and 1,943 non-compliant SDKs in $D_{CocoaPods}$.

Non-Compliance Type II. *iHunter* initially identified 74 SDKs with Non-Compliance Type II. However, due to the potential imprecision resulting from the adoption of prior NLP tools for data disclosure extraction in the privacy policies of these SDKs (see § 5.1), we conducted a manual verification of the results before using the results for measurement. As a result, we confirmed 58 SDKs with Non-Compliance of Type II across two sub-categories:

• *Omitted Disclosure.* A data flow has an omitted disclosure when there are no policy statements that disclose it. We found 53 SDKs that failed to disclose an average of 102 data harvesting practices. The most commonly omitted data types included system running status, cellular service provider, and apps' active time.

• *Incorrect Disclosure.* A data flow has an incorrect disclosure if a policy statement indicates that the flow will not occur (i.e., a negative sentiment sharing or collection statement) and there is no contradicting positive sentiment statement. On average, 7 SDKs were found to provide incorrect disclosures about their data practices.

Non-Compliance Type III. SDKs that harvest user data from other third-party SDKs are considered with Non-Compliance

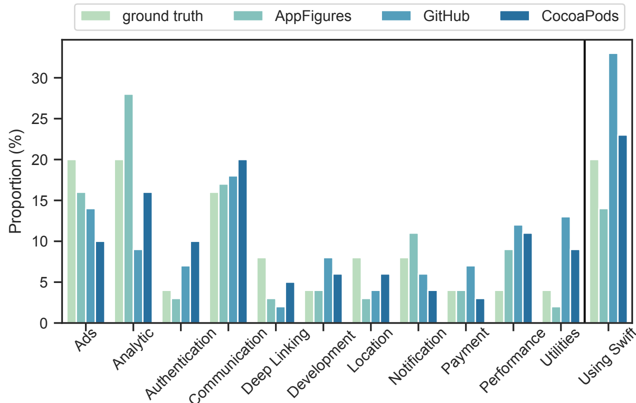

Figure 4: Distribution of non-compliant SDKs across different usage categories

Type III. Our analysis identified 45 SDKs with such non-compliance, which were integrated into 469 apps, targeting 15 victim SDKs. Figure 3 illustrates the victim SDKs and the corresponding number of Non-Compliance Type III that target them. The top 20 non-compliant SDKs, based on the number of apps affected, are considered. It should be noted that over half of these victimized SDKs are categorized as online social networks (OSN) and advertising, indicating that well-known OSN platforms and advertising providers are valuable sources of private user data for harvesting. Our study also uncovers the stealthy collection of advertising tracking tokens and access tokens from Facebook, Google, and WeChat by certain analytic SDKs. These tokens can potentially be exploited to gain unauthorized access to semantically rich personal data.

6.1.2 Distribution Source of Non-Compliant SDKs

During our measurement analysis, *iHunter* detected a total of 2,585 non-compliant SDKs across the three datasets. Specifically, 54 non-compliant SDKs were identified in $D_{AppFigures}$, 482 in D_{GitHub}, and 2,049 in $D_{CocoaPods}$. To understand potential differences in data collection practices among SDK sources, we calculated the average frequency of privacy violations for each dataset. As shown in Figure 6, there is a noticeable variation in the rate of privacy data collection across the datasets. Non-compliant SDKs collected from AppFigures based on AppFigures' statistics have an average of 9.9 practices of non-compliant privacy collection per SDK. In contrast, the top-starred SDKs from GitHub have an average of 2.9 non-compliant practices, while the SDKs randomly sampled from CocoaPods have an average of 4.2 non-compliant practices. Notably, the SDKs with high rankings in AppFigures are usually being widely adopted and favored by app developers — posing potentially high privacy risks.

Table 4: Severity Breakdown for the violations. *"No." refers to the number of privacy violations.*

Data Sensitivity	Data Examples	Non-Compliance Type	No.
Sensitive PII	Biometric Data,	Type I	322
	Health,	Type II	5
	Precise Location	Type III	22
PII	Email Address,	Type I	2,826
	Coarse Location,	Type II	67
	User ID	Type III	32
Non-PII	Crash Data,	Type I	2,388
	Performance Data,	Type II	70
	Diagnostic Data	Type III	7

6.1.3 SDK Categories

As shown in Figure 4, the predominant non-compliant SDKs serve purposes related to advertisement & monetization, analytics, and communication. Those SDKs have inherent motivations to amass a wealth of user data. Their data harvesting behavior is driven not only by their need to facilitate user analytics and cater to various commercial services but also by their significant reliance on advertising revenue and practices of monetizing user data, potentially selling data to external entities such as data brokers. Conversely, the open-source SDKs on GitHub are typically designed to assist with development, such as the Charts SDK [9] for plotting charts in apps. These types of utility SDKs typically may tend not to collect much user data leading to noncompliance.

6.1.4 Severity Breakdown for the Violations

To better characterize the severity of the violations we found, we further categorize data types into three groups: Sensitive PII, PII, and non-PII (based on definitions by the Department of Homeland Security [12]), with each group's relative severity regarding a privacy violation ranging from high to low. In our study, among the overall 5,739 non-compliance violations, we find 349 in Sensitive PII, 2,925 in PII, and 2,465 in non-PII. A detailed breakdown of each non-compliance type, along with their respective sensitivity levels, is presented in Table 4.

6.2 Real-World Impact of Non-Compliant SDKs

iHunter detected 2,585 SDKs as non-compliant, of which 694 (27.9%) contained data transmission out of the device. These SDKs have been integrated into 15,336 apps, which corresponds to 47.2% of our analyzed top-tier apps across 25 categories in the Apple App Store. This highlights substantial privacy risks for end-users. In this section, we study the real-world privacy impact of the non-compliant SDKs by examining their popularity and assessing how their non-compliance leads to non-compliance of iOS apps that integrate the SDKs.

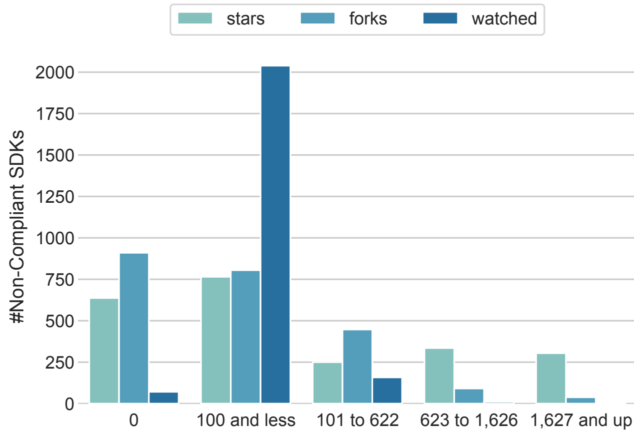

Figure 5: Distribution of GitHub *stars*, *forks*, and *watches* of non-compliant SDKs

6.2.1 Popularity of Non-Compliant SDKs

Assessing the popularity of non-compliant SDKs is crucial as it sheds light on their widespread adoption and potential ramifications in the broader iOS ecosystem. For this assessment, we consider two metrics: (1) engagement metrics on GitHub: *stars*, *forks*, and *watches*; (2) number of real-world integrations of the SDKs in real-world iOS apps.

Engagement on GitHub. For the non-compliant SDKs hosted on GitHub (2,291 in total), we evaluated user interactions by counting primary engagement actions: watching, forking, and starring. These actions elucidate the varying degrees and kinds of user engagement. As shown in Figure 5, a large proportion of non-compliant SDKs, specifically 38.8%, have received more than 100 stars, indicating a relatively high user preference for these SDKs.

Integration within apps. To measure the integration of the non-compliant SDKs in real-world scenarios, we employed a class structure matching algorithm (details in Appendix A) to scrutinize SDK presence within apps, which allows us to understand the prevalence of their adoption and distribution pattern. We found that 47.2% of apps integrated at least one non-compliant SDK.

6.2.2 Propagation of Non-Compliance into iOS Apps

App developers, based on privacy regulations such as the GDPR and Apple app store policies, are obligated to transparently disclose all data handling behaviors, including those performed by third-party libraries. From our dataset, we discerned 14,811 instances of non-compliant data collection across 339 SDKs integrated into 15,336 apps. Among these apps, only 1,358 apps provided links to privacy policies, and 1,040 apps provided privacy labels on the app's Apple app store pages. This section assesses the extent to which non-compliant SDKs integrated into apps led to the app's non-compliance (i.e., the apps' privacy policies or privacy labels did not disclose the data practices of the SDKs).

Non-Compliant privacy policy in apps. When app developers integrate third-party libraries that collect personal data, they remain the primary controller and are thus responsible for ensuring that users are comprehensively informed in the privacy policy, as stipulated by GDPR's Article 13 (i.e., Article 13[‡]). However, data practices performed by SDKs are often opaque to app developers. Without meticulous examination of these practices, non-compliant behaviors from SDKs might inadvertently propagate into apps, posing significant privacy risks to end-users. To evaluate the adequacy of privacy policy disclosures for data practices by SDKs, we collated apps' privacy policies and employed PoliCheck [40] to extract data items and their associated actions from these policies. Subsequently, PoliCheck's consistency model was utilized to verify the comprehensiveness and correctness of these disclosures. In our analysis, *iHunter* identified 6,393 non-compliant data practices across 106 SDKs, encompassing 1,358 apps. Notably, 1,827 (28.6%) of these practices were reflected in apps, inducing non-compliance of apps. Specifically, 279 (20.5%) apps omit to disclose 1,352 data items, most commonly omitting details like location, operating system, and device information. While 105 (7.7%) apps inaccurately claim 475 data collection actions do not occur. The most frequently incorrectly disclosed data items are user identifiers, phone numbers, and purchase history.

Non-Compliant privacy labels in apps. Furthermore, our study explored how non-compliant data practices of SDKs might affect the accuracy of the privacy labels in apps that incorporate them. Apple mandated that app developers must transparently disclose data collection practices throughout their applications, including those carried out by third-party affiliates. Failure to disclose these practices in integrated SDKs would be a violation of Apple's guidelines [32]. Since privacy labels specifically require the disclosure of data that is transmitted off the device, we only focus on non-compliant data transmission practices in SDKs. To examine whether such non-compliant data transmissions in SDKs are propagated to the app's privacy label, we check whether the data is in the app's privacy label, if it is, we regard the app's privacy label as non-compliant. To ascertain their disclosure compliance, we examined 1040 apps' privacy labels and scrutinized the representation of these non-compliant transmissions. We observed that 469 apps omit 593 data in their privacy label disclosure, indicating a significant The data items most frequently omitted include product interaction, payment info, and location.

[‡]When personal data is procured from the data subject (e.g., app user), the controller (app developer/owner) must provide, at the time when personal data is obtained, detailed information about the data processing.

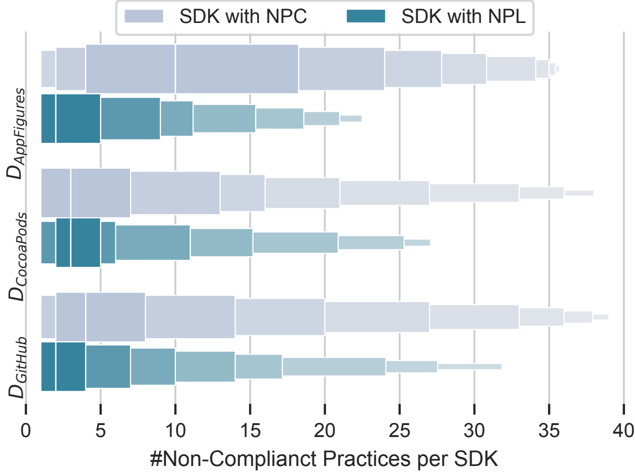

Figure 6: Distribution of Non-Compliant Privacy Collection (NPC) or Leakage (NPL) practices per SDK

6.3 Case Study

Non-Compliance Type I. *iHunter* identified a substantial number of widely used SDKs engaging in the processing of user privacy data without offering a corresponding privacy policy. Notable examples include high-profile SDKs such as SuperPlayer [24], KSPhotoBrowser [15], and LBLaunchImageAd [16], all of which accessed device model and system version information without providing privacy policies. An in-depth examination of the real-world ramifications of these three non-compliant SDKs revealed their presence in 564, 1,147, and 1,560 apps within dataset D_a, respectively. Moreover, we assess the degree to which these SDKs contribute to non-compliance in apps' privacy labels. Our findings revealed that 282, 519, and 741 applications exhibited nonconformity with their privacy labels due to the integration of SuperPlayer [24], KSPhotoBrowser [15], and LBLaunchImageAd [16], respectively, indicating a significant privacy risk posted to end-users by non-compliant SDKs.

Non-Compliance Type II. OpenMediation [18] is a popular SDK that is widely integrated into mobile games for advertising analytics and notifications. According to OpenMediation's privacy policy, the SDK is intended to collect only device information, such as the device model and screen size, solely for user identification purposes. However despite OpenMediation's stated privacy policy, *iHunter* uncovered that the SDK secretly collects additional sensitive data about the phone's running status. This includes information on RAM/storage usage, CPU thermal level, battery level, and precise system boot and app usage times. What is even more concerning is that this collected data is transmitted to OpenMediation's server without the knowledge or consent of the user. Another example of SDKs with discrepancies in their stated data collection practices is Sentry [21] and Buglift [8]. Both SDKs claim to collect only non-personal device information for identification purposes. *iHunter* revealed that Buglift secretly collects location data, which goes beyond the scope of non-personal device information mentioned in its privacy policy. Similarly, Sentry was found to collect data about users' activities, such as CPU time and storage write counts, which again exceeds the explicitly stated data collection purpose.

Non-Compliance Type III. In our case study, we present an example of Cross-Library Data Harvesting on iOS, involving a malicious third-party SDK called Teak [28]. To execute this illicit data harvesting, Teak employs reflection methods to intercept the Facebook user's access token after the login process, and these unlawfully obtained tokens are then sent to a remote server located at `https://gocarrot.com/games/XXX/users.json`. Notably, Teak fails to provide a transparent privacy policy that explicitly outlines this unauthorized data collection and transmission. Significantly, Facebook's terms of use regard access tokens as restricted data items, barring their sharing or transfer to a third party under any circumstances. Compounding the issue, Teak lacks a privacy policy that discloses this data collection behavior. Alarmingly, this SDK is embedded in numerous iOS applications, including those that rank high in popularity in the App Store, posing a significant privacy risk to Facebook users.

7 Discussion

Responsible disclosure. We have reported all our discovered privacy violations to related vendors or developers, including Apple, developers of non-compliant SDKs, and vendors of victim SDKs. The Apple security team told us they are conducting an investigation based on our reports. Some vendors of non-compliant SDKs have confirmed that they are investigating the issues based on our reports, such as Adyen [1], vk-ios-sdk [30], and Countly [11]. For instance, the development team behind the Adyen SDK [1] has informed us that they are in the process of implementing improvements to comply with the regulation requirements set to take effect in Spring 2024. Furthermore, for Type-III violations, some of the victim SDKs have acknowledged our results including Facebook and Twitter; for example, Facebook has taken actions against violating SDKs including Mobiburn [27], Teak [28] and RevMob [20] that collect user data from the Facebook SDK in real iOS apps and send data to their remote servers. Facebook told us that the actions include blocking Facebook Login for iOS apps that integrate the malicious SDKs, which could urge the affected iOS apps to get rid of the violating SDKs. With more vendor responses expected on the way, we will keep updating the results on our website [36].

Usage scenario of *iHunter*. Recently, Apple announced the development of a new mechanism called privacy manifests, aimed at addressing privacy concerns in third-party SDKs. Privacy manifests are files designed to detail the privacy practices of third-party code. Privacy manifests are files that outline the privacy practices of the third-party code. When an app is pre-

pared to be distributed, Apple's official IDE, Xcode [31], will synthesize privacy manifests from all third-party SDKs used by the app into a single report. This comprehensive report can help developers to create privacy labels and complement privacy policies. Privacy collection and leakage behaviors identified by *iHunter*'s taint analysis can greatly help SDK vendors create accurate privacy manifests, which will eventually benefit app developers and users.

8 Related Work

Privacy issues on iOS apps. PiOS [50] examined privacy leakage in iOS apps with a static analysis tool solving the unique indirect `objc_msgSend` calls of Objective-C. iRis [48] then studied the usage of private APIs in iOS apps with a combination of both static and dynamic analysis to improve the ability to resolve `objc_msgSend`. Other works [63, 90] surveyed the usage of private APIs or pursued a source-to-sink analysis using static and dynamic methods on the iOS platform. A crowdsourcing study in [37] indicated that access to users' privacy on devices is widespread in iOS Apps. [83] proposed a fine-grained, application-specific, and user-driven sandboxing for iOS apps to protect user privacy at runtime. Recent studies [59, 84] focused on the impact and compliance of Apple's newly introduced privacy labels. Compared to prior works, *iHunter* is the first to systematically study privacy issues in iOS Apps' supply chain.

Security analysis on iOS apps. Other than privacy analysis in iOS apps, prior works [44, 48, 49, 50, 52, 54, 61, 75, 82] also focused on analyzing the security of iOS apps with static or dynamic approaches, including efforts [48, 50, 52] to deal with dynamic features of Objective-C binaries. Different from PiOS [50] and iRiS [48] leveraging the IR of IDA Pro [56], [52] decompiled and lifted binaries into LLVM IR by the reverse-engineering tool dagger [38], and then performed slicing on the LLVM IR to detect vulnerabilities in iOS. [75] examined vulnerabilities in iOS apps' network services through dynamic vetting in a large number of app collections. Kobold [49] studied the security of IPC channels on iOS with a combination of a static analysis that extracts IPC service interfaces and a dynamic analysis that tests the security of the interfaces. iService [82] performed a top-down static analysis to detect and evaluate the impact of confused deputy problems in macOS Objective-C frameworks. Compared to prior works on iOS, *iHunter* is the first static analysis tool, to the best of our knowledge, to generally handle both Objective-C and Swift binaries.

Privacy issues on Android apps and supply chain. Taintdroid [51] is a system-wide dynamic taint tracking and analysis system capable of tracking privacy leakage on Android. Flowdroid [41] is a static taint analysis framework for Android apps that can be used to examine privacy leakage and security hazards. [86] analyzed transmission of sensitive data in Android apps to privacy leakage that is not related to GUI events. [73] examined privacy policy violations on Android apps and discovered a high portion (341 in 477) of violations.

With these off-the-shelf tools on Android, [39, 40, 45, 67, 73, 81, 88] extensively studied various privacy issues on Android apps. PoliCheck [40] introduces an entity-sensitive consistency model that considers the nature of personal data. PurPliance [43] presents an innovative consistency model, building upon the foundation of PoliCheck [40].

Recent studies [47, 70, 74, 77] have highlighted the alarming prospect of malicious SDKs gaining unauthorized access to users' sensitive data through both host apps and mobile devices. To tackle this issue, researchers conducted both static [67] and dynamic [46, 70] analysis techniques in the measurements of the Android apps' supply chain. This rampant integration and adoption of these SDKs by popular mobile apps have raised serious concerns about potential privacy breaches and data leaks in Android apps' supply chain. To defend or mitigate this threat, prior studies conducted isolation or interdiction approaches [57, 71, 78, 79] on these malicious SDKs.

9 Conclusion

We introduced *iHunter*, a comprehensive system designed to systematically scrutinize privacy compliance among iOS SDKs and detect potential violations. *iHunter* stands out with its unique methodology, utilizing symbolic execution to perform data flow analysis, skillfully analyzing binaries compiled from both Objective-C and Swift. Additionally, it leverages the power of NLP for semantic knowledge extraction, automatically identifying privacy-related taint sources and generating taint rules for iOS system APIs. The system culminates with a compliance check module specifically designed to identify three unique types of non-compliant SDKs. Running on 6,401 iOS SDKs, *iHunter* identified a significant number of non-compliance SDKs (40.4%). Those non-compliant SDKs led to the 15,336 non-compliance apps that integrated them. These results shed light on the pervasiveness and severity of privacy violations in the iOS software supply chain.

Acknowledgment

We would like to thank our shepherd and the anonymous reviewers for their valuable comments on this paper. Dexin Liu is supported in part by the National Key Research and Development Program of China (No.2021YFB3101802) and the Beijing Municipal Science & Technology Commission (No.Z231100010323002). The IU authors are supported in part by Indiana University's IAS Collaborative Research Award and by NSF CNS-2330265.

References

[1] Adyen. https://www.adyen.com.

[2] App store review guidelines. https://developer.apple.com/app-store/review/guidelines/#data-collection-and-storage.

[3] Appfigures. https://appfigures.com/top-sdks.

[4] Apple blocked apps for privacy violations. https://shorturl.at/aftw5.

[5] Apple privacy manifest. https://developer.apple.com/documentation/bundleresources/privacy_manifest_files?language=objc.

[6] Apple store app privacy details. https://developer.apple.com/app-store/app-privacy-details.

[7] Appsflyer. https://www.appsflyer.com.

[8] Buglife. https://www.buglife.com.

[9] Charts sdk. https://github.com/danielgindi/Charts.

[10] Cocoapods. https://cocoapods.org.

[11] Countly. https://countly.com.

[12] DHS definition of data. https://shorturl.at/ceSVW.

[13] Ghidra p-code reference manual. https://shorturl.at/cdKO2.

[14] Ghidra software reverse engineering framework. https://github.com/NationalSecurityAgency/ghidra.

[15] Ksphotobrowser. https://github.com/skx926/KSPhotoBrowser.

[16] Lblaunchimagead. https://github.com/AllLuckly/LBLaunchImageAd.

[17] Malicious ios sdk breaches user privacy for millions. https://www.helpnetsecurity.com/2020/08/24/malicious-ios-sdk.

[18] Openmediation. https://www.openmediation.com/en.

[19] Programming with objective-c. https://shorturl.at/puEV9.

[20] react-native-revmob. https://github.com/RevMob/react-native-revmob.

[21] Sentry. https://sentry.io/welcome.

[22] Sleigh: A language for rapid processor specification. https://shorturl.at/fHIY0.

[23] Spacy dependency parser. https://spacy.io/api/dependencyparser.

[24] Superplayer. https://github.com/LiteAVSDK/Player_iOS/blob/master/README-EN.md.

[25] The swift call convention. https://github.com/apple/swift/blob/main/docs/ABI/CallingConvention.rst.

[26] The swift language. https://developer.apple.com/swift.

[27] Taking legal action against those who abuse our platform. https://shorturl.at/nvzP5.

[28] Teak. https://docs.teak.io/home/index.html.

[29] Technologies in apple' technologies documentation for developers. https://developer.apple.com/documentation/technologies.

[30] vk-ios-sdk. https://github.com/VKCOM/vk-ios-sdk.

[31] Xcode. https://developer.apple.com/xcode.

[32] App privacy details on the App Store, 2021. https://developer.apple.com/app-store/app-privacy-details.

[33] ktoo: Macho/objc analysis + editing toolkit. https://github.com/cxnder/ktool, 2022.

[34] CCPA, 2023. https://oag.ca.gov/privacy/ccpa.

[35] GDPR, 2023. https://gdpr-info.eu/chapter-1.

[36] iHunter, 2023. https://sites.google.com/view/ihunterios.

[37] Yuvraj Agarwal and Malcolm Hall. Protectmyprivacy: detecting and mitigating privacy leaks on ios devices using crowdsourcing. In Proceeding of the 11th annual international conference on Mobile systems, applications, and services, pages 97–110, 2013.

[38] ahmedbougacha. dagger: Binary translator to llvm ir. https://github.com/ahmedbougacha/dagger.

[39] Benjamin Andow, Samin Yaseer Mahmud, Wenyu Wang, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Tao Xie. {PolicyLint}: investigating internal privacy policy contradictions on google play. In 28th USENIX security symposium (USENIX security 19), pages 585–602, 2019.

[40] Benjamin Andow, Samin Yaseer Mahmud, Justin Whitaker, William Enck, Bradley Reaves, Kapil Singh, and Serge Egelman. Actions speak louder than words:{Entity-Sensitive} privacy policy and data flow analysis with {PoliCheck}. In 29th USENIX Security Symposium (USENIX Security 20), pages 985–1002, 2020.

[41] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. Acm Sigplan Notices, 49(6):259–269, 2014.

[42] Xiaolong Bai, Luyi Xing, Min Zheng, and Fuping Qu. idea: Static analysis on the security of apple kernel drivers. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020, pages 1185–1202. ACM, 2020.

[43] Duc Bui, Yuan Yao, Kang G Shin, Jong-Min Choi, and Junbum Shin. Consistency analysis of data-usage purposes in mobile apps. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 2824–2843, 2021.

[44] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In 2016 IEEE Symposium on Security and Privacy (SP), pages 357–376. IEEE, 2016.

[45] Yi Chen, Mingming Zha, Nan Zhang, Dandan Xu, Qianqian Zhao, Xuan Feng, Kan Yuan, Fnu Suya, Yuan Tian, and Kai Chen. Demystifying hidden privacy settings in mobile apps. In 2019 IEEE Symposium on Security and Privacy (SP), pages 570–586. IEEE, 2019.

[46] Jonathan Crussell, Ryan Stevens, and Hao Chen. Madfraud: investigating ad fraud in android applications. In Andrew T. Campbell, David Kotz, Landon P. Cox, and Zhuoqing Morley Mao, editors, The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014, pages 123–134. ACM, 2014.

[47] Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang, and Carl A. Gunter. Free for all! assessing user data exposure to advertising libraries on android. In 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society, 2016.

[48] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. iris: Vetting private API abuse in ios applications. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015, pages 44–56. ACM, 2015.

[49] Luke Deshotels, Costin Carabas, Jordan Beichler, Răzvan Deaconescu, and William Enck. Kobold: Evaluating decentralized access control for remote nsxpc methods on ios. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1056–1070. IEEE, 2020.

[50] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. In Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011. The Internet Society, 2011.

[51] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for

realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS), 32(2):1–29, 2014.

[52] Johannes Feichtner, David Missmann, and Raphael Spreitzer. Automated binary analysis on ios: A case study on cryptographic misuse in ios applications. In Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks, WiSec 2018, Stockholm, Sweden, June 18-20, 2018, pages 236–247. ACM, 2018.

[53] Denzil Ferreira, Vassilis Kostakos, Alastair R Beresford, Janne Lindqvist, and Anind K Dey. Securacy: an empirical investigation of android applications' network usage, privacy and security. In Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, pages 1–11, 2015.

[54] Jingyi Guo, Min Zheng, Yajin Zhou, Haoyu Wang, Lei Wu, Xiapu Luo, and Kui Ren. ilibscope: Reliable third-party library detection for ios mobile apps. arXiv preprint arXiv:2207.01837, 2022.

[55] Hamza Harkous, Kassem Fawaz, Rémi Lebret, Florian Schaub, Kang G Shin, and Karl Aberer. Polisis: Automated analysis and presentation of privacy policies using deep learning. In 27th USENIX Security Symposium (USENIX Security 18), pages 531–548, 2018.

[56] Hex-Rays. Ida pro: The best-of-breed binary code analysis tool. https://hex-rays.com/IDA-pro.

[57] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. The ART of app compartmentalization: Compiler-based library privilege separation on stock android. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017, pages 1037–1049. ACM, 2017.

[58] Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In Roderick Bloem and Natasha Sharygina, editors, Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23, pages 43–50. IEEE, 2010.

[59] Konrad Kollnig, Anastasia Shuba, Max Van Kleek, Reuben Binns, and Nigel Shadbolt. Goodbye tracking? impact of ios app tracking transparency and privacy labels. In Proceedings of the 2022 ACM Conference on Fairness, Accountability, and Transparency, pages 508–520, 2022.

[60] Malte Kraus and Vincent Haupert. The swift language from a reverse engineering perspective. In Proceedings of the 2nd Reversing and Offensive-oriented Trends Symposium, pages 1–12, 2018.

[61] Yeonjoon Lee, Xueqiang Wang, Kwangwuk Lee, Xiaojing Liao, XiaoFeng Wang, Tongxin Li, and Xianghang Mi. Understanding {iOS-based} crowdturfing through hidden {UI} analysis. In 28th USENIX Security Symposium (USENIX Security 19), pages 765–781, 2019.

[62] Tianshi Li, Kayla Reiman, Yuvraj Agarwal, Lorrie Faith Cranor, and Jason I. Hong. Understanding challenges for developers to create accurate privacy nutrition labels. In Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[63] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In Man Ho Au, Barbara Carminati, and C.-C. Jay Kuo, editors, Network and System Security - 8th International Conference, NSS 2014, Xi'an, China, October 15-17, 2014, Proceedings, volume 8792 of Lecture Notes in Computer Science, pages 349–362. Springer, 2014.

[64] Yucheng Li, Deyuan Chen, Tianshi Li, Yuvraj Agarwal, Lorrie Faith Cranor, and Jason I. Hong. Understanding ios privacy nutrition labels: An exploratory large-scale analysis of app store data. In Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. DECAF: detecting and characterizing ad fraud in mobile apps. In Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014, pages 57–70. USENIX Association, 2014.

[66] Kangjie Lu. Practical program modularization with type-based dependence analysis. In 2023 IEEE Symposium on Security and Privacy (SP), pages 1256–1270. IEEE, 2023.

[67] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps. In NDSS, 2018.

[68] Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan, and Mizuhito Ogawa. A hybrid approach for control flow graph construction from binary code. In Pornsiri Muenchaisri and Gregg Rothermel, editors, 20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 2, pages 159–164. IEEE Computer Society, 2013.

[69] David A. Ramos and Dawson R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In Ajay Gulati and Hakim Weatherspoon, editors, 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016. USENIX Association, 2016.

[70] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, Chrisĭtian Kreibich, and Phillipa Gill. Apps, trackers, privacy, and regulators: A global study of the mobile tracking ecosystem. In 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018, 2018.

[71] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEXDROID: enforcing in-app privilege separation in android. In 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society, 2016.

[72] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In IEEE Symposium on Security and Privacy, 2016.

[73] Rocky Slavin, Xiaoyin Wang, Mitra Bokaei Hosseini, James Hester, Ram Krishnan, Jaspreet Bhatia, Travis D Breaux, and Jianwei Niu. Toward a framework for detecting privacy policy violations in android application code. In Proceedings of the 38th International Conference on Software Engineering, pages 25–36, 2016.

[74] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. What mobile ads know about mobile users. In 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016, 2016.

[75] Zhushou Tang, Ke Tang, Minhui Xue, Yuan Tian, Sen Chen, Muhammad Ikram, Tielei Wang, and Haojin Zhu. {iOS}, your {OS}, everybody's {OS}: Vetting and analyzing network services of {iOS} applications. In 29th USENIX Security Symposium (USENIX Security 20), pages 2415–2432, 2020.

[76] Zhushou Tang, Ke Tang, Minhui Xue, Yuan Tian, Sen Chen, Muhammad Ikram, Tielei Wang, and Haojin Zhu. Ios, your os, everybody's os: Vetting and analyzing network services of ios applications. In Proceedings of the 29th USENIX Conference on Security Symposium, SEC'20, USA, 2020. USENIX Association.

[77] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015, pages 151–167. IEEE Computer Society, 2015.

[78] Eran Tromer and Roei Schuster. Droiddisintegrator: Intra-application information flow control in android apps. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS

2016, Xi'an, China, May 30 - June 3, 2016, pages 401–412. ACM, 2016.

[79] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society, 2018.

[80] Jice Wang, Yue Xiao, Xueqiang Wang, Yuhong Nan, Luyi Xing, Xiaojing Liao, Jinwei Dong, Nicolás Serrano, Haoran Lu, XiaoFeng Wang, and Yuqing Zhang. Understanding malicious cross-library data harvesting on android. In Michael Bailey and Rachel Greenstadt, editors, 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021, pages 4133–4150. USENIX Association, 2021.

[81] Xiaoyin Wang, Xue Qin, Mitra Bokaei Hosseini, Rocky Slavin, Travis D Breaux, and Jianwei Niu. Guileak: Tracing privacy policy claims on user input data for android applications. In Proceedings of the 40th International Conference on Software Engineering, pages 37–47, 2018.

[82] Yizhuo Wang, Yikun Hu, Xuangan Xiao, and Dawu Gu. iservice: Detecting and evaluating the impact of confused deputy problem in appleos. In Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022, pages 964–977. ACM, 2022.

[83] Tim Werthmann, Ralf Hund, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Psios: bring your own privacy & security to ios devices. In Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, pages 13–24, 2013.

[84] Yue Xiao, Zhengyi Li, Yue Qin, Xiaolong Bai, Jiale Guan, Xiaojing Liao, and Luyi Xing. Lalaine: Measuring and characterizing non-compliance of apple privacy labels at scale. In 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim Marriott, CA, USA, August 9-11, 2023. USENIX Association, 2023.

[85] Luyi Xing, Xiaolong Bai, Tongxin Li, XiaoFeng Wang, Kai Chen, Xiaojing Liao, Shi-Min Hu, and Xinhui Han. Cracking app isolation on apple: Unauthorized cross-app resource access on mac os x and ios. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, page 31–43, New York, NY, USA, 2015. Association for Computing Machinery.

[86] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 1043–1054, 2013.

[87] Le Yu, Xiapu Luo, Jiachi Chen, Hao Zhou, Tao Zhang, Henry Chang, and Hareton KN Leung. Ppchecker: Towards accessing the trustworthiness of android apps' privacy policies. IEEE Transactions on Software Engineering, 47(2):221–242, 2018.

[88] Le Yu, Xiapu Luo, Xule Liu, and Tao Zhang. Can we trust the privacy policies of android apps? In 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 538–549. IEEE, 2016.

[89] Xiaokuan Zhang, Xueqiang Wang, Xiaolong Bai, Yinqian Zhang, and Xiaofeng Wang. OS-level side channels without procfs: Exploring cross-app information leakage on iOS. In Proceedings 2018 Network and Distributed System Security Symposium. Internet Society, 2022.

[90] Min Zheng, Hui Xue, Yulong Zhang, Tao Wei, and John C. S. Lui. Enpublic apps: Security threats using ios enterprise and developer certificates. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015, pages 463–474. ACM, 2015.

[91] Sebastian Zimmeck, Rafael Goldstein, and David Baraka. Privacyflash pro: Automating privacy policy generation for mobile apps. In NDSS, 2021.

[92] Sebastian Zimmeck, Peter Story, et al. Maps: Scaling privacy compliance analysis to a million apps. Proceedings on Privacy Enhancing Technologies, 2019(3):66–86, 2019.

Appendix A Setup of Evaluation and Measurement

Building SDK test-bed apps. To perform a taint analysis on the SDKs, we integrated each SDK into an initially empty app to create a test bed, from which we subsequently extracted the compiled binaries. To expedite this process and facilitate batch building, we developed a script that utilizes the `pod install` command to integrate SDKs into the empty app and the `XcodeBuild` command to compile the project. Additionally, we added the `-ObjC` and `-all_load` linker options as compiler parameters to load all code and symbols within the SDKs into the built app:

Analysis of SDK integration. In general, SDKs are commonly integrated into apps through two primary methods: **(a)** static integration into the app's binary or **(b)** dynamic library integration as a *.dylib* file, which is stored in the Frameworks folder of the *.ipa* file. To identify the SDKs used in the apps, we propose a rapid approach that involves analyzing the binary and *.dylib* files of the apps. ***Step ❶:*** We first performed *class-dump* on all apps and SDKs to obtain information about the classes in the binaries, which is similar to the process used in the class structure recovery stage. ***Step ❷:*** Then we generated a hash for every class and recorded the classes defined and used in every binary to an *SQLite* database. ***Step ❸:*** Finally, we conducted a matching process between the classes in each app and the classes in the SDKs and calculated how many classes of each SDK were found in each app. If most of the classes of an SDK were found in an app, we concluded that this app used the SDK. In our evaluation, the threshold ratio for SDK matching could be set to 0.75 since the classes could change with SDK updates.

Compliance criteria. Table 5 lists the pertinent sections to which *iHunter* refers when conducting compliance checks under prominent privacy laws, including GDP, CCAP, as well as Terms of Service (ToS).

Appendix B Other Details of Taint Analysis

Loop extraction. We have implemented a loop detection algorithm based on topological sorting. This algorithm operates on the CFG of every function and employs a stack-based approach to identify nested loops. The topological sorting algorithm sorts the nodes of the CFG in topological order, which means that if node A is before node B in the ordering, then there is no path from B to A in the CFG. This enables the algorithm to identify the loops by detecting back edges in the CFG. To detect nested loops, we use a stack-based approach that helps to maintain the loop hierarchy. Nested

Table 5: Classification of Non-compliances

Non-Compliance Type	Examples of Laws, Regulations or Terms of Use
Absence of Privacy Policy	**Article 12 - GDPR**- Transparent information, communication, and modalities for the exercise of the rights of the data subject: The controller shall take appropriate measures to provide any information referred to in Articles 13 and 14 and any communication under Articles 15 to 22 and 34 relating to processing to the data subject in a concise, transparent, intelligible and easily accessible form, using clear and plain language. **Article 13 - GDPR**- Information to be provided where personal data are collected from the data subject: Where personal data relating to a data subject are collected from the data subject, the controller shall, at the time when personal data are obtained, provide the data subject with all of the following information...
	Section 1798.100(b) - CCPA - Right to Know About Personal Information Collected, Disclosed, or Sold: A business that collects a consumer's personal information shall, at or before the point of collection, inform consumers as to the categories of personal information to be collected and the purposes for which the categories of personal information shall be used. A business shall not collect additional categories of personal information or use personal information collected for additional purposes without providing the consumer with notice consistent with this section. **Section 1798.130(a)(5) - CCPA** - Initial Notice to Consumers: A business that collects personal information about a consumer shall inform consumers as to the categories of personal information to be collected and the purposes for which the categories of personal information shall be used, which shall be provided at or before the point of collection.
Inappropriate Disclosure in Privacy Policy	**Article 5(1)(a) - GDPR**: "Personal data shall be processed lawfully, fairly and in a transparent manner ('lawfulness, fairness, and transparency')."
	Section 1798.100(b) - CCPA "A business shall not collect additional categories of personal information or use personal information collected for additional purposes without providing the consumer with notice consistent with this section."
Transgression of ToS Provisions	**Facebook Terms(3)(2)**: "Prohibition of proxying, requesting, collecting Product usernames, passwords, or misappropriation of access tokens."
	Facebook Terms(2)(1) "Obtain consent from people before publishing content or taking any other action on their behalf."

loops are pushed to the top of the stack and popped when the algorithm finishes analyzing a loop.

Recovery of Swift mangled calls. In Swift binaries, function call names are mangled into unique and indecipherable strings. For example, when initializing a Data instance using the `Foundation.Data.init(referencing:NSData)` function within the Swift Foundation Framework, the corresponding call in the compiled binary is transformed into `s10Foundation4DataV11referencingACSo6NSDataCh_tcfC` after being mangled. To overcome this issue, we developed a Swift function demangling tool capable of extracting all the relevant information about a Swift function call, such as the function name, the indexing path of the function (i.e., module and class name), parameter names, and parameter and return value types. By demangling the name of the `Data` initializer function, we obtained `Foundation.Data.init(referencing: __shared __C.NSData)-> Foundation.Data`, which provides more information than the Objective-C type message-send calling. Furthermore, we can utilize the additional information on the return and parameter types to enhance type inference and assist in resolving message sending in the program.

Taint propagation. During the under-constrained symbolic execution stage, we extract and retain intra-procedural data flow in the summaries. Consequently, propagating taint across the entire iOS binary no longer poses significant challenges. Our approach involves identifying all call sites of each source and sink, marking them as the starting and ending points of taint propagation, and subsequently traversing to construct a taint graph. From this graph, we can extract taint traces by addressing a classical graph reachability problem.

To generate the taint graph, *iHunter* conducts forward propagation within the virtual inter-DFG. This means that propagation occurs within each function's intra-DFG, with context switching when encountering control flow branches or function calls/returns. Propagation starts from the call sites of sources and terminates at the call sites of sinks. To expedite the process, we record all traversed paths and mark them as terminated nodes when traversal concludes. To extract all paths from the taint graph, we employ the flooding algorithm, recording which sources can reach every point and generating all traces for each (n_{source}, n_{sink}) pair.