國立政治大學資訊管理學研究所

碩士學位論文

使用字串分析揭露 iOS 執行檔之動態載入類別

Uncovering dynamically loaded classes of iOS executables with static string analysis

指導教授：郁 方 博士

研究生：林 君 翰 撰

中 華 民 國 一〇六 年 七月

Abstract

Millions of mobile apps have been published in Apple's AppStore with more than 15 billion downloads by iOS devices. In order to protect iOS users from malicious apps, Apple has strict policies which are used to eliminate apps before they can be published in the AppStore. In this paper we present a string analysis technique for iOS executables for statically checking policies that are related to dynamically loaded classes. In order to check that an app conforms to such a policy, it is necessary to determine the possible string values for the class name parameters of the functions that dynamically load classes. The first step of our approach is to construct the assembly for iOS executables using existing tools. We then extract flow information from the assembly code and construct control flow graphs (CFGs) of functions. We identify functions that dynamically load classes, and, for each parameter that corresponds to a dynamically loaded class, we construct a dependency graph that shows the set of values that flow to that parameter. Finally, we conduct string analysis on these dependency graphs to determine all potential string values that these parameters can take, which identifies the set of dynamically loaded classes. Taking the intersection of these values with patterns that characterize Apple's app policies (such as private/sensitive APIs), we are able to detect potential policy violations. We analyzed more than 1300 popular apps from Apple's AppStore and checked them against Apple's policy about the use of private APIs and the identifier for Advertising (IDFA). Our tool extracted more than 37000 string dependency graphs from these applications and our analysis reported 208 apps that compose the corresponding API with strings and have potential IDFA violations. Our analysis also found 372 apps that could have compose the private class name with string and 236 apps that could have load the private framework with path string; and could violate the private API usage policy.

Contents

List of Figures

List of Tables

1 Introduction

Mobile applications on iOS and Android platforms have been increasing remarkably in the past decade and have become the dominant software applications in history. According to the latest investigations statistics [1], android users are able to choose over 1.6 million apps on Google play, the official android applications market and iOS users can select over 1.5 million apps on Apple's App Store. In the last year annual Apple conference, WWDC 2015, Tim Cook announced that the App Store has crossed 100 billion app downloads.

Due to the explosive growth of smart mobile devices and applications, the number of malicious software on mobile platforms has kept increasing in recent years. The mobile software analysis report from Germany [2] showed that from January 2015 to September 2015, the number of the found mobile malware has crossed the number of malware in the whole 2014 year. The report from McAfee [3] also indicated a rapid growth rate in mobile malware.

One common malware behavior is to access users' private information improperly such as address book, calendar, and photos. Since these behaviors are legal and necessary under certain conditions, it is needed flow analysis to determine whether these sensitive data are properly used by applications. One famous malicious iOS application is named PATH [4, 5] that records users' locations and paths and uploads their address books to a third party remote server without authorization from users. Another example is the app called YourName that is used to edit photos with fancy transformations. It was caught uploading all the photos in the library upon installation. Note that both PATH and YourName were public available from Apple app store and were quite popular and widely recommended by many users before their malicious behaviors were revealed.

There are also other sensitive behaviors that may raise users concerns. Mobile applications with the third party SDKs can access user data from their official web services, e.g., Facebook offers iOS SDK to developers to design access functionalities around Facebook. Using such SDK implies the access of user's Facebook account and information. Hence, it is needed to trace the usage flow of the third party SDK of mobile applications. On the other hand, mobile ads play an important role in creating revenue for developers. In 2013, mobile revenue had reached over 17 billions [6], and analysts reported that revenue from mobile advertisements would have surpassed television advertising in 2017 [7]. However, those ads may also cause negative effects to users [8]. While Apple has a strict policy on the advertisement framework usage, there are example apps [9] that can bypass the App review checking. If an app contains no advertisement but accessing IDFA, the app should be rejected by Apple before publishing to the public. Last but not the least, private APIs are designed for internal usage of Apple developers, e.g., directly access sensitive data without authentications. These APIs are strictly forbidden to be used by external developers. Recent researches have shown that there were apps using private APIs but published in Apple app store [9].

The above behaviors may not show out in app descriptions or user interface. They could be triggered under certain conditions and could not be observed by users. In order to protect iOS users from malicious apps, Apple has strict policies which are used to eliminate apps before they can be published in the Apple app store. Many of them shall be prevented by the enforcement of the security development policy in the reviewing process of Apple. App review is the censoring process to all submitted apps that Apple conducts to determine whether they are reliable, perform as expected, and are free of offensive materials. Still, there are quite a few examples showing that malicious apps may bypass the check and are able to be published to public users. For these cases, users who rely on Apple blackbox check would be unaware of malicious behaviors until (serious) damages are observed. One main reason that app developers may bypass apple review process is using the dynamic loaded classes. In iOS programming, Objective-C allows programmers to load the class at runtime instead of static compilation. This pattern provides performance benefits such as delay loading the code until it is needed and increase the code modularity.

Listing 1: Load a Class Dynamically

```
NSBundle *b = [NSBundle bundleWithPath:@"/System/Library/Frameworks/
        AdSupport.framework"];
```

```
[b load];
Class c = NSClassFromString(@"ASIdentifierManager");
id si = [c valueForKey:@"sharedManager"];
}
```

Listing 2: Load a Class with String Computations

```
//...
NSString *name = [NSString stringWithFormat:@"%c%c%c%c%c%c%c%c%c%c%c%c%c%c%
        c%c%c%c%c",'A','S','I','d','e','n','t','i','f','i','e','r','M','a','n
        ','a','g','e','r'];
Class c = NSClassFromString(name);
//...
```

Let's show an example to bypass the checking of IDFA abuse. We start from a simple code in listing 1 that would be caught be Apple app review for loading the ASIdentifierManager class with the class name (a string value) by calling a C-function called NSClassFromString. After the class was loaded, it then gets the value of a static field named sharedManager to access users' private information. Note that the class is loaded dynamically via the NSClassFromString function. While the loaded class depends on the value of the parameter that can be manipulated through string operations, the dynamism could lead to a loophole on Apple's app review. In fact, one could load the class ASIdentifierManager without having any class associated with ASIdentifierManager at compile time, and bypass the check on IDFA abuse. (Apple requires developers to use the ASIdentifierManager class only for serving advertisements purpose. This requirement is also known as one of the most common reasons that cause iOS apps been rejected.)

For instance, listing 2 is a modified version of dynamically loading the ASIdentifierManager class. Note that in this version, the parameter of NSClassFromString is no longer a literal but a string variable called name. As the listing shows, the value of name is synthesized at runtime via concatenating 19 characters (by calling stringWithFormat function in NSString). In this case, searching constants appearing in binary [10][11] would find separated characters instead of the correct class ASIdentifierManager associated with the app. The way to dynamically load classes has added a loophole for developers to cheat the App Review process, hiding their behaviors via loading classes dynamically. Dynamic analysis [9] that observes the executions of iOS applications can reveal the runtime behavior when the class is loaded at runtime.

However, while the loaded classes depend on the values of string variables, there are various ways to raise obfuscation to uncover the loaded classes and invoked methods such as to manipulate values with advanced string operations such as replacement, to compose string operations with branch and loop structures, and to use external calls to get values from Internet or user inputs. Even if the instrumentation environment [9] is feasible, dynamic analysis that depends on observation of executions requires high coverage to witness malicious executions. While it is good for bug hunting, it is not sufficient to claim that the loaded classes are not involved inappropriate ones.

Listing 3 is a code snippet which embed a back-door program into an app to dynamically load any private class. The app will first load 3 payload strings from our remote server and check if the first payload string $p1$ is equals to "fire". The dynamic class loading process will be trigger if and only if the $p1$ is equals to "fire". Once the process is triggered, we do a string replacement operation on the second payload which will replace all the "x" with "p" in string $p2$. The replacement result will become the name of the dynamically loaded class. For example, if $p2$ is equals to "FTDeviceSuxxort" then manipulated result string will be "FTDeviceSupport" and FTDeviceSupport is one of the private classes which Apple do not allow developers to use if the app is intend to be publish to app store.

We have successfully embed the back-door code into an app and have passed the Apple's App Review. It is impossible for a dynamic analysis approach to discover this malicious behavior if the server never response a payload $p1$ as "fire". Also, it is impossible for a static analysis such as constant propagation to discover this malicious behavior because the class name never appears in the app's own binary. Further more, since we try to obfuscate the class name by doing a string replace manipulation, it is in vain to do filtering on all the external payload such as censoring all the network traffic because the dynamically loaded class's name never appears in the payload as well.

Listing 3: Load a Class Dynamically with External payload

```
NSString *p1 = a payload string which response from a remote server;
NSString *p2 = a payload string which response from a remote server;
NSString *p3 = a payload string which response from a remote server;
if([p1 isEqualToString:@"fire"])
{
  //...
  NSBundle *b = [NSBundle bundleWithPath:p2];
  [b load];
  NSString* name = [p3 stringByReplacingOccurrencesOfString:@"x" withString
        :@"p"];
  Class c = NSClassFromString(name);
  //...
}
```

The objective of this work is to provide a sound static analysis for systematic violation checking of iOS mobile applications. One key feature is the capability of characterizing dynamic loaded classes and invoked methods in iOS mobile applications such that all the potential policy violations can be detected. Specifically, we present flow and string analysis techniques for iOS executables for statically checking policies that are related to dynamically loaded classes. In order to check that an app conforms to such a policy, it is necessary to determine the possible string values for the class name parameters of the functions that dynamically load classes. The first step of our approach is to construct the assembly for iOS executables using existing tools. We then extract flow information from the assembly code and construct control flow graphs (CFGs) of functions. We identify functions that dynamically load classes, and, for each parameter that corresponds to a dynamically loaded class, we construct a dependency graph that shows the set of values that flow to that parameter. Finally, we conduct string analysis on these dependency graphs to determine all potential string values that these parameters can take, which identifies the set of dynamically loaded classes. Taking the intersection of these values with patterns that characterize Apple's app policies (such as private/sensitive APIs), we are able to detect potential policy violations.

Most malicious behaviors and violations of security policies, such as IDFA abuse and private API usage, are related to the loaded classes and invoked methods of mobile applications. However, due to the flexibility of Objective-C (actually most modern programing languages, such as PHP, Java reflections), developers can use string variables to load classes and invoke methods dynamically. This hinders the hurdle of effective program analysis and verification of system properties and detection of potential policy violations. Furthermore, mobile applications downloaded as executables are not available with source codes. This requires significant work to rebuild program flows at the assembly level. The closed system nature makes the iOS mobile applications even harder to be analyzed. Technically, this is the first work that integrates string analysis with flow analysis to resolve dynamic loaded classes of iOS mobile applications. We make the contributions on static binary flow analysis where we propose a context-aware flow graph construction that resolves registers for indirect jumps for inter procedure calls, and contribution on static string analysis where we propose parameter-aware dependency graph construction that resolves parameter values of functions with automata-based string analysis. In practice, we develop an end to end tool for policy violation detection of iOS applications.

2 Related work

Mobile malicious behaviors have been studied intensely in the past years. Most previous research and analysis tools target on droid applications, such as TaintDroid [12], AsDroid [13], FlowDroid [14], DroidRA [15] and Checker [16]. A large set of android application benchmarks has also been collected for study [17].

Many of these tools target on privacy leakage that Felt et al. [18] summarized such threats as to leak sensitive information about a user to an unknown source. A common static analysis to detect privacy leakage is using taint analysis approaches where sensitive data are marked as tainted and propagated during program computations. The taint analysis detects a violation whenever tainted data (a secret value) flow to an untainted tunnel (e.g., a an external function), and hence showing potential data leaks in apps. Flowdroid [14] is the typical static taint analysis tool for privacy leakage detection of android applications. It builds precise control flow graph with API calls labeled. It also collects a large data set of android applications as DROIDBENCH, which is the current most referred data set for analyzing android applications. However, their control flow graph constructions overlook dynamic loaded classes (that are invoked by java reflection in android applications) as we defined in this work and hence malicious behaviors may bypass the check via manipulating string variables. To address this issue, DroidRA [15] adopts constant propagation solver to infer the possible reflective calls in android applications. The limitation of using constant propagation is that they do not support fixpoint computation and advanced string operations such as replacement. Our proposed approach integrating string analysis with flow analysis can be applied to the Javabyte code in a similar manner to reveal the dynamic loaded classes and is able to deal with complicated program structures that DoidRA cannot deal with. Similar to DroidRA, Checker [16] framework adopts static analysis on constructing implicit control flow which causes by Java reflection and Android intent to aid developer checking the data-flow of applications. However, they required developers to use Java annotation to conduct the downstream static analysis, thus it can only be used in an invasive way to analyze annotated code and limit the ability to do security verification on the real-world android applications in the market.

An earlier work proposed by Mann and Starostin [19] defines the static analysis framework to detect leakage of privacy data in android applications. They sorted out private information into five categories: ”location data”, ”unique identifiers”, ”call state”, ”authentication data”, and ”contact and calendar data”, and put signatures on the methods and parameters of the methods that could be used to extract and transmit private information. However, their framework was also restricted to track explicit information flow.

Unlike our work reveals dynamic loaded classes and methods, the leakage of user's privacy data via dynamic loaded classes and methods are overlooked. For other kinds of property checking, AsDroid [13] detects whether there are "stealthy" behaviors in android applications by finding contradictions between user interface and program behaviors. The proposed work is orthogonal to such checking by providing precise program behaviors. Other than analyzing programs, Zhou et al. [20] proposed permission-based behavioral footprinting by extracting information in the manifest file of the apps to find the requested permissions. We extract information from meta data to build control flow graphs for a much more precise analysis.

Dynamic analysis that observes executions of applications poses an attractive solution for property checking, where the dynamic loaded classes and invoked methods are revealed and can be observed during execution. One typical dynamic analysis tool is TaintDroid [12]. TaintDroid automatically labels privacy-sensitive data, and propagates that label through files and variables. TaintDroid keeps a record of the label, responsible application and the destination for each transmission to external targets. Barbic et al. [21] draw system call dependency graphs by tracing program executions and log system calls. Dynamic analysis is limited to executions that are observed and is challenged on text input degeneration and coverage improvement. As we have stated, developers may hide malicious behaviors, e.g., they are triggered only for specific inputs or under specific circumstance. On the other hand dynamic analysis is good for bug hunting but incapable of proving system correctness. In practice, unlike android applications have the well-established execution platform to observe application behaviors, iOS applications do not have such a platform provided by Apple. This limits the feasibility for dynamic analysis on iOS applications.

Static analysis techniques for iOS applications have been proposed in relatively less research compared to android applications. PiOS is the pioneer work proposed by Egele et al. [22]. It is the first static binary analysis tool that could analyze iOS applications, and automatically determine if these applications would leak user's privacy data. Similar to the proposed approach, they decrypted binary of iOS applications, disassembled the binary, and built the control flow graph annotated with method calls, and detect data flow analysis to find out potential privacy leaks. They adopt constant propagations for dynamic loaded classes. We enhance their work in binary flow analysis by applying string analysis to resolve values of registers for reconstructing dependency graphs and characterizing values of parameters of sensitive functions. They evaluated their approach against more than 1,400 iPhone applications and showed the feasibility of binary analysis for iOS applications. However, there is no tool and data available to the public. After PiOS proposing, Werthmann, Hund, Davi, Sadeghi, and Holz [werthmann2013psios] proposed PSiOS, a new framework for privacy data security. PSiOS develops their own Objective-C static analyzer which can extract all relevant Objective-C structures that contain information about classes, methods, and inheritance relationships from Mach-O files and they leverage MoCFI [davi2012mocfi] to enforce control flow integrity dynamically on iOS devices running on ARM processors. iRiS proposed by Denget al. [9] adopts a hybrid approach that integrates static analysis and dynamic analysis. iRiS first employs the static flow analysis similar to PiOS for potential privacy leakage detection, but further facilitates dynamic analysis of iOS applications to resolve unsolvable calls and to witness violations. They added tense extension to the framework Valgrind [23] to initiate this very first dynamic analysis environment of iOS applications. We adopt string analysis to characterize all potential values instead of finding ones.

Yu et al. have proposed static binary analysis in AppBeach [yu2013appbeach, yu2014appbeach] and AppReco [11], where they scan all the constants in the assembly. The constants that match class and method names reveal potential loaded classes and invoked methods. While the class/method names are composed by characters, it is possible that the complete class name has never appeared in the executable and hence AppBeach/AppReco may not be able to identify all loaded classes (have false negatives); on the other hand, AppBeach/AppReco considers only the syntax appearance and order but overlook the actual program flows. This may raise false positives. In this work, we address these issues by investigating flow analysis techniques and string analysis techniques and improve the precision of the analysis significantly.

Finally, previous static binary code analysis have been studied with the focus on handling function entries and boundaries [24][25], and indirect control flows [26], and various tool kits [27][28], [29] have been provided to help users automate the binary analysis. Bitblaze [30] on the other hand provides a mixed platform that contains both dynamic and static analysis components for binary codes. Bitblaze is particularly useful for malware detection with verification tools integrated. We propose to incorporate the commercial tool IDAPro [28] with our analysis as the first step to decode binary bytes into machine instructions. Previous work that addresses complex code structures can be integrated and used to improve the precision of our analysis.

String analysis has been widely studied and applied in web application security. Java String Analyzer (JSA) [31] is grammar based string analysis tool for Java which can be used to detect various Web application errors [32][33][34][35]. Some state-of-art string constraint solvers [36][37][38][39][40] are also adopted in string property checking using a decision procedure on string equation with string length constrain supported. Since these string constraint solvers provided bounded analysis, they cannot produce sound result. SMT-based solver such as CVC4 [37], Z3-Str [38] and NORN [36] provided unbounded strings and integers constraint solving. Yu et al. have been working on automata-based string analysis [41][42][43] and release several tools such as Stranger [43] and Slog [44]. The string analysis techniques have been applied to web vulnerability detection and patch synthesis [45][46]. The techniques are orthogonal to our approach and can be used to integrate to improve precision and scalability in our string analysis phase. As far as we know, this is the first work to apply string analysis techniques to resolve dynamic loaded classed and invoked methods in mobile applications.

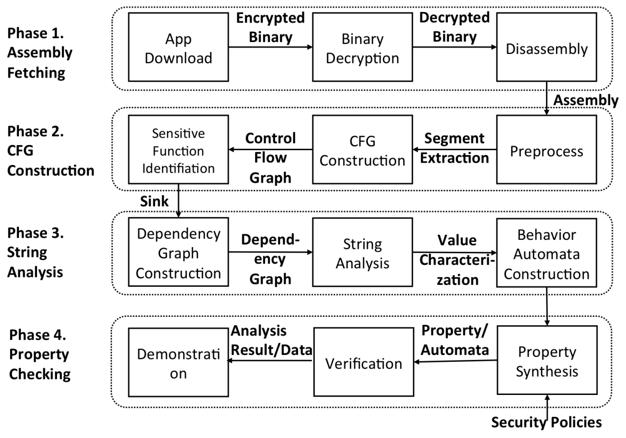


Figure 1: App Analysis Architecture

3 Overview

Our property verification of apps consists of four phases: 1) app fetching and decryption, 2) segment information extraction and control flow graph (CFG) construction, 3) dependency graph construction and string analysis, and 4) property synthesis and verification. Figure 1 shows the framework of the analysis processes.

In phase 1, we first download and install online apps from apples app store into a jail-broken iOS device, where we can access its file system directly to fetch the target binary. The binary is encrypted and is decrypted by the device with authentication upon execution. To decrypt the binary, one can insert break points right before the execution [10], or apply the third party binary decryption tool [29] to generate the decrypted binary. The decrypted binary can then be analyzed with disassembler tools such as IDA pro [28]. We plan to use the IDA tool to yield the plain text format assembly code. Note that an iOS app's binary is an Mach Object (Mach-O), and its assembly split into multiple segments contain various meta information such as subroutine entries, external calls, constant strings, mapping tables, etc, in addition to the assembly body of its subroutines.

In phase 2, we extract needed information from assembly segments first. Our goal is to construct the control flow graph (CFG) for each sub routine, and resolve register values of indirect jumps to link these routines. During the CFG construction, we also mark dependency relations of registers for each assembly statement. To identify sensitive functions, we find call-external-C-function-node or call-external-method-node and resolve their register values to identify which ones are relevant to the target (sensitive) function. When a sensitive function is identified, we then build the dependency graphs for its parameters. This can be done by traversing the dependency relations from the corresponding register (sink) backwardly up to constants or external inputs.

In phase 3, for each sink, we build its string dependency graph that specifies how input values flow to the sink. The sink values define the values of the parameters of target functions. For each dependency graph, we conduct forward string analysis on the graph to characterize all potential values of the sink node. We adopt automata-based string analysis where the automata associated with the sink node accepts all possible values of the sink node. We start from constants and arbitrary values of external inputs and manipulate string operations along with automata constructions until a fixpoint has been reached at the sink node of the dependency graph. The automata are then used to determine all the dynamic loaded classes and invoked methods.

In phase 4, we formalize security policies, such as IDFA abusing or private API usages, as properties on flows of loaded classes and invoked methods, and formalize app behaviors on policy-related classes and methods as behavior automata. We can then check property violation via formal verification.

3.1 A Quick Example

The sample code of IDFA abusing we introduced in prior section is in Objective-C source code. However, what we download from the App Store is App's binary. Therefore, the actual input for our analysis are against assembly codes. The simplified assembly code after compiling source code in Listing 2 are listed in the "Insturction" column of table 1,2,3,4,5,6,7 and 8. To increase the comprehension of those assembly code we reordered instructions of the assembly code by the program slicing of each register or memory address we interested in. A program slicing of variable x is a sequence of code which relevant to the value of x. The real order for the assembly instructions can be derived from address number in "Addr." column in each of the tables. Also, the pseudo code for each instruction are provided in "Pseudo Code" column in a C-like format. In the pseudo code, we use a array variable M as the the data structure of memory. We use $M[x] = y$ denote storing value y into the memory addressed at x, $y = M[x]$ denote loading value into variable y from memory address x.

In this subsection, we will walk through the process how we verify if an App has the problem of IDFA abusing or not from the assembly code in those assembly code tables.

We will have to identify the sink in the assembly first. A sink is the parameter value that passed into the sensitive function like $NSClassFromString$. Since $NSClassFromString$ only take one parameter, our sink here in the example is the value of first parameter passed into $NSClassFromString$ function. Once the position of sink is identified, we will construct the string dependency graph for the value of sink. Figure 2 shows the example of the completed dependency graph for sink of IDFA abusing sample code in Listing 2. A dependency graph indicate how the sink value is synthesized before passed into the sensitive function. We can tell from Figure 2 that the sink value is depends on the result of a concatenation operation. And that concatenation result is the string value after concatenating 18 individual characters: $A, S, I, d, e, n, t, i, f, e, r, M, a, n, a, g, e, r$.

3.1.1 Construct dependency graph from assembly

After knowing the purpose of constructing the dependency graph, in this subsection, we are going to introduce the high level concept of how we construct the dependency graph from assembly code in table 1,2,3,4,5,6,7 and 8. The $NSClassFromString$ function call is located at the table 1 where the BLX instruction cause the program to jump to the subroutine stub of $NSClassFromString$. Since ARM assembler will have the 1st to 4th parameter stored in register R0 to R3 individually before conducting a function call, the sink value (the 1st parameter) of $NSClassFromString$ must currently stored in R0. To derived value of R0, we will need to do a backward slicing searching on R0's program slicing. From table 2 where the slicing of R0 is listed, we can tell from the pseudo code that the value of R0 is depends on the value in $M[SP + \#0xE0 + var\_38]$. Continuing searching backward in the slicing of R0, we known that $M[SP + \#0xE0 + var\_38]$ further depends on R0 again (see instruction 8682). And R0's value, as instruction 867A in table 2 shows, is depends on the return value from calling a function because the ARM assembler will have any return value of a function call stored into R0. However, the problem we face in instruction 867A is that we must resolve what exactly is the called target function stored in R12.

Table 1: Assembly Code: Calling NSClassFromString

Addr.	Instruction	Pseudo Code
8686	BLX _NSClassFromString	R0 = NSClassFromString(R0)

Table 2: Assembly Code: Slicing of R0 (Param#1 of NSClassFromString)

Addr.	Instruction	Pseudo Code
867A	BLX R12	R0 = call R12 ⇒ objc_msgSend(R0,R1,R2,R3, M[SP+#0xE0+var_E0],...) ⇒ stringWithFomrat(R2,R3, M[SP+#0xE0+var_E0],...)
8682	STR R0, [SP,#0xE0+var_38]	M [SP,#0xE0+var_38] = R0
8684	LDR R0, [SP,#0xE0+var_38]	R0 = M [SP,#0xE0+var_38]

Observing the backward slicing of R12 in table 3, we can tell that R12 is equals to $M[SP + \#0xE0 + var\_74]$ (instruction 8676) and that $M[SP + \#0xE0 + var\_74]$ is further equals to R0 (see instruction 85FC). The value of R0, according to instruction 85FA, should equals to the value of $M[R0]$. From instruction 85F8 and instruction 85F0, we know that the value of R0 in instruction 85FA should be $\_objc\_msgSend\_ptr\_0 - 0x85FC + PC$. The evaluated value for this expression is actually computing a physical address for LDR instruction in instruction 85FA. Since IDA Pro has already computed a virtual address $\_objc\_msgSend\_ptr\_0$ for that physical address for static data in assembly, we can derived the actual value by just using that virtual address without really executing the assembly. By looking up the data segment in assembly, we know what instruction 85F8 is trying to load from the memory is actually an function called *objc_msgSend*. The *objc_msgSend* function plays a role as an Objective-C method invoker. In Objective-C, a method is invoked by message sending and *objc_msgSend* simply do the task of sending that message. When doing message sending, there are three key elements: receiver, selector, and sent data. Receiver is also known as instance in Object-Oriented (OO)

Programming. When invoking an instance method, the receiver is that instance object; when invoking an class method (or static method), the receiver is the class-instance (meta-instance) of that class. Selector is the method name of invoked method. Sent data are parameters values that need to pass into the method while invoking it. The receiver is the first parameter of *objc_msgSend* function, so it is stored in R0; the selector is the second parameter, so it is stored in R1. The 3rd and 4th parameter is for storing 1st and 2nd sent data (1st and 2nd parameter for invoked method). If invoked method has more then 2 parameters, they will be sored into stack memory and the detail rule for this will be introduced in later section.

Table 3: Assembly Code: Slicing of R12

Addr.	Instruction	Pseudo Code
85F0	MOV R0, #(_objc_msgSend_ptr_0 - 0x85FC)	R0 = _objc_msgSend_ptr_0 - 0x85FC
85F8	ADD R0, PC	R0 = R0 + PC
85FA	LDR R0, [R0]	R0 = M[R0]
85FC	STR R0, [SP,#0xE0+var_74]	M [SP+#0xE0+var_74] = R0
8676	LDR.W R12, [SP,#0xE0+var_74]	R12 = M [SP+#0xE0+var_74]

Now that we know R12 is *objc_msgSend*, so the return value which will be stored into R0 in instruction 867A depends on which exactly the method will *objc_msgSend* invoke. To answer this question, we must resolved the receiver in R0 and selector in R1. By backward searching in the slicing of R0 (Receiver) in table 4, we know the value of R0 can be fetch from assembly data segment using virtual address $classRef\_NSString$, and that is the class-instance of class $NSString$ (the high-level string type in Objective-C). By searching the slicing of R1 (Selector) in table 5, we can tell that the value of R1 can be fetch from assembly data segment by using virtual address $selRef\_stringWithFormat\_$, and that is, a method name "$stringWithFormat:$". Since the receiver and selector is now clear, we know the invoked method is the static method called $stringWithFormat$ in $NSString$ class. $NSString$'s $stringWithFormat$ method can be used to generate "printf-like" formatted string value giving a formatter string and a argument list. For example, given a formatter string "%c%c" and argument list $'i','s'$, it will return "is". (Objective-C method call $[NSStringstringWithFromat:@"\%c\%c",'i','s']$ will return "is")

Now we know $stringWithFormat$ is going to be invoked and its return value is going

Table 4: Assembly Code: Slicing of R0 (Receiver)

Addr.	Instruction	Pseudo Code
860A	MOV R0, #(classRef_NSString - 0x8616)	R0 = classRef_NSString - 0x8616
8612	ADD R0, PC	R0 = R0 + PC
8614	LDR R0, [R0]	R0 = M[R0]
8616	STR R0, [SP,#0xE0+var_7C]	M [SP,#0xE0+var_7C] = R0
861E	LDR R0, [SP,#0xE0+var_7C]	R0 = M [SP,#0xE0+var_7C]

Table 5: Assembly Code: Slicing of: R1 (Selector)

Addr.	Instruction	Pseudo Code
85FE	MOV R0, #(selRef_stringWithFormat_ - 0x860A)	R0 = selRef_stringWithFormat_ - 0x860A
8606	ADD R0, PC	R0 = R0 + PC
8608	STR R0, [SP,#0xE0+var_78]	M[SP+#0xE0+var_78] = R0
8618	LDR R0, [SP,#0xE0+var_78]	R0 = M [SP+#0xE0+var_78]
861A	LDR R0, [R0]	R0 = M[R0]
861C	STR R0, [SP,#0xE0+var_80]	M[SP+#0xE0+var_80] = R0
8622	LDR R1, [SP,#0xE0+var_80]	R1 = M[SP+#0xE0+var_80]

to be stored in R0 at instruction 867A (in table 2) and this returned value is also the sink's value. However, to figure out how this sink value is synthesized, we still need to find out the value of formatter string and value in argument list. The formatter string is the first parameter of $stringWithFormat$ (the 3rd parameter of $objc\_msgSend$ function), so it is stored in R2. From the slicing of R2 in table 6, we can tell that R2's value can be looked up from assembly data segment by using virtual address $cfstr\_CCCCCCCCCCCC$, and the value is a string "%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c". From this formatter string, we can get a very important information for our dependency graph, and that is, the sink value is depends on a concatenation operation among 18 characters. That's why we have a concatenation-node as sink-node's predecessor in Figure 2.

Table 6: Assembly Code: Slicing of R2 (Param#1 of stringWithFormat)

Addr.	Instruction	Pseudo Code
85BC	MOV R0, #(cfstr_CCCCCCCCCCCCCC - 0x85C)	R0 = cfstr_CCCCCCCCCCCCCC - 0x85C
85C4	ADD R0, PC	R0 = R0 + PC
85EA	STR R0, [SP,#0xE0+var_6C]	M[SP+#0xE0+var_6C] = R0
8626	LDR R2, [SP,#0xE0+var_6C]	R2 = M [SP+#0xE0+var_6C]

To complete rest part of the dependency graph, we must know what exactly those concatenated 18 characters are. The first character is the first item in formatting argument list, and it is the 2nd parameter of $stringWithFormat$ (4th parameter of $objc\_msgSend$). So the first character must be stored in R3 right before invoking the $stringWithFormat$ method. By looking up R3's slicing in table 7, we can tell that the first character value is 'A'; that's why we have literal-node 'A' as concatenation-node's first predecessor in the dependency graph. In the case of the second character, it is the second item in formatting argument list, and it is the 3rd parameter of $stringWithFormat$. As mentioned, the 3rd parameter (sent data) should be stored in the stack memory, in this example, the memory address is $[SP, \#0xE0 + var\_E0]$. By backward searching the slicing of, $[SP, \#0xE0 + var\_E0]$, we can tell that its value is character 'S'. That's why we have literal-node 'S' as concatenation node's second predecessor in the dependency graph.

Table 7: Assembly Code: Slicing R3 (Parma#2 of stringWithFormat)

Addr.	Instruction	Pseudo Code
85C6	MOVS R3, #0x41	R3 = 'A'

The value of rest character can be derived in the similar way, and all the remain predecessor of concatenation node will be constructed after that. Once those literal-nodes are construct, our dependency graph (Figure 2) is completed.

3.1.2 Forward analysis on dependency graph

The dependency graph disclose that the sink value is actually synthesize by concatenating various characters. A dependency graph is a directive graph. Every nodes on the dependency graph denotes a string expression and every edge on the dependency graph shows the dependency relation, that is, the string expression of a node is depends on its successor string expressions. Therefore, in the case of 2, we can say that the sink's string expression is depends on the result after the concatenation, and the result of the concatenation is further depends on those literal string expressions $(A, S, I, d, ..., g, e, r)$. On the dependency graph, there are four kinds of nodes: sink-node, operation-node, literal-node and arbitrary-node. We use a sink-node to denote sink's string expression, use operation-node to denote doing a string operation on one or more string expressions such as doing a concatenation. As for literal-node, we use it to denote the literal string expression such as static string or character. Finally, we use arbitrary-node to denote arbitrary string expression. Once the dependency graph is constructed we will do a forward analysis on the graph and verify if the sink string expression statisfiy property which relate to IDFA abusing or other undesired pattern. We use symbolic finite automata in the forward analysis to verify sink's property. In forward analysis, we first defined some pattern

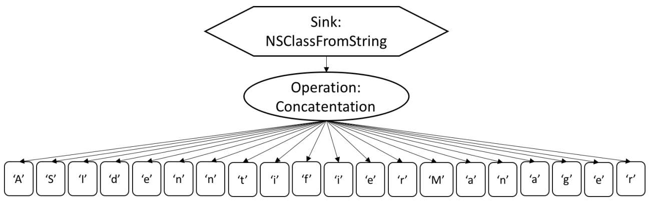


Figure 2: Synthesize class name by characters

DFA M_P which accept all suspicious string such as "$ASIdentifierManager$" for IDFA abusing. After that, we construct DFAs based on dependency nodes' string expression in topological order. When it comes to a literal-node, we will construct a DFA accepting only that literal value and when it comes to an arbitrary node, we will construct a DFA which accept arbitrary string. When it comes to operation-node, we will construct a post-image DFA which accept all possible string after the operation. When it comes to a sink node, we will construct a DFA M_S accepting whatever its predecessor DFA accept. Once the DFA M_S for sink is derived, we will construct a discriminant DFA M_D where $L(M_D) = L(M_S) \cap L(M_P)$. Since M_D will accept all the suspicious string that might flows to the sink, we will verify the App be determine if $L(M_D) = \emptyset$. If $L(M_D)$ is not an empty set, then the App stratify our predefined pattern, for example, the App might abuse the $ASIdentifierManager$ class.

Table 8: Assembly Code: Slicing of [SP,#0xE0+var_E0] (Param#3 of stringWithFormat)

Addr.	Instruction	Pseudo Code
85C8	MOVS R1, #0x53	R1 = 'S'
8620	STR R1, [SP,#0xE0+var_84]	M [SP+#0xE0+var_84] = R1
862C	LDR.W R9, [SP,#0xE0+var_84]	R9 = M [SP+#0xE0+var_84]
8630	STR.W R9, [SP,#0xE0+var_E0]	M [SP+#0xE0+var_E0] = R9

4 Flow analysis

In the previous section, we give a high level quick example to construct a string dependency graph of the sink value directly from assembly code level and do the property checking on the string dependency graph. The given example assume we have a very clear picture about every variables slice and the assembly code's execution flow. However, from the raw assembly code, we actually have no idea about those variable's (register and memory value) slice and execution flow before doing the binary flow analysis.

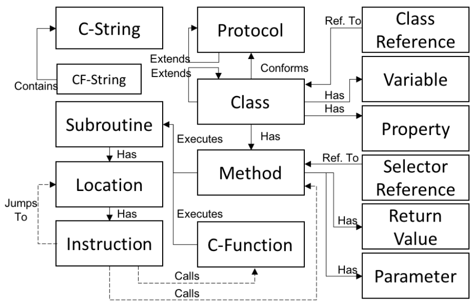


Figure 3: Entity Relation Diagram of iOS APP Assembly

In this section, we discuss about developing binary flow analysis modules with which we are able to rebuild program flows of mobile applications directly from their executables. The sample code in listing 2 is in Objective-C source code, which is not available in most cases. In reality, what we can download from Apple app store is encrypted binary executable and its meta data. With the help of decryption packages [29] and the binary analysis tool [47], we are able to get assembly segments as our inputs for flow analysis. We implement downloading, decryption, and generating assembly as an automatic process. The binary flow analysis module then consists of two main parts: 1) segment information extraction and 2) control flow graph construction. Figure 3 shows the flow structure and meta data we intend to generate from the assembly of each app. The entities and relations of solid lines can be extracted directly from assembly segments. The dash lines that indicate jumps of the blocks need to be resolved by traversing programs.

4.1 Segment Information Extraction

The first step of our process is to extract needed information such as entries of subroutines and constant strings from the assembly. There are various segments of iOS app assembly in ARMv7. Each segment carries its own information that is needed for our analysis. Below we list some common segments of an iOS APP's assembly in ARMv7.

- Header segment contains the meta-data of the assembly such as CPU type and the number of loaded commands.
- _text segment contains application instructions that are compiled from the Objective-

C or Swift source codes of mobile applications. This segment is composed of a bunch of subroutines. Subroutines are callable programs and each of them has a unique symbol called subroutine-label. A subroutine contains a sequence of instructions inside. Location-labels placed in subroutines are used to define a sub-section of the subroutine. By scanning the _text segment, we can collect all the instructions of subroutines and their locations.

- _stub segments may have various names such as _stubs, stub_helper, _symbolstub, _picsymbolstub. A stub in Mach-O binary is created for calling external functions. These functions are dynamically linked into the program. Similar to _text segments, _stub segments has a bunch of subroutines. Each subroutine has few instructions since it simply redirects execution to an external dynamic-linked function.

- _cstring segment is used to store the constant string values in assembly. Each constant string value is associated with a unique symbol called C-string-label that is used in the assembly instead of the string value.

- _lazy_symbol segment is used to store the mapping between pointers (lazy-symbol-label) and their references that are usually the imported external function.

- _objc segments (a segment whose name starts with "_objc") are used to store constant data of Objective-C entities such as Objective-C classes, methods, protocols, properties, variable (class fields) and so on. By scanning _objc segments, we can reconstruct the declaration information in apps source codes. For example, _objc_classlist segment is used to store a list of class labels. Each class label is a unique symbol that represents a declared class. _objc_methname segment is used to store the list of method names each associated with a method-name-label. The label is used as selector to invoke the corresponding method. _objc_classname segment is used to store the class names each associated with a class-name-label. The label is used as receiver to determine the class instance reference. _objc_methtype segment is used to store the signature of methods which disclose the parameter types and the return type of a method. _objc_const segment defines data structure of classes. _objc_ivar segment records a list of instance-variable-label. By scanning these _objc segments, one can reconstruct the classes and the protocols that these objects belong to.

- import segment is used to store the information for linked library and imported external functions.

We built an ARMv7 assembly syntax parser that is able to derive needed information as specified in Figure 2 from these segments. The remaining part is to resolve (indirect) jumps of blocks to make the control flow graph complete.

We've define our own ARMv7 assembly language grammar and use the parser generator ANTLR4 to do the lexical analysis and parsing to convert the assembly code into structural syntax tree in order to extract information from each of the segments. In the grammar, we've defined lexical analysis rules for the lexer to tokenize the assembly code. Tokens generally contains all the assembly's keywords, flag symbol, instruction name, register name and operators. For example, listing 4 shows the snippet of the lexical grammar for recognizing a register name token in the assembly. It defined that a valid register token is 'R0' to 'R15', 'A1' to 'A4' or 'V1' to 'V8'. After the assembly code has been tokenized into a token stream, we then input the token stream into the parser to get the syntax tree. The parser program is generated from our parser grammar; listing 5 is the snippet of our ARMv7 assembly parser grammar. The 1st line defined that an assembly is consist of some directives, instruction and EOL (end-of-line) tokens, and it may end with one EOF (end-of-file) token. The 2nd line defined that an directive can be categorized as data-definition-directives, control-directives, miscellaneous-directives and so on. Further more, in line 3, it defined that in the data-definition-directives category, it contains ARMv7 directive such as DCB, MAP and so on. In line 4, it defined that a DCB directive may start with an identifier (id) follow by a DCB token, and after that DCB token, one or more quoted-strings token (or number-expressions) will appear.

Listing 4: Snippet of ARMv7 lexical grammar

```
ARM_REG : 'R' ( '1'[0-5] | ([0-9])) | 'A' [1-4] | 'V' [1-8];
```

Listing 5: Snippet of parser grammar

```
asm      : (directive|instruction|EOL)+ EOF?;
directive: dataDefDirective | ctrlDirective | miscDirective|...;
dataDefDirective: dirDCB | dirMAP | ...;
dirDCB : id? DCB (QUOTED_STRING | numExpr)  (',' (QUOTED_STRING | numExpr))
        *;
numExpr: ...;
```

Table 9: Segments of iOS APP Assembly

Header
__text
__stub_helper
__symbolstub1
__picsymbolstub4
__cstring
__objc_methname
__objc_classname
__objc_methtype
__lazy_symbol
__nl_symbol
__objc_classlist
__objc_protolist
__const
__objc_selrefs
__objc_classrefs
__objc_superrefs
__objc_data
__CFString
__objc_ivar
__data
Import

4.1.1 __text Segment

__text segment contains application instructions which compiled from the Objective-C or Swift source code of APPs. This segment is composed of a bunch of subroutines.

Subroutines are callable programs just like functions in source code and each of them has a unique symbol called subroutine-label. A subroutine will contain a sequence of instructions inside. Some location-labels can be placed into a subroutine which used to define a sub-section in the subroutine. To increase the consistency in system implementation of BinFlow, BinFlow will automatically add a pseudo-location called "Entry" for every subroutine so that every instruction in it can belong to a location block just good. By scanning the *__text* segment, BinFlow can collect data of subroutines and their locations as well as every instruction in it.

4.1.2 Stub Segments

The name of stub segments may have many possibility such as *__stubs*, *__stub_helper*, *__symbolstub*1 , *__picsymbolstub*4. A stub in Mach-O binary is created for calling external functions which are dynamically linked into the program. The content inside stub segments is much the same as in *__text* segment and is composed of some subroutines as well. However, the amount of instructions in stub subroutines is little in general because these subroutines only handle the action of calling external dynamic-linked functions. Listing 6 shows an example of a stub which is used to call "malloc" function in the external library. The first line in the listing is used to define the subroutine-label of this stub subroutine and is called "_malloc" . The second line in the listing is the only instruction of this subroutine which will write the actual address of implementation subroutine in the external library to the program counter (PC) and trigger a jump to external library code. Whenever the APP intends to call external *malloc* function, it will call this internal stub first then the stub will take over the task and call to the implementation code in the library. Since the content type in stub segments are similar to those in *__text* segment, the way that BinFlow collect and store the data in stub segments is same as *__text* segment.

Listing 6: An example of stub-helper subroutine of malloc function

```
_malloc
LDR PC, =__imp__malloc
```

4.1.3 __cstring & __CFString Segment

__cstring (C-String) segment is used to store the constant string value in assembly. Every constant string value will get a unique symbol called C-string-label. Listing 7 shows an simplified *__cstring* area in assembly. Line 1 declares the start of *__cstring* segment. Line 2 declares a constant string value "*Hello World*" and its C-string-label is called "*aHelloWorld*". After scanning every entry in the *__cstring* segment, BinFlow will record the mapping relation between C-string-label and its constant string value.

__CFString (CFString) segment is used to store constant CFString variable in APP binary and every CFString in this segment will have a CFString-label as its symbol. According to Apple's documentation, CFString variable provides a suite of efficient string-manipulation and string-conversion functions, and most important of all, CFString is toll-free bridged with its Cocoa Foundation counterpart, NSString class. In other words, *__CFString* segment also store the constant NSString value in APPs. Since a CFString will encapsulate a C-String, the CFString-label will be mapped to a C-string-label and by further lookup the C-string-label in *__cstring* segment, BinFlow can figure out the representing string value of a constant CFString.

Listing 7: Example of __cstring segment

```
1 AREA __cstring, DATA, ALIGN=0
2 aHelloWorld        DCB "Hello World",0
3 aNsmutableparag    DCB "NSMutableParagraphStyle",0
4 aIconshare         DCB "IconShare",0
```

4.1.4 __lazy_symbol & __nl_symbol_ptr Segment

__lazy_symbol segment will store the mapping relationship between pointers (lazy-symbol-label) and imported external function which is lazily bound. *__nl_symbol*$_p$*tr* segment will store an array of pointers (nl-symbol-label) and its non-lazily bound data which will be bound at the time the APP binary is loaded. Line 2 in Listing 8 shows an example entry in *__lazy_symbol* segment; a lazy-symbol-label *_UIApplicationMain_ptr* will reference to an function *__imp__UIApplicationMain*. Line 2 in Listing 9 shows an example entry in *__nl_symbol_ptr* segment; nl-symbol-label *_objc_msgSend_ptr_*0 is bound with function *__imp__objc_msgSend*. BinFlow will scan these two segments and record the reference relationship between one label to another.

Listing 8: Example of __lazy_symbol segment

```
1 AREA __lazy_symbol , DATA
2 _UIApplicationMain_ptr DCD __imp__UIApplicationMain
3 _NSLog_ptr DCD __imp__NSLog
4 _NSStringFromClass_ptr DCD __imp__NSStringFromClass
```

Listing 9: Example of __nl_symbol_ptr segment

```
1 AREA __nl_symbol_ptr , DATA
2 _objc_msgSend_ptr_0 DCD __imp__objc_msgSend
3 _objc_msgSendSuper2_ptr DCD _objc_msgSendSuper2
```

4.1.5 __objc Segment

A segment whose name starts with "*__objc*" is used to store constant data of Objective-C entities such as Objective-C classes, methods, protocols, properties, variable (class fields) and so on. By scanning these segments, BinFlow can reconstruct the declaration information in APP's source codes.

__objc_classlist segment will store a list of class-labels, which are unique symbols representing each declared class. Listing 10 is an example assembly of *__objc_classlist*; line 2 represent the class-label *_OBJC_CLASS_$_ViewController* for a class called *ViewController*. Notice that BinFlow did not know exactly what the class name for class *_OBJC_CLASS_$_ViewContr* is at this time. It should further look into and cross reference data in other segments such as *__objc_classname* segment and *__objc_const* segment so that it can fetch other information behind the class label of the class.

Similar to *__objc_classlist* segment, *__objc_protolist* segment store a list of protocol-labels that represent the existence of some protocol in the APP. And further information for those protocols can be found in other segments such as *__objc_const* and *__objc_data* segment.

Listing 10: Example of __objc_classlist Segment

```
AREA __objc_classlist, DATA
DCD _OBJC_CLASS_$_ViewController
DCD _OBJC_CLASS_$_MySubClass
DCD _OBJC_CLASS_$_AppDelegate
DCD _OBJC_CLASS_$_MyClass
DCD _OBJC_CLASS_$_TestDelegate
```

__objc_methname segment stores a list of constant method name string value. As shown in Listing 11 the data format in *__objc_methname* segment is just like in *__cstring* segment, every method name will be mapped by a unique symbol call method-name-label. For instance, line 2 indicate that a method-name-label called *sel_viewDidLoad* mapped to a constant name "*viewDidLoad*". In Objective-C programming, these method name is also known as "selector" which used to select a set of methods regardless of their owner classes. When a method call is performed at runtime, the program uses two key elements to determine which subroutine should be invoked. One is the class instance reference (receiver) and the other is selector (method name). The method invocation mechanism in iOS executable will be further discussed in the later section.

Listing 11: Example of __objc_methname Segment

```
1 AREA __objc_methname, DATA, ALIGN=0
2  sel_viewDidLoad DCB "viewDidLoad",0
3  sel_literalConcat_appendFormat DCB "literalConcat_appendFormat",0
4  sel_bundleWithPath_ DCB "bundleWithPath:",0
```

__objc_classname segment shares the same data format as *__objc_methname*. A class-name-label will be mapped to a class name constant string.

__objc_methtype segment is used to store the signature of methods or properties. A signature for a method, in general, will disclose the method's return value type and the parameter types it takes; signature for a property will contain the type of that property.

Listing 12 shows an example code fragment in *__objc_methtype* and every method type will be mapped by a method-type-label. Line 2 is a method's signature and its method-type-label is $aV804$. The signature definition of $aV804$ is described by the constant string value "$v8@0:4$". "$v8$" means that the method's return type is a void type. "@0" means that the accepting type of the first parameter is a generic object (also know as *id* type) and ": 4" means the type of the second parameter is a selector type (a data structure that representing method name). Although the method signature discloses that this method accepts two parameters, the parameter declared in the Objective-C source code is actually zero– no parameter is required. This is because the compiler automatically adds two extra parameters to every method declared in the source code. One is used to hold the sender instance (owner instance of the called method) and the other is the method's selector. Line 3 in Listing 12 is the signature of an property. The method-type-label is *aUiwindow* and it represents that the property will return a generic object.

Listing 12: Example of __objc_methtype Segment

```
1 AREA __objc_methtype, DATA, ALIGN=0
2 aV804 DCB "v8@0:4",0
3 aUiwindow DCB "@",0x22,"UIWindow",0x22,0
```

__objc_ivar segment record a list of instance-variable-label which is some symbol for instance variable. As example shows in Listing 13, there are two instace-variable-labels. One is *_OBJC_IVAR_$_ViewController._webView* and the other one is *_OBJC_IVAR_$_AppDelegate._u* Similar to *__objc_classlist* and *__objc_protolist* segment, BinFlow only knows that there exist two instance variables. As for the further information of the two variable's information such as variable name and owner classes, it needs to take a further looks into *__objc_const* segment.

Listing 13: Example of __objc_ivar Segment

```
AREA __objc_ivar, DATA
_OBJC_IVAR_$_ViewController._webView DCD 4
_OBJC_IVAR_$_AppDelegate._window DCD 4
```

The segments we just discussed in the *__objc_∗* family are, in terms of a database system, normalized entities; hence we got very little information from each individual entity. To get the whole picture of a class or protocol, we need to reconstruct and maintain the relations of among *__objc_classlist*, *__objc_protolist*, *__objc_methname*, *__objc_methtype*, and *__objc_ivar* segment. And this task is done by scanning the *__objc_const* and *__objc_data* segment.

4.1.6 __objc_const & __data Segment

In *__objc_const* segment, there exist some different data structure in it. We will first discuss about collection data structure in this segment and they are: *__objc2_meth_list*, *__objc2_ivar_list*, *__objc2_prop_list*, *__objc2_prot_list* structure.

Listing 14 is an example content in *__objc_const* segment and line 14 define a method list.The method-list-label (identify symbol) of the method list data structure is called *OBJC_INSTANCE_METHODS_ViewController*. In the same line, the hexadecimal number "0x1E" means the following 30 entries of *__objc2_meth* data structure is belong to this method list. Let's take the first method data , which is a *__objc2_meth* structure, as an example. It disclosed that the method name of this method entry can be looked up from *__objc_methname* segment using method-name label *sel_viewDidLoad*. The method's signature can be looked up from *__objc_methtype* segment using method-type-label *aV804*. The subroutine which will be executed when this method is called can be looked up from *__text* segment by using subroutine-label *__ViewController_viewDidLoad_*.

Listing 14: Example of __objc2_meth_list structure

```
_OBJC_INSTANCE_METHODS_ViewController __objc2_meth_list <0xC, 0x1E>
__objc2_meth <sel_viewDidLoad, aV804, __ViewController_viewDidLoad_+1>
__objc2_meth <sel_didReceiveMemoryWarning, aV804,
        __ViewController_didReceiveMemoryWarning_+1>
__objc2_meth <sel_testMethod_____, aV2804i8i12i16i,
        __ViewController_testMethod______+1>
...
```

Listing 15 is an example data of a *__objc2_ivar_list* structure which used to represent a list of instance variables. Line 1 declare that there is an instance-variable-list-label called *_OBJC_INSTANCE_VARIABLES_AppDelegate* and in the same line, the decimal number "1" in the tuple that enclosed by "¡,¿" means that there is an entry of *__objc_ivar* data structure belongs to this list. Line 2 is the declaration of the __objc_ivar data. *_OBJC_IVAR_$_AppDelegate._window* is its instance-variable-label and its variable name can be looked up in *__objc_methname* segment using method-name-label "*a_window*". The variable's type can be looked up in the *__objc_methtype* segment by using method-type-label "*aUiwindow*".

Listing 15: Example of __objc2_ivar_list structure

```
1 _OBJC_INSTANCE_VARIABLES_AppDelegate __objc2_ivar_list <0x14, 1>
2 __objc2_ivar <_OBJC_IVAR_$_AppDelegate._window, a_window, aUiwindow, 2, 4>
```

Listing 16 is an example for *__objc2_prop_list* structure which representing a collection of properties. Line 1 indicated that the symbol (or property-list-label) of this property list is called "*UIApplicationDelegate_$properties*" and there are only one member in this collection. The only property member in this collection is declared in line 2 and the property's name can be looked up in *__cstring* segment or *__objc_methname* segment by using label "*aWindow*". As for the property type, it can be looked up in the *__cstring* segment or *__objc_methname* segment by using label "*aTUiwindowN*".

Listing 16: Example of __objc2_prop_list structure

```
1 UIApplicationDelegate_$properties __objc2_prop_list <8, 1>
2 __objc2_prop <aWindow, aTUiwindowN>
```

For *__objc2_prot_list* structure, it is a list of protocol-labels that represent a set of protocols. Line 1 Listing 17 declare such a protocol list whose protocol-list-label is called "*UIApplicationDelegate_$prots*" and it has only one member. The only member is a protocol-label in line 2 is called *_OBJC_PROTOCOL_$_NSObject*. To get more information for this protocol *_OBJC_PROTOCOL_$_NSObject* such as its methods, BinFlow will further look up data in *__objc_const* and *__data* segment by using this protocol-label.

Listing 17: Example of __objc2_prot_list structure

```
UIApplicationDelegate_$prots __objc2_prot_list <1>
DCD _OBJC_PROTOCOL_$_NSObject
```

After collecting those collection data from *__objc2_meth_list*, *__objc2_ivar_list*, *__objc2_prop_list*, and *__objc2_prot_list*, BinFlow can combine them and reconstruct the whole picture of class hierarchy, protocol hierarchy, class members, and protocol members by further looking into the *__objc2_class*, *__objc2_class_ro* and *__objc2_prot* data structure in *__objc_const* and *__data* segment:

For reconstructing information of a class (such as class hierarchy and class member), they are provided in the structure of *__objc2_class_ro* and *__objc2_class* as shown in Listing 18. There are two kinds of data for a single class, the metadata and instance data. Metadata of a class provide the information such as static method list, static variable list, and static property list. Instance data of a class is its instance information such as instance method list, instance variable list, instance property list and so on. Metadata of a class is described in a pair of *__objc2_class_ro* and *__objc2_class* data structure and so does instance data of that class. To sum up, in order to fully describe a class, it takes four entry of data as shown in Listing 18. Line 2 is an example of *__objc2_class_ro* data structure which store the metadata of a class; *ViewController_$metaData* is the identifier of this data entry and *aViewcontroller* is a C-string-label which can be used to find the actual name of the class. Line 2 is an example of *__objc2_class_ro* data structure which has identify label *_OBJC_METACLASS_$_ViewController* and store detail information for class metadata. *_OBJC_METACLASS_$_ViewController* is the class-label for this class and *_OBJC_CLASS_$_UIViewController* is the class-label for the class's superclass. It is because the super-class-label is provided in the class metadata, BinFlow can reconstruct the class hierarchy by looking up super class's metadata recursively by the super-class-label. In the end of Line 2, *ViewController_$metaData* is the identifier used to find the *__objc2_class_ro* data entry which should be paired with this metadata. For parsing the instance information for this class, they are store in line 3 and

4, *_OBJC_INSTANCE_METHODS_ViewController* is the method-list-label used to find out the methods which belong to this class. Since the class example in Listing 18 does not have any property, variable or even conforming any protocols, we don't see any property-list-label, ivar-list-label, or protocol-list-label, instead, they appear as "0" in the entry to denote the emptiness of these collections.

Listing 18: Example of __objc2_class_ro structure

```
ViewController_$metaData __objc2_class_ro <0x81, 0x14, 0x14, 0,
        aViewcontroller, 0, 0, 0, 0, 0>
_OBJC_METACLASS_$_ViewController __objc2_class <_OBJC_METACLASS_$_NSObject,
        _OBJC_METACLASS_$_UIViewController, __objc_empty_cache,0,
        ViewController_$metaData>
ViewController_$classData __objc2_class_ro <0x80, 0xA2, 0xA2, 0,
        aViewcontroller,_OBJC_INSTANCE_METHODS_ViewController, 0, 0, 0, 0>
_OBJC_CLASS_$_ViewController __objc2_class <
        _OBJC_METACLASS_$_ViewController,_OBJC_CLASS_$_UIViewController,
        __objc_empty_cache, 0,ViewController_$classData>
```

For reconstructing the information of a protocol (such as protocol hierarchy and protocol members), they are provided in the structure of *__objc2_prot*. As listing 19 shows, there are 8 element (wrapped between < and > symbol) inside the *__objc2_prot* structure, and the protocol-label *_OBJC_PROTOCOL_$_JSObjectDelegate* is the unique identifier used to reference to this data entry. The 2nd element *aJsobjectdelega* is a class-name-label which can be used to look up the protocol's real name in the *__objc_classname* segment. The 3rd element *JSObjectDelegate_$prots* is the protocol-list-label which can be used to took up the protocol's super protocols and BinFlow can reconstruct the protocol hierarchy from this information. As for the protocol's members, the 4th element *_OBJC_INSTANCE_METHODS_JSObjectDelegate* in the structure is the method-list-label used to lookup the instance methods of the protocol and the 5th is the method-list-label used to lookup the class methods of this protocol. Since the example in listing 19 does not have any class method, the 5th element is represent as "0". The 6th element *_OBJC_INSTANCE_METHODS_JSObjectDelegate_*0 is another method-list-label which can be used to lookup the optional instance methods of the protocol and the 7th element inside the structure is method-list-label which can be used to lookup the optional class methods of the protocol. The 8th element inside the structure is the property-list-label which can be used to lookup the property member of this protocol. Since the example protocol has no class method or property member, the 7th and 8th element are denoted as "0" due to the emptiness.

Listing 19: Example of __objc2_prot structure

```
_OBJC_PROTOCOL_$_JSObjectDelegate __objc2_prot <0, aJsobjectdelega ,
        JSObjectDelegate_$prots ,_OBJC_INSTANCE_METHODS_JSObjectDelegate , 0 ,
        _OBJC_INSTANCE_METHODS_JSObjectDelegate_0 , 0,0>
```

4.1.7 Import Segment

The last segment in iOS APP's assembly is "import" segment where used to store the information for linked library and imported external function. As example content shown in Listing 20, we see the application is going to link three external functions "free","malloc",and "printf" and their reference label in assembly are "__imp__free", "__imp__malloc", "__imp__printf".

Listing 20: Example of "import" segment

```
; Imports from /usr/lib/libSystem.B.dylib
IMPORT __imp__free
IMPORT __imp__malloc
IMPORT __imp__printf
...
```

4.2 Control flow graph construction

After preprocessing the segments, we have collected most information to build the control flow graph. In fact for each subroutine, we shall be able to build nodes accordingly. The rest work is to resolve jumps to connect executions among subroutines. We define nodes

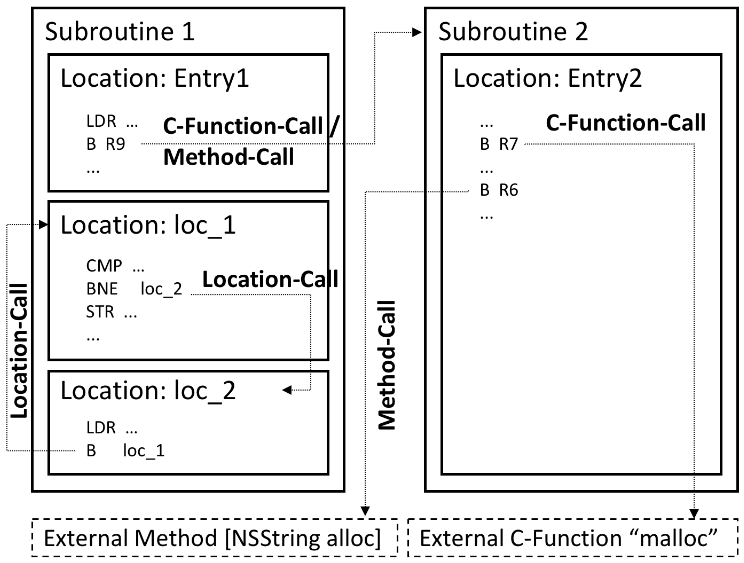


Figure 4: Relations among subroutines, locations, and instructions

of our control flow graphs as follows: 1) subroutine-node: a node denotes a starting point of a subroutine. A subroutine-node does not have predecessors, 2) location-node: a node denotes a starting point associated with a location-label in a subroutine, 3) instruction-node: a node denotes a single instruction, 4) call-subroutine-node: a node denotes the action to jump to the entry point of an internal subroutine, 5) call-location-node: a node denotes the action to jump to a location in an internal subroutine, 6) call-external-c-function-node: a node denotes the call to an linked external function (external API), where the function' subroutine instructions are not defined in assembly, 7) call-external-method-node: a node denotes the call to a method in the linked external library, and 8) call-unknown-node: a node denotes that we fail to resolve the jumping target. Figure 5 shows an example of the control flow graph that we intend to construct after processing the segments. There are three kinds of jumps in app assembly: C function calls, method calls and location calls. A C function call (or a method call) is triggered by a jumping between two subroutines. A location call occurs within a subroutine. In ARMv7, there are several ways to do a location jump, e.g., using the B instruction with the location-label. By looking at the location-label, it is straightforward to find what location is jumped to. However things become more complicated when calling a C function or method.

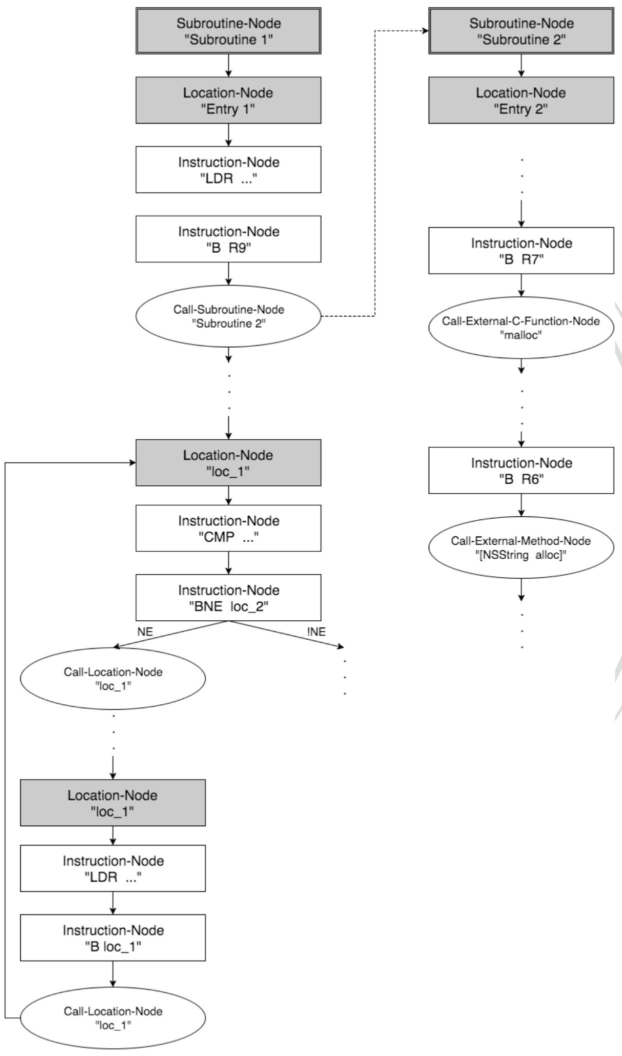

Subroutine-Node "Subroutine 1"
Location-Node "Entry 1"
Instruction-Node "LDR ..."
Instruction-Node "B R9"
Call-Subroutine-Node "Subroutine 2"
Location-Node "loc_1"
Instruction-Node "CMP ..."
Instruction-Node "BNE loc_2"
NE
!NE
Call-Location-Node "loc_1"
Location-Node "loc_1"
Instruction-Node "LDR ..."
Instruction-Node "B loc_1"
Call-Location-Node "loc_1"
Subroutine-Node "Subroutine 2"
Location-Node "Entry 2"
Instruction-Node "B R7"
Call-External-C-Function-Node "malloc"
Instruction-Node "B R6"
Call-External-Method-Node "[NSString alloc]"

Figure 5: Flow of subroutine, locations and instructions

Notice that node on CFG will also carry the dependency relation which shall be used in backward slicing process during constructing string dependency graph later in phase 3. For example, if the instruction of an instruction-node is $MOV R0, R1$, then this node will also carry a dependency relationship indicating that the value of R0 depends on the value of R1. In our implementation, we use a key-value pair to record this kind of value dependency, we call the left value R0 a dependency-key and call the right value R1 a dependency-value. In this way, when visiting node on the control flow graph, we can query the node with some dependency-key and see if that node carried the dependency relationship of that key or not. The number of key-value pairs carried in a single node depends on how many assignment operation will that instruction produce, so it is possible that a node might carry multiple dependency key-value pair. And also, register and memory address can be both dependency-key and dependency-value since they could appear as a left or right value in a statement. However, a constant value such as a label or immediate value can only be a dependency-value because they only act as right value.

Edges in our CFG are used to denote the execution flows and jumps. If the instruction shall trigger a location call, C function call or method call, the instruction-node will have an edge which direct to one of the following call-node: call-location-node, call-subroutine-node, call-external-C-funcation-node, call-external-method-node, or call-unknown-node.

A pair of branching edges in our CFG is used where there is a conditional execution. Figure 5 shows an example of a BNE instruction which conditionally trigger a jump to another location. The "NE" on the instruction name is one of the optional condition symbols for the B instruction. In ARM assembly, a lot of instructions include B instruction can carry an extra condition symbol indicating the conditional execution. Whether or not to execute of an instruction depends on the current status of the program, and to record the status of the current program, ARM uses an additional Program Status Register (PSR) and it will store some useful flags called Condition Code Flags to determine if the conditional instruction should be executed or not.

When the NE condition of BNE instruction on figure5 is hold, the program will conduct a location call and jump to location loc_2 in $Subourtine1$. If the jump fails to be triggered due to the condition does not hold, the next instruction follows by the BNE instruction will be executed thus form a conditional branch here in the example. As figure 5 shows, the branch is denoted by a branching edges on the BNE instruction node and the condition symbol is also carried on each side of the branching edge.

Figure 6 shows an example of assembly code that triggers a function-call with B instruction in line 5, where the function called in line 5 depends on the value of register $R6$. We need to resolve the value of $R6$ in order to get the target function. To deal with such case, we plan to adopt static backward slicing to resolve $R6$. Upon constructing the graph, we first scan the instruction in each subroutine in forward manner and record the dependency of values in each instruction when it is visited. For instance, when we first see the MOV instruction in line 1, we record the dependency of R2's value in an dependency entry telling that R2's value depends on label *_malloc* (denoted as $R2 : \_malloc$). In line 2, the STR instruction stores the value of $R2$ into the memory address $R9$, so we record the dependency that the memory value at address $R9$ depends on $R2$, (denoted as $R9] : R2$). In line 3, we record that $R4$ depends on $R9$ ($R4 : R9$) and for LDR instruction in line 4, we record that R6 depends on memory value at address $R4$ ($R6 : [R4]$). So when the forward scanning process finally comes to line 5, we realize that there is a jump that needs to resolve and the values depends on $R6$. We can then start the backward slicing searching to find out the value of $R6$. Looking back to the dependency relations, we have $R6 : [R4]$ in line 4, where we know the value of $R6$ depends on memory value at address $R4$.To refer the reference of memory addresses, we adopt un-interpreted function , saying that, $a = b \implies M[a] = M[b], \forall$ address a, b in random access memory M. The dependency $R4 : R9$ in line 3 hence can be further interpreted as $[R4] : [R9]$. This helps us to realize that memory value at address $R4$ depends on memory value at address $R9$. Looking back on the dependency in line 2, we get that memory value at address $R9$ depends on value of $R2$ ($[R9] : R2$), and finally, in line 1, we refer that R2's value depends on the label *_malloc*. By looking up the information that we collect in the segments information extraction, we find the label refers to an external C functions stub subroutine (from the $_stub$ segment). Thus, we are able to find that the jump is actually an external C function call whose subroutine-label is called *_malloc*. We then build a call-external-c-function-node and add an edge from it to the entry node of the subroutine *_malloc*.

Listing 21: Instructions

```
MOV    R2, _malloc
STR    R2,[R9]
MOV    R4,R9
LDR    R6,[R4]
B      R6
```

Listing 22: Dependencies

```
R2 : _malloc
[R9] : R2
R4 : R9
R6 : [R4]
————
```

Figure 6: An external c function call and the dependency relations

Things become more complicated when it is a method call. In Objective-C, a method call in runtime is actually triggered by *_objc_msgSend* function with the first parameter defines the instance of the class and the second parameter defines the method. This is called indirect jump since we need first resolve the target function (*_objc_msgSend*), and then resolve the parameter values $R0$ to refer to the class instance and $R1$ to refer to the method.

Figure 7 shows a sample assembly segment that triggers a method call to method *_alloc* of class *MySubClass*. The branching instruction in line 10 indicates that the target function can be resolved by using R9's slice in line 7 to 9, where it loads the value from memory using a pointer. We recognize that the iOS compiler typically adopts a sequence of 3 instructions: MOV, ADD and LDR instructions to do it. As shown in line 7, the MOV instruction loads the virtual address "$\_objc\_msgSend\_ptr\_0 - 0xAB0A$" into register $R9$. And then, in line 8, the ADD instruction adds the value of program counter (program counter is the value in register $R15$ in ARMv7, and PC is its alias) to get the physical memory address at runtime. Finally, the instruction in line 9 uses this physical address to find the target value in memory and load it into $R9$. This patterns allows us to overlook the computation such as ADD and to carry only dependency relations in the forward phase. In this case, we simply refer the dependence relation $R9 : R9$ in line 8 and $R9 : \_objc\_msgSend\_ptr\_0$ in line 7. In line 10, we realize that there is a jump and strats to resolve the value of $R9$. In line 9, we know that R9's value depends on memory value at address of R9 (R9 : [R9]). In line 8, there is a transparent dependency ($R9 : R9$). In line 7, we find that $R9 :_o bjc_msgSend_ptr_0$ but not $[R9]$. Appling the uninterpreted function relation, we have $[R9]$: [*_objc_msgSend_ptr_0*]; in other words, that memory value at address $R9$ depends on memory value at (virtual) address *_objc_msgSend_ptr_0*. By looking up label *_objc_msgSend_ptr_0*, we find the reference to a stub subroutine of an external C-function called *objc_msgSend*. Knowing that the jump is calling the function *objc_msgsend*, we need to resolve $R0$ (class instance) and $R1$ (method) to refer the correct subroutine for this method call. Similarly, we backward slicing searching the relations, we can resolve $R0$ as a class reference that points to a class called $MySubClass$, and resolve $R1$ as a reference to the method *alloc*. We then construct a call-external-method-node associated with the class and method.

Listing 23: Instructions

```
1                   MOV  R8,#(classRef_MySubClass −
                         0xAB42)
2                   ADD R8,PC
3                   LDR R0,[R8]
4                   MOV R6, #(selRef_alloc − 0xAB38)
5                   ADD  R6, PC
6                   LDR  R1,[R6]
7                   MOV  R9,#(_objc_msgSend_ptr_0 −
                         0xAB0A)
8                   ADD R9, PC
9                   LDR R9,[R9]
10                  BLX   R9
```

Listing 24: Dependency

```
1                   R8 :  classRef_MySubClass − 0
                          xAB42
2                   R8 : R8
3                   R0 : [R8]
4                   R6 : selRef_alloc − 0xAB38
5                   R6 : R6
6                   R1 : [R6]
7                   R9 : _objc_msgSend_ptr_0 − 0
                          xAB0A
8                   R9 : R9
9                   R9 : [R9]
10                  ————
```

Figure 7: A objective c method call and the dependency relations

5 String analysis on classes loaded from strings

In this section, we focus on string analysis for property checking. We are equipped with a bunch of analyzable information of apps after their CFGs have been generated. We are particularly interested in checking properties that are associated with dynamic loaded classes. The property checking module consists of two parts: (1) string dependency graph constructions for dynamically loaded classes and (2) string analysis and property checking.

5.1 String dependency graph constructions

A string dependency graph can be used to describe the composition of a string value that we interested in inside a program. For example, we are interesting about the parameter of *NSClassFromString* in line 15 of Listing25, and it is called *strC*, and from line 14, we know that it is the result of concatenating variable *strA* and *strB*. From line 4 to line 12, we can tell that there are two possible value for *strA*, "Geo" or "Mail". And from line 13, we know that *strB* can only be a constant string "Services".

On a string dependency graph, the variable value that we interested in is also called sink (in this case, *strC* is the sink), and Figure 8 shows the corresponding dependency graph for sink *strC*. It begin with a sink-node #1 which is used to represent the sink point and it is followed by a successor concat-node #2 which used to denote the concatenating operation. The successors of concat-node #2 are parameters of the concatenating operation and the parameters are represented by node #2 and #5. Since the first parameter of this concat-node #1 can have two possibility, node #2 is a union-node used to describe the uncertainty. As for node 5, it is a constant string in source code, so we used a literal-node to represent such constant string "Services". Similarly, the two parameter of union-node #2 are both constant string in source code, so node #3 and node #4 are also literal node.

We used source code in Listing 25 just to helping the explanation of the corresponding relation with its dependency graph on figure 8. However, BinFlow's input is assembly, not source code, and what BinFlow will get after all the analysis we just mention in previous subsections are control flow graph. Therefore, in this section, we will first clarify how BinFlow construct the string dependency graph from control flow graph by using an algorithm, and based on the algorithm, we will further enumerate and explain some example of different situations which might occur on the construction of string dependency graph.

Listing 25: Objective-C smaple code for string operation

```
1 -(void)usePrivateAPI
2 {
```

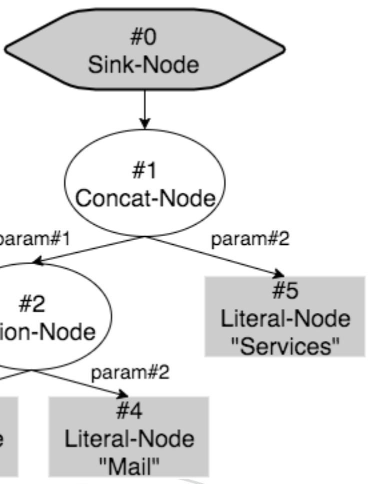


Figure 8: String dependency graph corresponding to listing 25

```
//...
NSString* strA = nil;
if(...)
{
strA = @"Geo";
}
else
{
strA = @"Mail";
}
NSString* strB = @"Services";
NSString* strC = [NSString stringWithFormat:@"%@%@",strA,strB];
Class someClass = NSClassFromString(strC);
//...
}
```

5.1.1 Dependency Graph construction algorithm

Before going through the algorithm, it is necessary for us to define some terms first:

Dependency-Key & Dependency-Value : dependency-key and dependency-value are carried in the instruction-node on a control flow graph. For example, on line 1 of Figure 6, the dependency-key for this instruction is register $R2$ and dependency-value is label *_malloc*. In other words, values that appears on left-side of the colon is dependency-key, and values that appears on the right-side is dependency value. Register and memory address can be both dependency-key and dependency-value since they could appears on the both side of colons. However, labels can only be dependency-value.

Dependency-Entry: a dependency-entry is used to identify every unique individual dependency-node . A dependency-entry is actually a tuple of (c, k, S). c represent a control-flow-node. k represent a dependency-key such as register or memory address. S is the call stack status of c's owner subroutine. Every dependency-entry is uniquely mapped to only one dependency-node, so that we can identify the existence of this dependency-node during the graph construction and to prevent re-create the same dependency node.

The algorithm for string dependency graph generation is actually a process of node production. In the very beginning, we only have a single sink-node in the uncompleted dependency graph. Then the algorithm will try to "extend" the sink-node by generating that sink-node's direct successor nodes. After we have those direct successor nodes generated, we continue to generate those nodes' direct successor nodes (the second generation of sink-node) in the same manner recursively. Once we verify that every node in the dependency graph has been extended then the algorithm is terminated.

Algorithm 1 is the algorithm which used to extend the successors of a dependency node d in a dependency graph G, also, the dependency-entry of node d is represented by a tuple of (c, k, S). To generate full dependency graph from its sink-node in the very beginning, we can invoke the recursion Algorithm 1 just like Listing 26.

As we just mention, to generate a completed dependency, we must start from a single sink-node first and recursively extends it and its successor nodes. Listing 26 is an example of making such initial sink-node of NSClassFromString sink and pass that sink-node as an input of Algorithm 1. In line 1, we initialize an empty dependency graph and then in line 2 and line 5 we make such sink-node d and insert it into G. The sink-node can be uniquely identified by its dependency-entry (c, k, S). Where c is a control flow node that calls to C-function $NSClassFromString$ and since what we interest in is the first parameter (the class name string value) of $NSClassFromString$, the dependency-key k should be register $R0$. As for initial call stack S, since we are in the initial level and no jump have ever occurs, the S is an empty stack now. Finally, in line 8, we invoke Algorithm 1 by passing the dependency graph G, initial sink-node d and its dependency-entry (c, k, S).

Algorithm 1 starts with a while-loop ranged from line 1 to line 16; this while-loop take the responsibility to do the backward search on the control flow graph where control flow node c exists in. If control flow node c carries some information about the dependency key k and can indicate that what dependency-value of this dependency-key k is mapped to, the while-loop will be breaked since $k \in depKey(c)$, otherwise, some update will be conducted inside the while loop. The main updates inside the while-loop is ranged from line 8 to 15. Since currently there is no dependency information carried on the control flow node c, we must keep searching backwardly from c's predecessors, and the predecessors of c is denoted as collection P. On line 11 , if c has only one predecessor, we simply update c to the only predecessor node $P[0]$. On line 13 , if c has more then one predecessors, then it means we are passing through a set of merging edges that point to c in the same time and implies that we must search all those paths and union those possible results. Therefore, we call another function in Algorithm 5 $HandleMultiPredNode$ to handle this situation. In Algorithm 5, we make a new union-node *n_union* which carries the dependency-entry (c, k, S) on dependency graph G and then we connect the dependency-node d to this *n_union*. And for each predecessor p in P, we make a temp-node $n_{unioned}\}$ which carried the dependency-entry of (p, k, S) and then we connect n_{union} to $n_{unioned}$; after that, to continue the extending process, we recursively call $ExtendDepNode$ to extend $n_{unioned}$ on line 8.

It is worth noting that on line 1 we call a method $connectToIfExist(G, (c, k, S))$ to prevent re-creating the dependency-node that has same dependency-entry. The logic of $connectToIfExist$ is shown on Algorithm 6. On line 1, it will return $false$ if currently there is no dependency-node which carry the dependency-entry of (c,k,S), otherwise, it will connect the node d to the node that can be identified by dependency-entry (c, k, S) on graph G. In Algorithm 1, you will continuously see this pattern before we intend to create any new dependency-node on dependency graph G and to simplify the explanation of Algorithm 1, we are no going to mention this pattern anymore.

On line 9, if c has no predecessor, it means that c is a subroutine-node, and current k is now storing the parameter that pass into this subroutine when someone is calling this subroutine. Since c is a subroutine-node, the way to extend d depends on the status of stack S. If S is an empty stack, then it means that there is no limitation on the candidate of callers of this subroutine. In other words, every call-subroutine-nodes that calls to subroutine node c (callee) could have the chance to act as the caller and take responsibility to generate the parameter value storing in k. Therefore, on line 2, we use X to denote the set of call-subroutine-nodes (the callers). If we can't found any callers that call to this subroutine, we will make an unknown-node on line 4 to represent that we can't solve dependency-entry (c, k, s) any further, otherwise, on line 7, we will first make a new union-node called n_{union} which has dependency-entry (c, k, s) on dependency graph G and then connect d to u_{union}. And for each call-subroutine-node x in caller set X, we'll make a temp-node $n_{unioned}$ which has dependency-entry $(predcessors(x)[0], k, S)$ and then connect n_{union} to $n_{unioned}$. To continue extending temp-node $n_{unioned}$, we recursively apply $ExtendDepNode$ algorithm to $n_unioned.$ Notice that the first item (control flow node) in the dependency-entry tuple of temp-node $n_{unioned}$ on line 11 is $predecessor(x)[0]$ (the first predecessor of x) instead of x. The reason is that x is a call-subroutine-node and $predecessor(x)[0]$ is an instruction-node which will trigger the call, and the dependency-key "after" calling the subroutine (such as return value) is carried in node x while the dependency-key "before" calling the subroutine (such as parameters) is carry in the node $predecessor(x)[0]$ or predecessors of it; since our target is finding the dependency-key k "before" calling the subroutine, the first item (control flow node) in dependency-entry tuple of $n_{unioned}$ will be $predecessor(x)[0]$.

From line 8 to 15, they are the main backward searching process, however, in the while-loop and before entering such range, we need to do some modeling for some special cases listed in line 2 to 7.

The first cases is modeling memory relocation function such as $_objc_storeStrong$ from line 2 to line 2. According to clang's official documentation, this function takes two parameters and the full signature is shown in Listing 27. Parameter $object$ is a valid pointer and value is null or a pointer point to a valid object. What will be perform inside this function is that *object* pointer will be updated and point to *value*. Therefore, in line 2, when c is a call-external-C-function-node and it calls to function $objc_storeStrong$, we then find out memory address store in R0 (the *object* parameter) and R1 (the *value* parameter). In other words, when we pass through this $objc_storeStrong$, we will got an additional dependency relation that says "$R0 : R1$" ($R0$ depends on $R1$) due to the pointer update. Therefore, on line 2 to 4, we first find out what address is stored in $R0$ (what does first parameter *object* pointer point to) by calling $BackwardFindMemAddr$ algorithm. The $BackwardFindMemAddr$ algorithm simply doing a backward control flow node searching to see if $R0$ depends on a static memory address and return that address as $addr_{R0}$. we determine if current k is equals to address in $R0$ ($addr_{R0}$), if it is, then it means that value originally stored in $addr_{R0}$ is now update to value stored in R_1 (the second parameter of) therefore we will make a temp-node n_{tmp} which carry a dependency-entry (c, R_1, S). After that, connect d to n_{tmp} and we extends node n_{tmp} by recursively calling $ExtendDepNode$.

The second case that need to be modeled is return value of function call from line 5 to line 7. Since return value will be stored in $R0$ after function call if the function has a return value. Therefore, we have to do such additional handling when k is $R0$ (when the value that we interesting now is depends on $R0$). Beginning from line 5, if k is now $R0$ and d is a Call-Subroutine-Node calls to subroutine n_{sbrt}. Then the $R0$ should depends on the return value after calling subroutine n_{sbrt} if it has return value. Since the callee subroutine will stored the returned value in $R0$, we will have to do the backward search from each of n_{sbrt}'s terminal control flow nodes (a subroutine could have multiple terminal control flow nodes if theres's any branches). Since different control flow in that callee subroutine will stored different return value into $R0$, we will first connect d to a union node n_{union} in line 6 to handle the non-determinism of return value. After that, for each callee's control flow terminal node c_{term}, we will make a temp-node n_{tmp} which carry a dependency-entry (c_{term}, R_0, S'). Notice that $S\prime$ is the caller stack after pushing the caller node c into original caller stack S. The reason why we push c into the stack is because we want to memorize the caller. We will need the information of such caller node when handling the case where there is no more step in the backward searching process in Algorithm 4.

The final case that need to be modeled before conducting the main backward searching process is handling the return value of calling external C-Function or external method begin from line 15 of Algorithm 1. If we can not determine the return value type of such external C-Function or external method, we will connection d to a Unknown-Node which denote that the value can not be resolved. Otherwise, we will model the return value of those functions which we already know the returned value type such as common string manipulations or arithmetic operations.

Once finishing handling those special cases, we will do the main backward searching process from line 8 to line 15. In the backward searching process, we will do different processing for different size of the predecessors P of current control flow node c. If current c has no predecessor ($|P| = 0$), it means the iteration have reached the root control flow node (a Subroutine-Node) and still can't find a control flow node c where $k \in depKey(c)$ (k is the current dependency-key which the searching loop is searching for). We will handle this situation in Algorithm 4. If current c has only one predecessor, then we simply update c to that predecessor control flow node and continue the backward searching iteration. If current c has multiple predecessors ($|P| > 1$), it means current control flow node c is actually a merge of some branches. In this case we will have to search in different branches and we handle this situation in Algorithm 5.

Line 17 to 34 is the block after breaking the backward searching while-loop. The execution will enter this block if and only if we found a control flow node c where $k \in depKey(c)$. In short, we have found a control flow node c which can answer what dependency value v will the dependency-key k mapped to. There are two situation for that dependency-value v. One is that the v is actually a literal (also known as a label in assembly) and is an atomic part in program which won't depends on anything else. The other is that the v is actually a dependency-key which can further mapped to another dependency-value. For the first one, we will make a literal node $n_{literal}$ and connect d to $n_{literal}$. For the second case where v is actually a dependency-key (line 23 to 34) , we will use the same updating logic same as line 8 to 15 in the while-loop.

Listing 26: Exmaple of generating dependency graph from sink node

```
G = Empty dependency graph;
c = Call-External-C-Function-Node which calls to NSClassFromString
    function;
k = R0;
S = [];
d = makeSinkNode(G);

//Generate dependency graph for first parameter of NSClassFromString
    function:
ExtendDepNode(G,d,(c,k,s));
```

Algorithm 1 ExtendDepNode($G, d, (c, k, S)$)

```
Input: $G$: The dependency graph which contain node $d$; $d$: The dependency node that need to extends itself and connect to its successors; $(c, k, S)$: The dependency entry of $d$
1: while do$\neg(k \in depKey(c))$;
2:     if $c \in$ Call-External-C-Function-Node $\wedge c$ calls objc_storeStrong then
3:         $HandleMemoryRelocation(G, d, (c, k, S))$;
4:     end if
5:     if $k = R_0$ then
6:         $HandleReturnValue(G, d, (c, k, S))$;
7:     end if
8:     $P = predecessors(d)$;
9:     if $|P| = 0$ then
10:        $HandleNoPredNode(G, d, (c, k, S))$;
11:    else if $|P| = 1$ then
12:        $c = P[0]$;
13:    else
14:        $HandleMultiPredNode(G, d, (c, k, S))$;
15:    end if
16: end while
17: $v = depVal(c, k)$;
18: if $v \in Label$ then
19:    if $d.connectToIfExist(G, (c, k, S))$ then
20:        $n_{literal} = makeLiteralNode(G, (c, k, S), literalOf(v))$;
21:        $d.connectTo(n_{literal})$;
22:    end if
23: else
24:    $P = predecessors(d)$;
25:    if $|P| = 0$ then
26:        $HandleNoPredNode(G, d, (c, v, S))$;
27:    else if $|P| = 1$ then
28:        if $\neg connectToIfExist(G, (c, v, S))$ then
29:            $n_{tmp} = makeTempNode(G, (P[0], v, S))$ ;
30:        end if
31:    else
32:        $HandleMultiPredNode(G, d, (c, v, S))$;
33:    end if
34: end if
```

Algorithm 2 HandleMemoryRelocation$(G, d, (c, k, S))$

```
1: $addr_{R0} = backwardFindMemAddr(c, R_0)$;
2: if $addr_{R0} = k \wedge \neg d.connectToIfExist(G, (c, R_1, S))$ then
3:     $n_{tmp} = G.makeTempNode(G, (c, R_1, S))$;
4:     $d.connectTo(n_{tmp})$;
5:     $ExtendDepNode(G, n_{tmp}, (c, R_1, S))$;
6: end if
```

Algorithm 3 HandleReturnValue$(G, d, (c, k, S))$

```
1: if $d \in$ Call-Subroutine-Node then
2:     $n_{sbrt} = calledSubroutineNode(d)$;
3:     $S\prime = S.push(c)$;
4:     if $\neg d.connectToIfExist(G, (c, k, S\prime))$ then
5:         $n_{union} = makeUnionNode(G, (c, k, S\prime))$;
6:         $d.connectTo(n_{union})$;
7:         for $c_{term} \in terminateNodes(s)$ do
8:             if $\neg d.connectToIfExist(G, (c_{term}, R_0, S\prime))$ then
9:                 $n_{tmp} = makeTempNode(G, (c_{term}, R_0, S\prime))$;
10:                $u_{union}.connectTo(n_{tmp})$ ;
11:                $ExtendDepNode(G, n_{tmp}, (c, R_0, S\prime))$;
12:            end if
13:        end for
14:    end if
15: else if $d \in$ Call-External-C-Function-Node $\vee$ $d \in$ Call-External-Method-Node then
16:    if $d$ calls to modeled function then
17:        make operation node;
18:    else if $d$ calls to method which has unknown return value type then
19:        $n_{unknown} = makeUnknownNode(G)$ ;
20:        $d.connectTo(n_{unknown})$;
21:    end if
22: end if
```

Listing 27: Signature of objc_storeStrong Function

```
id objc_storeStrong(id *object, id value);
```

The example dependency graph in Figure 2 on only contains a very simple concatenation-node. The manipulation on string variables can be varied, e.g., using a replace operation to substitute strings. String replacement in Objective-C can be done by calling instance method $stringByReplacingOccurrencesOfString$: $withString$: in $NSString$ class. The replacement operation has three parameters as its inputs: 1) the original string, 2) the match pattern, and 3) the replacement, and replaces the occurrences of the matched pattern to the replacement. In our dependency graph, we will build input nodes for the replacement-node when we observed the replace operation call. On the other hand, the dependency graph can be a directed cyclic graph. Consider the code segment in Listing 28. The dependency graph that has cyclic dependency relation where the variable s initially is an empty string, and appends character ”A” in a for-loop up to N. The cyclic dependency graph in Figure 9 discloses that the sink of Listing 28 is actually synthesized by concatenating various characters.

A dependency graph is a directive graph. A node in the dependency graph denotes

Algorithm 4 HandleNoPredNode$(G, d, (c, k, S))$

```
if $|S| = 0$ then
    $X = callerNodes(c)$;
    if $|X| = 0$ then
        $n_{unknown} = makeUnknownNode(G)$ ;
        $d.connectTo(n_{unknown})$;
    else
        $n_{union} = makeUnionNode(G, (c, k, S))$;
        $d.connectTo(n_{union})$;
        for $x \in X$ do
            if $\neg n_{union}.connectToIfExist(G, (predcessors(x)[0], k, S))$ then
                $n_{unioned} = makeTempNode(G, (predcessors(x)[0], k, S))$;
                $n_{union}.connectTo(n_{unioned})$;
                $ExtendDepNode(G, n_{unioned}, (predcessors(x)[0], k, S))$;
            end if
        end for
    end if
else
    $c_{caller} = S.pop()$ ;
    if $\neg d.connectToIfExist(predcessors(c_{caller})[0], k, S)$ then
        $n_{tmp} = makeTempNode(G, (predcessors(c_{caller})[0], k, S))$;
        $d.connectTo(n_{tmp})$;
    end if
end if
```

Algorithm 5 HandleMultiPredNodes$(G, d, (c, k, S))$

```
if $\neg d.connectToIfExist(G, (c, k, S))$ then
    $n_{union} = makeUnionNode(G, (c, k, S))$ ;
    $d.connectTo(n_{union})$;
    for $p \in P$ do
        if $\neg n_{union}.connectToIfExist(G, (p, k, S))$ then
            $n_{unioned} = makeTempNode(G, (p, k, S))$;
            $n_{union}.connectTo(n_{unioned})$ ;
            $ExtendDepNode(n_{unioned}, (p, k, S))$;
        end if
    end for
end if
```

a string expression and an edge in the dependency graph shows the dependency relation. Once we have the depedency graph, we can apply string analysis to resolve the values of the sink node given the values of the input nodes.

Besides the string dependency graph for dynamically loaded class names, we also construct string dependency graph for path of the dynamically loaded frameworks' path with the same technique. The listing 1 shows a sample code to dynamically load the framework bundle of the loaded classes. It is needed when the framework (contain the classes) does not statically included in the compile time. To load the framework dynamically, one should call a static method $+[NSBundlebundleWithPath:]$ and pass the path string as the first parameter. This is typically done before a program is trying to use iOS private

Algorithm 6 ConnectToIfExist$(G, d, (c, k, S))$

```
if $\neg exist(G, (c, k, S))$ then return $false$;
end if
$d' = nodeOf(G, (c, k, S))$;
$d.connectTo(d')$; return $true$;
```

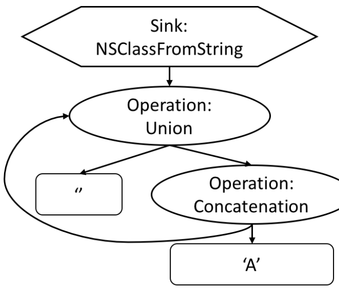


Figure 9: A Cyclic Dependency Graph

classes because a private framework can be easily identified if it is statically loaded in compile time. To construct the string dependency graph of the framework bundle path, we treat the first parameter (stored in $R2$) as a sink when *objc_msgSend* appears to invoke $+[NSBundlebundleWithPath:]$ method and adopt the same steps to infer the sink node's all successors. By construct the string dependency graph for bundle path, we now know how a bundle path string is synthesized.

Listing 28: Synthesize string with loop

```
NSString* s = @"";
for(int i = 0; i < N; i++){
s = [NSString stringWithFormat:@"%@%@",s,@"A"];
}
Class c = NSClassFromString(s);
```

5.2 String analysis and property checking

Once we have the dependency graph constructed, we can do the forward string analysis directly on the dependency graph to verify if the app satisfies a desired security policy, e.g., calling a sensitive API using *NSClassFromString* function or loading a private framework using $+[NSBundlebundleWithPath:]$. We first define undesired patterns (in regular expression) for each kind of sensitive functions. For example, for the sink of IDFA abusing, we define the undesired pattern as regular expression of "*ASIdentiferManager*". We use automata-based symbolic string analysis [41, 48] techniques where the values that string expressions can take during program execution using deterministic finite automata

(DFA).

Formally speaking, let us define the set of *Operation P* using abstract grammar:

1. $R ::= \emptyset | a | RR | R + R | \bar{R} | R^*$

2. $S ::= s1 | s2 | s3 | ...$

3. $P ::= \boldsymbol{unknown} | R | S + S | SS | S[S \rightarrow S]$

where $s1, s2, ...$ denote string expression and $a \in \Sigma$ is an literal symbol. Observe that R is the class of the regular expression. A string operation is the union $(S + S)$, concatenation (SS) or replacement $(S[S \rightarrow]S)$ of string expression. A *dependencygraph* is a directed graph $G = \langle V, E, \boldsymbol{cmd} \rangle$ with a vertex labeling function $\boldsymbol{cmd} : V \rightarrow P$. An edge $(v, v\prime) \in E$ means that the operation associated with $v\prime$ depends on the operation associated with v. Each vertex of the dependency graph represents a string expression (that is constructed by string operation that may use other string expressions, i.e., other vertices in the dependency graph). An $\boldsymbol{unknown}$ operation obtains a string from an external source such as return value of calling external APIs which might return arbitrary strings or fetching string from user input via GUI. A vertex associated with a regular expression specifies a set of string constants. Subsequently, a vertex labeled by an $\boldsymbol{unknown}$ operation or a regular expression has no successors (since they do not depend on any other string expression). In addition to $\boldsymbol{unknown}$ and regular expressions, union, concatenation, and replacement operations can be specified in a vertex of the dependency graph as an operation-node. The final vertex in a dependency graph (i.e., the vertex with no successors) denotes a sink, i.e., a parameter string expression on sensitive function that can be target of an abusing or attack, such as IDFA abusing.

Let L_U be a regular language of undesired strings (specified by the undesired pattern) of sink. Let $G = \langle V, E, \boldsymbol{cmd} \rangle$ be a dependency graph. During forward analysis we construct a deterministic finite automaton for each vertex $v \in V$ based on the operation $\boldsymbol{cmd(v)}$ associated with v and the DFAs that annotate the successors of v. Hence, each step of the forward analysis corresponds to a post-image computation for a string operation. When $\boldsymbol{cmd(v)} = \boldsymbol{unknown}$, we would like to represent an arbitrary string. Hence we construct a DFA accepting an arbitrary string in Σ^*. Similarly, when $\boldsymbol{cmd(v)} = R$ for some regular expression R, we construct a DFA accepting the regular language specified by the expression R. When $\boldsymbol{cmd(v)} = S_l + S_r$ where $s_l, s_r \in S$ denote two successors of the vertex v, we construct a DFA accepting the union of strings from the two successors. Let M_l and M_r be the DFAs of the successor s_l and s_r respectively. Then, the DFA M of v accepts the union of the languages of M_l and M_r. That is, $L(M) = L(M_l) \cup L(M_r)$. When $\boldsymbol{cmd(v)} = S_lS_r$, we construct a DFA accepting strings from the successor s_l followed by those from the successor s_r. The DFA M of v subsequently accepts the concatenation of the languages of M_l and M_r. I.e., M has the property that $L(M) = uw : u \in L(M_l), w \in L(M_r)$. Finally, when $\boldsymbol{cmd(v)} = s_o[s_f \rightarrow s_t]$, we construct a DFA accepting any pattern from s_o whose substrings from s_f are replaced by strings from s_t. Let M_o,M_f,M_t be deterministic finite automata of the successor s_o , s_f , and s_t respectively. The DFA M of v accepts the following language $\{w : k > 0, w_1x_1w_2x_2w_kx_kw_{k+1} \in L(M_o),\ w = w_1y_1w_2y_2w_ky_kw_{k+1}, x_i \in L(M_f), y_i \in L(M_t) forall1 \leq i \leq k, andw_j \notin ux\prime v : x\prime \in L(M_f), u, v \in \Sigma^* forall1 \leq j \leq k+1\}$.

After the DFA M_s for the sink $s \in V$ is obtained, M_s accepts (an over approximation) of all string values that can reach the sink assuming all $\boldsymbol{unknown}$ vertices (if any) can take arbitrary string values. $L(M_s) \cap L_U$ is the language of all malicious string that can reach the sink node. Let S_s accept this language, if $L(S_s) \neq \emptyset$, then we consider the sink is malicious since some undesired string might flow to the sink.

6 Evaluation

6.1 Implementation

We have implemented an end-to-end tool called Binflow for analyzing iOS mobile applications. The tool first downloads apps from the AppStore into a jail-broken iOS device such as iPhone. This process is is achieved by using a browser automation library called Selenium and a GUI automation library called Sikuli to have access control on iTunes, which is a native OS X application. We systematically complete the user authentication and the download process, send the IPA files to the iOS device via SSH File Transfer Protocol (SFTP), and install them via "ipainstaller." We then fetch the decrypted app binary by "Clutch2" and use SFTP again to send decrypted ones back to the server to execute the IDA Pro, which is a disassembler created by Hex-Ray. With IDA Pro, we finally generate the plain text of the ARMv7 assembly as the input for segment information extraction and control flow constructions. We build a bunch of json files as the information representation and build string dependency graphs for sinks that are identified during the control flow graph constructions.

6.2 Apps and Their String Dependency Graphs

We have run our tool to collect top apps in apple app store systematically. Many of our attempts failed due to the failures of installation and decryption. The disassembled apps include popular iOS Apps in each category such as Google Drive and Dropbox (Productivity), Facebook and Skype (Social Networking), Spotify Music (Music), Google Translate (Reference), Youtube (Photo&Music), Angry Bird Fight! RPG Puzzle (Games) and Amazon (Shopping).

Among them, we have successfully analyzed 1304 public apps (end-to-end) that contain hundreds of thousands of sub routines in total. For each app, we extract the corresponding segment information, build flow graphs for subroutines and connect them as the whole control flow graph with indirect jump analysis. For property checking, we identify two kinds of sinks $NSClassFromString$ and $BundleWithPath$; the first can be used to load classes dynamically while the later is used to load the framework from a bundle path dynamically.

We found 31641 sinks from 454 apps that have $NSClassFromString$ to load a class dynamically and 6343 sinks from 262 Apps to load the framework from a bundle path. Figure 10 summarizes the dependence graphs we have generated. Of those

37984 (31641+6343) sinks, there are 28% of the graphs have their sink assigned directly by a single literal. (In this case a constant scan or propagation can be used to resolve the value of the sink). There are 72% of the graphs contain string manipulations including union, concatenation and replacement.

6.3 Unknown Nodes in String Dependency Graph

Besides, we also found that 5377 string dependency graph (from 459) contains unknown-nodes. An unknown-nodes is created typically because of three kinds of situation:1) Arbitrary memory value: the string expression depends on a value in the memory under specific address but we fail to further trace down to statically resolve the value. 2) Arbitrary register value: the string expression depends on a register value that could not be further traced down anymore. 3) Arbitrary return value from external function/method calls: the string expression depends on return value of external function/method calls. For situation 3, it can be further divided into two cases: 3.1) We know the exactly what that external function/method is. 3.2) We have no idea about what is the name of that external function/method because we failed to resolve its name on CFG construction stage.

The reason why situation 1 happens is because the memory value is actually stored in runtime such as GUI input, HTTP response payload and so on. We have found 112 of our dependency graphs (93 apps) contains such kinds of unknown node.

As for the reason of situation 2, the register value can be a parameter value of some subroutine which not called by any other subroutine on the control flow graph. Generally, those subroutines are delegated methods of some protocols which will be invoked dynamically by external library on runtime, we cannot determine the value of its parameter by using static analysis so the value will remain unknown. We have found 273 of our dependency graphs (150 apps) contains unknown node of situation 2.

For situation 3.1, we have found 2424 dependency graph out of 348 apps contains such kind of unknown-node, and for situation 3.2, weve found 896 out of 310 apps contains such kind of unknown-node.

Figure 11 shows the occurrence times of the 4 different types of unknown-node in our experiments. The arbitrary unmodeled function/method's return value is the most common reason why we get unknown-node in string dependency graphs. The unknown-node on the string dependency graph will cause our analysis to take it as an arbitrary string and make an over-approximation.

The value uncertainty of those unknown-nodes can be categorized into 2 types as figure 12 shown: external source uncertainty and internally computed value uncertainty.

The external source, for example, might be an HTTP response message from the Internet or user input from GUI; in this case, assuming the value as an the arbitrary string seems reasonable because those sources can really return any value and can cause a so-called backdoor security concern. All of the arbitrary memory value, arbitrary register value and part of the arbitrary return value of known/unknown function/method can cause external source uncertainty. Since we have successfully submit an app which contains such back-door behavior to AppStore, and proven that Apple's black-box censoring is not enable to guarantee that all the 3rd party apps on AppStore can not access private APIs, assuming all the value from external world as arbitrary strings is necessary.

The internally computed value is the value that is computed by the program itself and can be predicted or modeled if we know the computation's input values in advance. As figure 12 showes, some of the return value from function/method can return internally computed value. For example, the substringWithRange method in NSString class will return a string's substring by a given range. If we know the original string and the range in advance, then we can know the result substring. The method return a string value but we haven't represented the returned string using reasonable automaton yet. Currently, we simply assume the result can be any string because we haven't modeled such operation yet. In the future, we will do a complete survey of methods of Objective-C NSString class and lower level C/C++ string function and try to represent the return value in reasonable automaton to reduce the false positive rate.

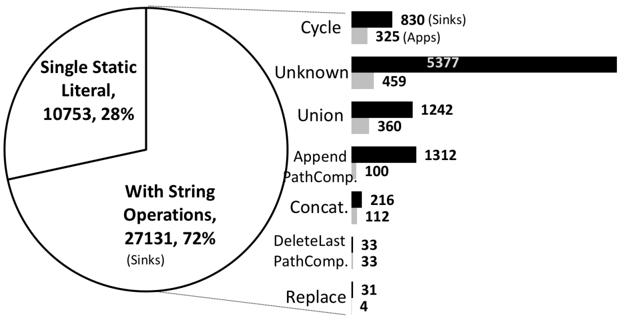


Figure 10: Types of string dependency graphs

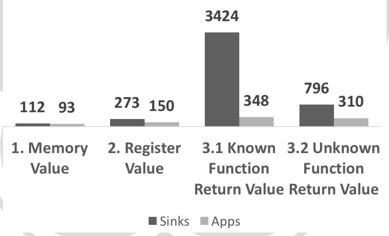
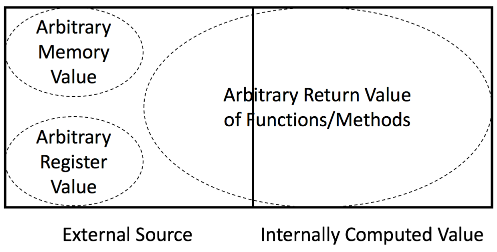


Figure 11: Types of unknown nodes



Arbitrary Memory Value
Arbitrary Register Value
Arbitrary Return Value of Functions/Methods
External Source
Internally Computed Value

Figure 12: Types of uncertainty

6.4 Property Checking

For sinks of *NSClassFromString*, we check: 1) private class usage, and 2) IDFA usage; while for sinks of *BundleWithPath*, we check the private framework usage.

To verify private class usage (Property 1), we construct the undesired pattern as the automaton that accepts all iOS private class names. These private classes are collected by analyzing the iOS SDK framework. In the 31641 dependency graphs/454 apps, we have found 1339 instances/372 apps that have their sink values intersecting with the values of the undesired pattern, i.e., they could have dynamically load private classes using *NSClassFromString*. However, taking a close look, we found that all of these 372 dependency graphs have at least one unknown node (the value may be from an external source) that is over approximated in our analysis as arbitrary strings, and hence these instances could be false alarms and need to be further investigated.

To verify the IDFA usage (Property 2), we use a particular string */ASIdentifierManager/* as its undesired pattern. In the 31641 dependency graphs/454 apps, we found 597 string dependency graphs/208 apps that contain no unknown nodes but have the sink value intersecting the undesired pattern (witnesses of loading */ASIdentifierManager/* dynamically). Specifically, there are 18 apps have their class loaded from string operations, e.g., apps *Valletti*, *InstituteforEmergingIssues*, *DailyAstronautsPuzzlesCollection−Jigsaw4Kids&BoysFun*, and *Cejfe...* have sinks as shown in Figure 13. Since Apple ruled the IDFA usage strictly during the AppReview, IDFA abusing has become one of the top 10 reasons for app to be rejected by AppReview. The strange way to load *ASIdentifierManager* raises two issues: 1) If the app is using IDFA in a regular way, why do not they load *ASIdentifierManager* statically in compile time? 2) If they do need to load the *ASIdentifierManager* class in runtime, why do they have synthesized the class name with unnatural strings? After we execute one of the app called "Valletti" on iPhone, we found no ads and confirmed its IDFA abuse.

To verify private framework usage, we use the regular expression */.∗PrivateFrameworks.∗/* as its undesired pattern. In the 6505 sinks of *BundleWithPath*, we found there are

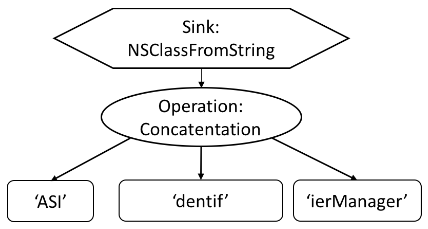


Figure 13: A finding sample that loads class name with substrings

1704 instance out of 241 Apps could have dynamically loaded the private framework bundle. However, these graphs also contain unknown nodes and may be false alarms in our analysis.

7 Conclusion

Most malicious behaviors and violations of security policies are related to the framework and class usages. We present the work that integrates string analysis with flow analysis to resolve dynamic loaded classes and frameworks of iOS mobile applications. We proposed a context-aware flow graph construction that resolves registers for indirect jumps and proposed the parameter-aware dependency graph construction for string analysis. We implemented an end-to-end analysis tool and discovered potential violations of public online iOS applications.

References

[1] "Number of apps available in leading app stores as of july 2015," http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores, (Visited on 01/04/2016).

[2] "G data mobile malware report threat report: Q3/2015," https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q3_2015_EN.pdf, (Visited on 01/04/2016).

[3] "Mcafee labs threats report november 2015," http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-nov-2015.pdf, (Visited on 01/04/2016).

[4] "Path," https://itunes.apple.com/us/app/path/id403639508?mt=8, (Visited on 01/04/2016).

[5] "Path app under fire for unauthorized address book upload," http://appleinsider.com/articles/12/02/07/path_app_under_fire_for_unauthorized_address_book_upload.html, (Visited on 01/04/2016).

[6] "Mobilead2013," http://www.emarketer.com/Article/Driven-by-Facebook-Google-Mobile-Ad-Market-Soars-10537-2013/1010690, (Visited on 01/04/2016).

[7] "Gartner says mobile advertising spending will reach $18 billion in 2014," http://www.gartner.com/newsroom/id/2653121, (Visited on 01/04/2016).

[8] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond, "Truth in advertising: The hidden cost of mobile ads for software developers," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 100–110.

[9] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private api abuse in ios applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 44–56.

[10] F. Yu, Y.-C. Lee, S. Tai, and W.-S. Tang, "Appbeach: Characterizing app behaviors via static binary analysis," in *Proceedings of the 2013 IEEE Second International Conference on Mobile Services*. IEEE Computer Society, 2013, p. 86.

[11] Z. R. Fang, S. W. Huang, and F. Yu, "Appreco: Behavior-aware recommendation for ios mobile applications," in *2016 IEEE International Conference on Web Services (ICWS)*, June 2016, pp. 492–499.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for real-time privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[13] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *36th International Conference on Software Engineering, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 1036–1046.

[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, p. 29.

[15] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: taming reflection to support whole-program analysis of android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 318–329.

[16] P. d. B. SILVA FILHO, "Static analysis of implicit control flow: resolving java reflection and android intents," 2016.

[17] DroidBench, "Droidbench benchmarks," https://github.com/secure-software-engineering/DroidBench.

[18] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11, 2011, pp. 3–14.

[19] C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM, 2012, pp. 1457–1462.

[20] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets,” in *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, ser. NDSS’12, 2012.

[21] D. Babić, D. Reynaud, and D. Song, “Malware analysis with tree automata inference,” in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV’11, 2011, pp. 116–131.

[22] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, “Pios: Detecting privacy leaks in ios applications.” in *NDSS*, 2011, pp. 177–183.

[23] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[24] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, “Byteweight: Learning to recognize functions in binary code,” in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC’14. USENIX Association, 2014, pp. 845–860.

[25] X. Meng and B. P. Miller, “Binary code is not easy,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. ACM, 2016, pp. 24–35.

[26] T. Reinbacher and J. Brauer, “Precise control flow reconstruction using boolean logic,” in *Proceedings of the Ninth ACM International Conference on Embedded Software*, ser. EMSOFT ’11. ACM, 2011, pp. 117–126.

[27] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, “BAP: A binary analysis platform,” in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 463–469.

[28] Hex-Rays, “IDAPro,” https://www.hex-rays.com/products/ida.

[29] Dynist, "Dynist: Tools for binary instrumentation, analysis, and modification," https://github.com/dyninst.

[30] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, ser. ICISS '08, 2008, pp. 1–25.

[31] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium (SAS)*, ser. LNCS, vol. 2694. Springer-Verlag, June 2003, pp. 1–18, available from `http://www.brics.dk/JSA/`.

[32] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on.* IEEE, 2004, pp. 645–654.

[33] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th international conference on World Wide Web.* ACM, 2005, pp. 432–441.

[34] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *ACM Sigplan Notices*, vol. 42, no. 6. ACM, 2007, pp. 32–41.

[35] ——, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 171–180. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368112

[36] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "String constraints for verification," in *International Conference on Computer Aided Verification.* Springer, 2014, pp. 150–166.

[37] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "A dpll (t) theory solver for a theory of strings and regular expressions," in *International Conference on Computer Aided Verification.* Springer, 2014, pp. 646–662.

[38] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A z3-based string solver for web application analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* ACM, 2013, pp. 114–124.

[39] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *Proceedings of the eighteenth international symposium on Software testing and analysis.* ACM, 2009, pp. 105–116.

[40] G. Li and I. Ghosh, "Pass: String solving with parameterized array and interval automaton," in *Haifa Verification Conference.* Springer, 2013, pp. 15–31.

[41] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra, "Automata-based symbolic string analysis for vulnerability detection," *Formal Methods in System Design*, vol. 44, no. 1, pp. 44–70, 2014.

[42] F. Yu, T. Bultan, and O. H. Ibarra, "Relational string verification using multi-track automata," in *International Conference on Implementation and Application of Automata.* Springer, 2010, pp. 290–299.

[43] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for php," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2010, pp. 154–157.

[44] H.-E. Wang, T.-L. Tsai, C.-H. Lin, F. Yu, and J.-H. R. Jiang, *String Analysis via Automata Manipulation with Logic Circuit Representation.* Cham: Springer International Publishing, 2016, pp. 241–260. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-41528-4_13

[45] F. Yu, M. Alkhalaf, and T. Bultan, "Patching vulnerabilities with sanitization synthesis," in *Proceedings of the 33rd International Conference on Software*

Engineering, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 251–260. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985828

[46] F. Yu, C.-Y. Shueh, C.-H. Lin, Y.-F. Chen, B.-Y. Wang, and T. Bultan, "Optimal sanitization synthesis for web application vulnerability repair," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 189–200. [Online]. Available: http://doi.acm.org/10.1145/2931037.2931050

[47] "Ida: About - hex-rays," http://www.hex-rays.com/products/ida, (Visited on 01/04/2016).

[48] F. Yu, C.-Y. Shueh, C.-H. Lin, Y.-F. Chen, B.-Y. Wang, and T. Bultan, "Optimal sanitization synthesis for web application vulnerability repair," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 189–200.