MobileFindr: Function Similarity Identification for Reversing Mobile Binaries

Yibin Liao, Ruoyan Cai, Guodong Zhu, Yue Yin, and Kang Li

University of Georgia, Athens, GA, USA
{liao, ruoyan, guodong, yin, kangli}@cs.uga.edu

Abstract. Identifying binary code at function level has been applied to a broad range of software security applications and reverse engineering tasks, including patch analysis, vulnerability assessment, code plagiarism detection, malware analysis, etc. However, various anti-reverse engineering techniques (e.g., obfuscation, anti-emulator, etc.) employed by the mobile apps make existing approaches ineffective when performing function identification. In this paper, we propose MobileFindr, an on-device trace-based function similarity identification framework on the mobile platform. MobileFindr runs on real mobile devices and mitigates many prevalent anti-reversing techniques by extracting function execution behaviors via dynamic instrumentation, then characterizing functions with collected behaviors and performing function matching via distance calculation. We have evaluated MobileFindr using real-world top-ranked mobile frameworks and applications. The experimental results showed that MobileFindr outperforms existing state-of-the-art tools in terms of better obfuscation resilience and accuracy.

Keywords: Reverse engineering · Similarity identification · Dynamic instrumentation.

1 Introduction

With the general availability of closed-source applications, there is a need to identify function similarity among binary executables. For instance, in the automatic patch-based exploit generation, detecting the function similarity/difference between a pre-patch binary and post-patch binary reveals the patched vulnerability [22–24, 41], and such information can be explored automatically within a few minutes [19], and generate 1-day exploits [39]. Performing function similarity measurement between intellectual property protected software binaries and suspicious binaries indicate potential cases of software plagiarism [26, 32, 34, 43, 44]. Detecting similar malicious functionality between different binary malware samples is another appealing application emerged in malware analysis, since the majority of malware samples are not brand new program but rather repacks or evolutions of previous known malicious function code [31, 35].

An inherent challenge shared by the above applications is the absence of source code. Binary executable becomes the only available resource to be analyzed. A number of semantics-aware binary differencing or function similarity

detecting methods have been proposed. One category is to use static analysis, which is usually based on control-flow graph (CFG) comparison [22–24, 46]. At a high level, the CFG based approach extracts various robust features for a node in the control flow graph [22, 24], or learns higher-level numeric feature representations from the control flow graph [23], or converts the control flow graph into embeddings [46], then perform similarity searching for the target functions. Although these studies have demonstrated that CFG based methods can be effective and scalable, all of these methods exclude obfuscated binaries, which appeared in a large number of mobile apps. Basic block semantics modeling is another approach for similarity measurement [25, 34, 41]. It represents the input-output relations of a basic block as a set of formulas, and then use theorem prover to perform the equivalence checking. However, the theorem prover is computationally expensive and impractical for large code bases of many real world mobile apps [22].

Another category relies on dynamic analysis, which is usually based on runtime execution behavior comparison. For example, previous work by Ming et al. achieves this by collecting system or API calls to slice out corresponding code segments and then check their equivalence with symbolic execution and constraint solving [35]. However, their trace logging component is an emulator based system, which cannot handle the environment-sensitive mobile apps that can detect sandbox environment. Egele et al. built a system called BLEX to capture the side effects of functions during execution [21]. Xu et al. built a tool called CryptoHunt to capture the specific features of cryptography functions with boolean formula [45]. All of their implementation are based on Intel's Pin framework [33], which is not work on mobile platforms generally with ARM instruction set architecture.

In this paper, we aim at improving the state of the art by proposing *trace-based function similarity mapping*, a hybrid method to efficiently search for similar functions in mobile binaries. Regardless of the optimization and obfuscation difference, similar code must still have semantically similar execution behavior, whereas different code must behave differently [21]. Our key idea is to capture the dynamic behavior features during the execution of a function along a runtime trace. More precisely, we propose to record a variety of dynamic runtime information as dynamic behavior features via dynamic instrumentation, and use stack backtrace information to locate corresponding functions that can be represented with these features. Then we calculate the similarity distance based on such features and return a list of similar functions ranked by the score of distance.

We have designed and implemented a system called *MobileFindr*, and evaluated it with a set of mobile examples under different obfuscation scheme combinations. Our experimental results show that our system can successfully identify fine-grained function similarities between mobile binaries, and outperform existing state-of-the-art approaches in terms of better obfuscation resilience and accuracy. Our evaluation with top-ranked real-world mobile apps also demonstrated the effectiveness of our system.

Correspondingly, our contributions in this paper are:

- We have proposed a novel approach, *trace-based function similarity mapping*, to perform function similarity measurement on mobile platforms. Our key solution is to capture observable dynamic behaviors along an execution trace via dynamic instrumentation, and characterize functions with such behaviors. Our approach exhibits stronger resilience to various anti-reverse engineering techniques for mobile apps. To best of our knowledge, this is the first work having such ability on mobile platforms.
- We have proposed a variety of dynamic features to record during the function execution, which allow us to approximate the semantics of a function without relying on the source code access.
- we have implemented a system called *MobileFindr* and source code is publicly available at GitHub: https://github.com/tigerlyb/MobileFindr.
- We have demonstrated the viability of our approach for top-ranked real-world mobile frameworks and apps.

The rest of this paper is organized as following. Section 2 introduces background and challenges. Section 3 presents the details of our system design and implementation. Section 4 presents our evaluation and results. Discussion and limitations are presented in Section 5. Then we present related work in Section 6, and conclude the paper in Section 7.

2 Background

This section introduces the background of reverse engineering, presents the popular tools that help for reverse engineering mobile apps, including various debuggers, disassemblers, decompilers, etc. Then we demonstrate motivating examples and describe possible reverse engineering challenges that can affect the state of the art function identification methods.

2.1 Reverse Engineering Mobile Apps

Reverse engineering is the process of taking a program's binary code and recreating it so as to trace it back to the original source code. It is being widely used in computer software security to enhance product features without knowing the source: find security flaws, test code compatibility, add new features or redesign the product, understand the design of malicious code, etc. In this section, we present popular reverse engineering tools for mobile apps as follows:

- **Debugger**: helps developer to understand how the program behaves at runtime without modifying the code, and allows the user to view and change the running state of a program. With the release of Xcode 5, the LLDB debugger [12], which is part of the LLVM compiler development suite, becomes the foundation for the debugging experience on Apple platforms. LLDB is

fully integrated with Xcode and provides deep capabilities in a user-friendly environment. For Android platform, both LLDB and JDB (Java debugger) are integrated in the Android Studio debugger [1]. By default, Android Studio automatically choose the best option for the code you are debugging. For example, if you have any C or C++ code in the project, Android studio debugger select LLDB to debug your code. Otherwise, Android Studio uses the Java debug type.

- **Disassembler**: a software tool which transforms binary code into a human readable mnemonic representation called assembly language. Many disassemblers are available on the market, both free and commercial. Apktools [2] and baksmali [15] are free tools that can disassemble the dex format used by Dalvik, Android's Java VM implementation. The most powerful commercial disassembler is IDA Pro [9], published by Hex-Rays. It can handle binary code for a huge number of processors and has open architecture that allows developers to write add-on analytic modules.
- **Decompiler**: a software tool used to revert the process of compilation. Decompilers are different from disassemblers in one very important aspect. While both generate human readable text, decompilers generate much higher level text, which is more concise and much easier to read. For example, Android developer can use Dex2jar [5] to convert dex file to class file, and then open it in JD-GUI [10] to display Java source code. Hex-Rays Decompiler [8] is a IDA Pro extension that converts native processor code into human readable C-like pseudocode text.

2.2 Challenges

The software security community relies on such reverse engineering tools to analyze and validate programs. However, various anti-reverse engineering techniques employed by the latest mobile apps make existing reverse engineering tools ineffective. For instance, the anti-debugging and anti-emulator techniques employed by mobile apps limit the usage of many dynamic analysis tools [28, 30, 40]. Code obfuscation scheme provide strong protection against automated static reverse engineering tools. Moreover, different mobile apps tend to use different obfuscation techniques and even same app changes obfuscation options when updating its version. In this paper, we focus on analyzing iOS apps. Nowadays iOS developers heavily rely on code obfuscation to evade detection since iOS is a close-source platform. Therefore, in this section, we introduce different code obfuscation features as well as motivating examples for understanding each features.

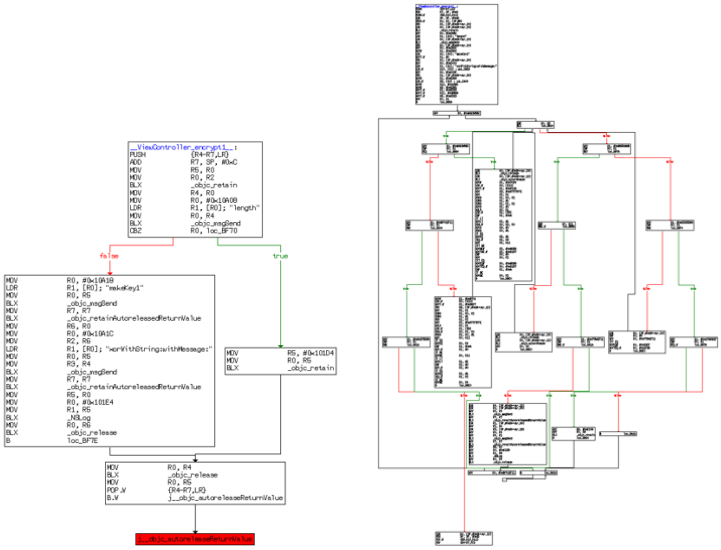
Code Obfuscation Obfuscation aims at creating obfuscated code that is difficult for humans to understand. Obfuscation techniques include modifying names of classes, fields, and methods, reordering control flow graphs, encrypting constant strings, inserting junk code, etc. To obfuscate mobile apps, we rely on a state-of-the-art open-source obfuscation tool, Obfuscator-LLVM 4.0 [29], which supports popular obfuscation transformations as follows.

– **Control Flow Flattening**: The purpose of this pass is to completely flatten the control flow graph of a program. The flag option *-split* activates basic block splitting, which improve the flattening when applied together.
– **Instructions Substitution**: The goal of this obfuscation technique simply consists in replacing standard binary operators (like addition, subtraction or boolean operators) by functionally equivalent, but more complicated sequences of instructions.
– **Bogus Control Flow**: This method modifies a function call graph by adding a basic block before the current basic block. This new basic block contains an opaque predicate and then makes a conditional jump to the original basic block. The original basic block is also cloned and filled up with junk instructions chosen at random.

```
1.  - (NSString *)encrypt1:(NSString *)message {
2.      if ([message length] == 0) {
3.          return @"NULL";
4.      }
5.
6.      NSString *key = [self makeKey1];
7.      NSString *encryptedMsg = [self xorWithString:key
    withMessage:message];
8.
9.      NSLog(@"encrypt1: %@", encryptedMsg);
10.     return encryptedMsg;
11. }
```

Fig. 1: A Motivating Example: Code

Obfuscation Example We use the example in Figure 2 to illustrate code obfuscation on iOS platform. Figure 1 shows the Objective-C source code of a function called *encrypt1*. It takes a string message as input and xor the message with a key, then return the encrypted message. Figure 2a shows the original control flow graph without any obfuscation, which only contains 4 basic blocks. While Figure 2b is the obfuscated version (combined all three obfuscation options above) of that function. As mentioned in Section 1, existing static approaches that rely on control flow graph similarity and basic block level comparison will likely not be able to make a meaningful distinction in this scenario. Alternative



(a) (b)

Fig. 2: A Motivating Example: CFG

approaches, such as dynamic approaches, either rely on Pin tool or emulator-based system to capture execution behavior. Pin tool is not able to work on analyzing most mobile apps, since ARM processors dominate mobile platforms. The anti-emulator techniques employed by mobile apps also limit the usage of such emulator-based analysis system. To address the above mentioned challenges in the scope of matching function for mobile binaries, we design a novel on-device dynamic instrumentation system.

3 Design and Implementation

In this section, we first illustrate the design of our approach, and then detail the implementation of our system.

3.1 Overview

We present *trace-based function similarity mapping*, a hybrid method to efficiently search for similar functions in mobile binaries. More precisely, we propose to record a variety of dynamic behavior features during the execution of a function along an execution trace. We define the concept of "dynamic behavior features" broadly to include any information that can be derived from observations made during execution. Our approach works as the following: given two mobile apps A, B and a function of interest F from A. Both F and any executed functions from B are characterized with dynamic behavior features. Then

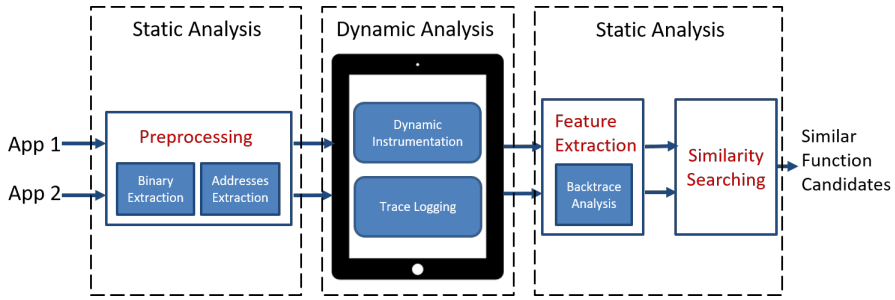


Fig. 3: Schematic Overview of Trace-based Function Similarity Mapping System

we compute similarity scores between F and each function f from B, to identify which functions in B are similar to F. The novelty of our approach lies in the follows.

- What features are useful for semantic similarity comparisons?
- How these features are captured on mobile platforms?
- How to characterize a function with such features?

Figure 3 illustrates the architecture of our system, which comprises four stages: preprocessing, on-device dynamic analysis, feature extraction and similarity searching. The preprocessing stage, as shown in the left side of Figure 3, involves two parts: binary extraction and address extraction. It dumps the mobiles binaries from the app and extract addresses for all functions and imported libraries and frameworks. All the extracted addresses are passed to the on-device dynamic analysis stage for instrumentation and trace logging usage. The recorded traces will be analyzed by the feature extraction stage. Then we perform the similarity searching based on the function features obtained from feature extraction stage. Next, we will present each step of our system in the following sections.

3.2 Preprocessing

Binary Extraction When you download an iOS app from the App Store, Apple injects a special 4196 byte long header into the signed binary encrypted with the public key associated with your iTunes account. For this step we choose Clutch [4], to decrypt and dump app binary. Then we disable the ASLR (Address Space Layout Randomization) to get the correct function addresses. ASLR makes the remote exploitation of memory corruption vulnerabilities significantly more difficult by randomizing the application objects location in the memory. By default iOS apps are compiled with *-pie* flag (Generate Position-Dependent Code). This flag is automatically checked in the latest version of *Xcode* in order to use ASLR. We leverage the tool *removePIE* [6] to disable the ASLR by flipping the PIE flag. After that, we put the binary back to the app and re-sign it with *ldid* [11].

Address Extraction We utilize IDA Pro [9] to disassemble the binary obtained from previous step, extract function addresses as well as imported library addresses and framework addresses through IDAPython API. This component is implemented with 155 lines of Python code. Listing 1.1 shows an example of a function address table extracted from the iOS app binary. Each line consists of starting address (e.g., 0x11834), ending address (e.g., 0x11980) and function name (e.g, *prepareToRecord* from the class *MovieRecorder*). Listing 1.2 shows an example of library addresses, which only consist the starting addresses and library names.

Listing 1.1: Function Addresses

```
...
0xb7ea,0xb964,-[VideoSnakeViewController
    toggleRecording:]
0xe2cc,0xe51c,-[VideoSnakeSessionManager
    startRecording]
0x111d8,0x1128c,-[MovieRecorder initWithURL:]
0x1161c,0x116a8,-[MovieRecorder delegate]
0x11834,0x11980,-[MovieRecorder prepareToRecord]
0x11d48,0x11ebc,-[MovieRecorder finishRecording]
...
```

Listing 1.2: Library Addresses

```
...
0x1606c,__Block_copy
0x1607c,__Block_object_assign
0x1608c,__Block_object_dispose
0x1609c,__Unwind_SjLj_Register
0x160ac,__Unwind_SjLj_Resume
0x160bc,__Unwind_SjLj_Unregister
...
```

3.3 On-device Dynamic Analysis

The on-device dynamic analysis stage performs dynamic instrumentation and trace logging in order to record the needed information.

Dynamic Instrumentation We utilize Frida [7], a dynamic instrumentation toolkit, to inject scripts in app process that monitor the dynamic behavior during execution. Frida lets you inject snippets of JavaScript or your own library into native apps. Frida's core is written in C and injects Googles V8 engine into the target processes, where the JavaScript gets executed with full access to memory, hooking functions and even calling native functions inside the process.

Trace Logging In our implementation we chose features that capture a variety of system level information (e.g., system calls), as well as higher level attributes, such as libc calls, objc calls, framework API invocations as follows.

- **System Calls**: e.g., *read*, *write*, *open*, etc. defined in *libsystem_kernel.dylib*
- **Library Calls**: e.g., *memset*, *memcpy*, *free*, etc. defined in *libSystem.B.dylib*, *_objc_getClass*, *_objc_getProtocol*, etc. defined in *libobjc.A.dylib*
- **Framework APIs**: e.g., *OpenGLES*, *CoreMedia*, *UIKit*, etc.

We leverage the Frida API to inject JavaScript at the library addresses and framework addresses to record the invocations of such features above, and generate a backtrace for the current thread, returned as an array of native pointer addresses for the subsequent steps.

3.4 Feature Extraction

Listing 1.3 illustrates the logged trace data, which consists of arrays of addresses. Each line indicates an invocation of library call or framework API call, followed by its stack backtrace information. First, we transform the addresses to function names according to the address table obtained from the preprocessing stage. For instance, 0x1609c is the starting address of *__Unwind_SjLj_Register*, 0x11892 is in the range of 0x11834 and 0x11980, which indicate the library *__Unwind_SjLj_Register* is called by function *prepareToRecord*. The rest can be done in the same manner. Listing 1.4 illustrates a full translated results from Listing 1.3.

Listing 1.3: Stack Backtrace: Address

```
...
0x1609c,0x11892,0xe498,0xb92e,0xb15a
0x1621c,0x118c0,0xe498,0xb92e,0xb15a
0x1620c,0x118fc,0xe498,0xb15a
...
```

Listing 1.4: Stack Backtrace: Name

```
        ...
        __Unwind_SjLj_Register,-[MovieRecorder
            prepareToRecord],-[VideoSnakeSessionManager
            startRecording],-[VideoSnakeViewController
            toggleRecording:],sub_B120
        _dispatch_get_global_queue,-[MovieRecorder
            prepareToRecord],-[VideoSnakeSessionManager
            startRecording],-[VideoSnakeViewController
            toggleRecording:],sub_B120
        _dispatch_async,-[MovieRecorder prepareToRecord],-[
            VideoSnakeSessionManager startRecording],-[
            VideoSnakeViewController toggleRecording:],
            sub_B120
        ...
```

Next, we match these library calls or framework API calls to its corresponding caller functions as features. Listing 1.5 represents features of function *prepareToRecord*, in JSON format. The feature extraction component is implemented with 280 lines of Python code.

3.5 Similarity Searching

The function feature representation is a length-N feature list. We chose Jaccard index to measure the similarity between lists. We define sim(f, g) to be the similarity score between function f and g. We perform similarity searching as the following: starting with a known reference function in a trace, we are searching for mobile binaries containing similar functions by calculating similarity score and listing top K similar function candidates.

Listing 1.5: Function Features

```
{
        "name" : "-[MovieRecorder prepareToRecord]",
        "features" : [
                [
                        "__Unwind_SjLj_Register",
                        "_dispatch_get_global_queue",
                        "_dispatch_async",
                        "__Block_object_assign",
                        "__Unwind_SjLj_Unregister"
                ]
        ]
}
```

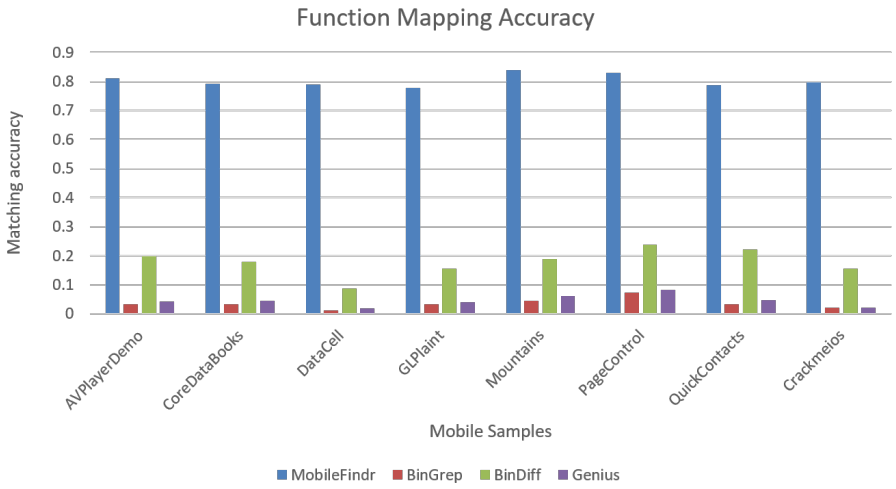


Fig. 4: Function Mapping between Obfuscated Version and Non-obfuscated Version

4 Evaluation

In this section, we evaluate our system from several objectives. Particularly, we conduct our experiments to evaluate whether our system outperforms existing binary similarity detection tools in terms of better obfuscation resilience and accuracy. We designed two controlled datasets so that we have a ground truth to assess comparison results accurately. We also evaluate the effectiveness of our system in analyzing real world top-ranked iOS apps from Apple App Store.

4.1 Experiment Setup

Our on-device dynamic analysis is performed on a 32GB Apple Jailbroken iPad (4th Generation) running iOS 8.3. The configuration of our testbed machine for feature extraction and similarity searching is shown as follows.

- CPU: Intel Core i7-6700K Processor (Eight-core with 4.00GHz)
- Memory: 64GB
- OS: Ubuntu Linux 14.04 LTS
- Python Version: 2.7.12
- IDA Pro Version: 6.6

4.2 Ground Truth Dataset

Data 1 First, we collect 8 sample codes with different functionalities from official Apple developer website. For each sample we build both non-obfuscated version and obfuscated version. The obfuscated version combines all three settings in Table 1.

Data 2 Then we test our system with third-party frameworks or libraries that are commonly used by popular mobile apps. In practice, programmers usually take advantage of existing frameworks or libraries to speed up their developments. In our evaluation, we choose *AFNetworking* and *SDWebImage*, top-two ranked open source frameworks [16] as the reference implementation. Our purpose is to detect such frameworks or libraries that commonly used in different mobile apps. To this end, we collect 8 open source projects from GitHub, and reuse the provided APIs from two libraries above. We built sample apps with non-obfuscated version and 7 different combinations of the obfuscation settings, which results in 64 apps in 8 different types. We kept the debug symbols as they provide a ground truth and enable us to verify the correctness of matching using the functions symbolic names.

Table 1: Different Obfuscation Types and Flag Settings

	Type	Flag Setting
1	control flow flattening	-fla, -split, -split_num=3
2	instruction substitution	-sub, -sub_loop=3
3	bogus control flow	-bcf, -bcf_loop=3, -bcf_prob=40

4.3 Obfuscation Options

As mentioned in section 2, we use Obfuscator-LLVM to obfuscate our ground truth mobile samples. Table 1 lists specific obfuscation settings that we use to build our ground truth iOS samples. We integrate Obfuscator-LLVM into Xcode, and enable the three obfuscation features described in Section 2, and apply different settings as shown in Table 1.

4.4 Peer Tools

We compare our tools with other state-of-the-art similarity detection or diffing tools that open to public: BinDiff, BinGrap, Genies. BinDiff [17] is a comparison tool for binary files, that assists vulnerability researchers and engineers to quickly find differences and similarities in disassembled code. BinGrap [3] is also a static analysis tool that perform function similarity searching, but it can output a list of functions in order of similarity. Genius is a bug search engine that performs function similarity detection based on mapping raw features of a function into a higher-level numeric vector where each dimension of the vector is the similarity distance to a categorization in the codebook. However, only partial code is available, including raw feature extraction and search. Therefore, we re-implement Genius' two core steps, codebook generation and feature encoding in python. We utilized Hungarian algorithm for calculating bipartite graph matching cost and normalized spectral clustering [38] for ACFGs (Attributed Control Flow

Graph) clustering. In evaluation phrase, we adopt Nearpy [14] for LSH (Locality Sensitive Hashing) [18] and search. We used SQLite to store function information and encoded vectors. As mentioned in section 1, BLEX [21], BinSim [35] and CryptoHunt [45] are not able to work on iOS platforms. To the best of our knowledge, we are the first to propose a dynamic strategy for comparing mobile binary code. This is the reason why we did not compare our evaluation to these dynamic approaches.

4.5 Evaluation Results

The first evaluation for data 1 is shown in Figure 4. For each sample, We randomly select functions from non-obfuscated version as reference functions, then perform our *trace-based function similarity mapping* to see if we can locate the same function in obfuscated version. The second evaluation for data 2 is shown in Figure 5. We randomly select one app from each type of apps as reference known app, and select commonly used APIs in *AFNetworking* and *SDWebImage* from that app as query functions. Then we perform *trace-based function similarity mapping* for searching the given functions in the rest apps, and list top K candidates for each app based on the similarity score. We only compare with Genius and BinGrep since BinDiff is a one-to-one mapping tool, which cannot list more than 1 candidate.

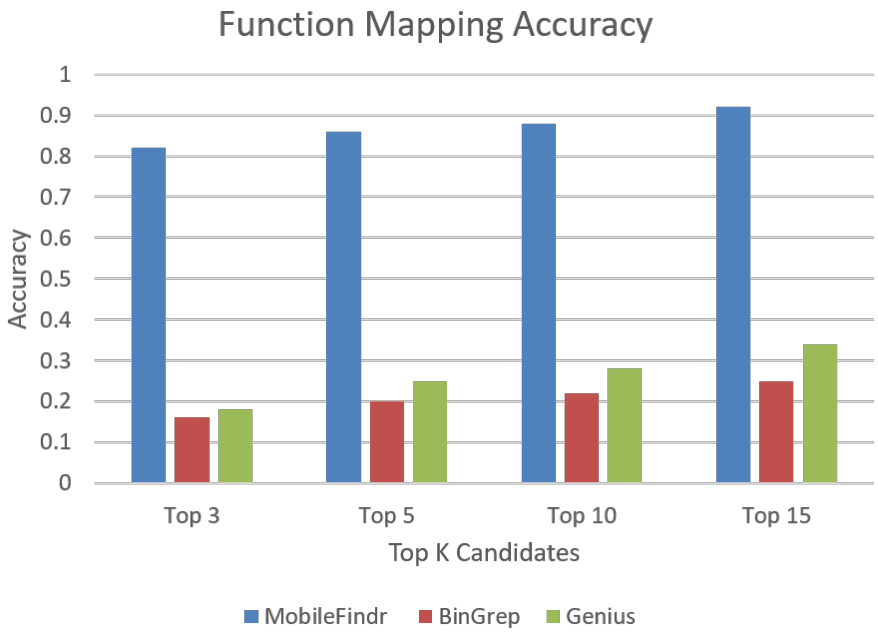


Fig. 5: Function Mapping Evaluation for Popular Third-party Frameworks

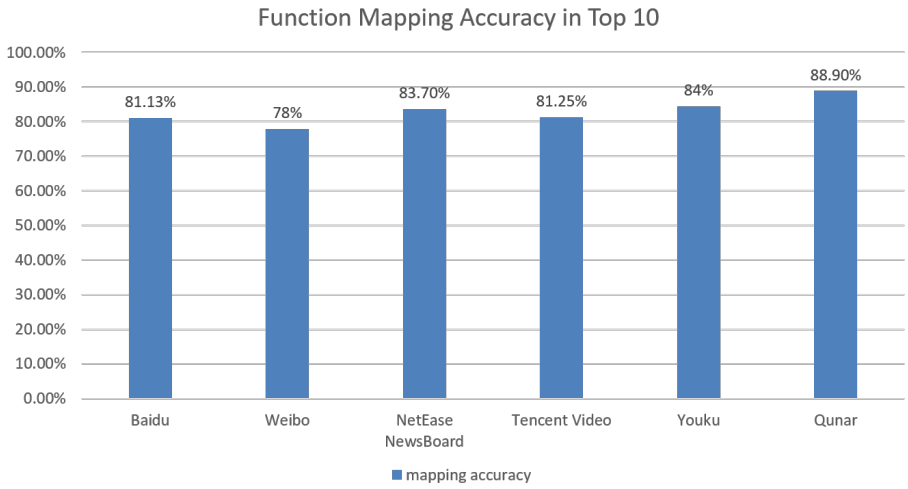


Fig. 6: Function Mapping Evaluation in Real-world Apps

Our evaluation results show that MobileFindr can achieve more than 80% accuracy in average from top 3 to top 15 similar functions, which outperforms other tools in terms of much more better accuracy and obfuscation resilience.

4.6 Real-world App Case Study

We tested MobileFindr using real-world apps to evaluate its efficiency. We evaluated 6 top-ranked iOS apps in different types, such as search engine, social networking, etc. For instance, Baidu is the world's largest Chinese search engine. We downloaded two different versions of Baidu app, version 930 and version 935. We chose version 930 as reference app and performed a simple web searching with key words: "security" for trace logging. We collected 430 functions in this trace, and then perform *trace-based function similarity mapping* to search similarity functions in the new version 935, and listed top 10 similar function candidates. MobileFindr achieve 81.13% accuracy with less than 10 minutes. While matching the same 430 functions in Genius, it only achieved 59.7% accuracy, but spent around 2 hours in training, more than 40 hours when handling function graph embeddings. Figure 6 shows the function mapping results for the 6 real-world apps.

5 Discussion

In this section, we discuss the limitations of our system and potential solutions to be investigated in future work.

First, a challenge that we already touched upon in Section 4 is the fact that our approach needs manual verification efforts for real world iOS apps, since

we don't have access to their source code. The candidate similarity ranking produced by our system gives an ordered list of matched functions that have to be manually inspected by an analyst to verify if those functions are actually similar. Some of the existing dynamic approaches [35, 45] rely on symbolic execution to generate a set of symbolic formula, and then use theorem prover to perform the equivalence checking. However, the theorem prover is computationally expensive and impractical for large code bases of many real world mobile apps. Such an automatic verification would be ideal, but surely is a research topic in itself and is outside the scope of this work.

Second, the incomplete path coverage is a concern for all dynamic analysis system, including ours. The possible solutions are to explore more paths by automatic input generation [27, 36]. To trigger as many dynamic behaviors as possible for trace logging, we can leverage the idea of Malton [47], which proposed an efficient path exploration technique that employs in-memory concolic execution with an offloading mechanism and direct execution engine. We leave it as future work.

Third, the functions considered by us need to have a certain amount of complexity for the approach to work effectively. Otherwise, the relatively low combination number of library calls leads to a high probability for collision. Hence, we only considered functions with at lease five basic blocks, as noted in Section 4. For instance, the potential for bugs in small functions, however, is significantly lower than in large functions, as shown in [13]. Hence, in a real-world scenario this should be no factual limitation.

6 Related Work

There has been a substantial research on detecting binary code similarity. Existing semantics aware binary matching techniques can be classified into two categories. One is based on static information including numeric features and structural features [20, 22, 23, 34]. Many numeric features (e.g. the number of basic blocks, the number of edges, logic instructions,local variables, etc) and control flow graph has been demonstrated to be robust across compilers and different compile options in previous work [24, 25]. The other one executes target code and collect runtime behavior [21, 35, 42, 45]. Common execution behaviors includes stack and heap memory access, system call sequences and library calls, registers values, execution path, etc.

The combination of collected features represent as a signature of target code for matching step. It is vital to identify robust features and correctly characterize target code with the features. Bindiff [17] as an efficient binary diffing tool using a graph theoretic approach to find similarities and differences. The graph isomorphism detection on pairs of function works well when two semantically equivalent binaries have similar control flow. But CFG changes across architectures and compilers. In [23], Genius maps raw features of a function into a higher-level numeric vector where each dimension of the vector is the similarity distance to a categorization in the codebook. However, one common limitation

of static approaches is incapable of handling obfuscated code. BLEX [21] collects execution side effects during function execution and uses a multidimensional vector as function signature for similarity assessment. It relies on Pin framework and can not apply to mobile binaries.

The techniques of binary matching have been driven towards to solve security problems. One common case in vulnerability assessment is that secure analysts would want to use a sample of vulnerable binary without source code to search for the similar bug across all the softwares installed in the company devices [22,37]. It is challenging for vulnerability assessment in a large code base for the following reasons: first, most commercial software projects are closed-source and only available in the binary form without debug information. Second, different versions of software may be compiled on different optimization levels and different compile tool-chain, which would radically changes both the number of nodes and structure of edges in both the control flow graph and the call graph. Third, pervasive code protection schemes, such as class and method rename, encryption of strings, control flow obfuscation and virtualization of code, render code analysis time consuming. Our evaluation have considered above situations and demonstrate that our approach can handle it.

With rapid development of open-source projects, the similarity between an licensed protected binary and a suspicious binary indicates a potential case of software plagiarism [34,43]. Existing code similarity measurement methods have been proved to be useful but remain far from perfect. Some software plagiarism detection approaches based on dynamic system call sequences have also been proposed [32,43], but they incur false negatives when the number of system calls are insufficient or when system call replacement is applied. Most of the existing methods are not effective in the presence of obfuscation techniques. Another obfuscation resilient method [34] based on symbolic execution and theorem proving bears high computational overhead.

7 Conclusion

We proposed MobileFindr, an on-device trace-based function similarity mapping system for reverse engineering mobile apps. It records a variety of dynamic runtime information as dynamic behavior features via dynamic instrumentation, and use stack backtrace information to locate corresponding functions that can be represented with these features. We evaluated it with a set of examples under different obfuscation scheme combinations. Our experimental results show that our system can successfully identify fine-grained function similarities between mobile binaries, and outperform existing state-of-the-art approaches in terms of better obfuscation resilience and accuracy. Our evaluation with top-ranked real-world frameworks and apps also demonstrated the effectiveness of our system. To the best of our knowledge, we are the first to propose a dynamic strategy for function similarity identification on the mobile platform, which is capable of mitigating many anti-reverse engineering techniques.

References

1. Android studio - debug your app. https://developer.android.com/studio/debug/index.html, accessed: 2018-01-30
2. Apktool - a tool for reverse engineering android apk files. https://ibotpeaches.github.io/Apktool/, accessed: 2018-01-30
3. Bingrep. https://github.com/hada2/bingrep, accessed: 2018-01-30
4. Clutch 2.0.4. https://github.com/KJCracks/Clutch/releases/tag/2.0.4, accessed: 2018-01-30
5. dex2jar. https://github.com/pxb1988/dex2jar, accessed: 2018-01-30
6. Disable aslr on ios applications. http://www.securitylearn.net/2013/05/23/disable-aslr-on-ios-applications/, accessed: 2018-01-30
7. Frida. https://www.frida.re/, accessed: 2018-01-30
8. Hex-rays decompiler. https://www.hex-rays.com/products/decompiler/index.shtml, accessed: 2018-01-30
9. Ida. https://www.hex-rays.com/products/ida/index.shtml, accessed: 2018-01-30
10. Jd-gui. http://jd.benow.ca/, accessed: 2018-01-30
11. ldid. http://iphonedevwiki.net/index.php/Ldid, accessed: 2018-01-30
12. The lldb debugger. https://lldb.llvm.org/, accessed: 2018-01-30
13. More complex = less secure: Miss a test path and you could get hacked. http://www.mccabe.com/pdf/MoreComplexEqualsLessSecure-McCabe.pdf, accessed: 2018-01-30
14. Nearpy. https://github.com/pixelogik/NearPy, accessed: 2018-01-30
15. smali/baksmali wiki. https://github.com/JesusFreke/smali/wiki, accessed: 2018-01-30
16. Top 10 libraries for ios developers. https://www.raywenderlich.com/177482/top-10-ios-developer-libraries, accessed: 2018-01-30
17. Zynamics bindiff. https://www.zynamics.com/bindiff.html, accessed: 2018-01-30
18. Andoni, A., Indyk, P.: Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on. pp. 459–468. IEEE (2006)
19. Brumley, D., Poosankam, P., Song, D., Zheng, J.: Automatic patch-based exploit generation is possible: Techniques and implications. In: Security and Privacy, 2008. SP 2008. IEEE Symposium on. pp. 143–157. IEEE (2008)
20. David, Y., Partush, N., Yahav, E.: Similarity of binaries through re-optimization. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 79–94. ACM (2017)
21. Egele, M., Woo, M., Chapman, P., Brumley, D.: Blanket execution: Dynamic similarity testing for program binaries and components. USENIX (2014)
22. Eschweiler, S., Yakdan, K., Gerhards-Padilla, E.: discovre: Efficient cross-architecture identification of bugs in binary code. In: NDSS (2016)
23. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 480–491. ACM (2016)
24. Flake, H.: Structural comparison of executable objects. In: Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, number P-46 in Lecture Notes in Informatics. pp. 161–174. Citeseer (2004)
25. Gao, D., Reiter, M.K., Song, D.: Binhunt: Automatically finding semantic differences in binary programs. In: International Conference on Information and Communications Security. pp. 238–255. Springer (2008)

26. Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., Choi, H.: Adrob: Examining the landscape and impact of android application plagiarism. In: Proceeding of the 11th annual international conference on Mobile systems, applications, and services. pp. 431–444. ACM (2013)
27. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: NDSS. vol. 8, pp. 151–166 (2008)
28. Herremans, D.: Morpheus: automatic music generation with recurrent pattern constraints and tension profiles (2016)
29. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM – software protection for the masses. In: Wyseur, B. (ed.) Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015. pp. 3–9. IEEE (2015). https://doi.org/10.1109/SPRO.2015.10
30. Kirat, D., Vigna, G.: Malgene: Automatic extraction of malware analysis evasion signature. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 769–780. ACM (2015)
31. Lindorfer, M., Di Federico, A., Maggi, F., Comparetti, P.M., Zanero, S.: Lines of malicious code: insights into the malicious software industry. In: Proceedings of the 28th Annual Computer Security Applications Conference. pp. 349–358. ACM (2012)
32. Liu, C., Chen, C., Han, J., Yu, P.S.: Gplag: detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 872–881. ACM (2006)
33. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Acm sigplan notices. vol. 40, pp. 190–200. ACM (2005)
34. Luo, L., Ming, J., Wu, D., Liu, P., Zhu, S.: Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 389–400. ACM (2014)
35. Ming, J., Xu, D., Jiang, Y., Wu, D.: Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In: Proceedings of the 26th USENIX Security Symposium. USENIX Association. pp. 253–270 (2017)
36. Moser, A., Kruegel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: Security and Privacy, 2007. SP'07. IEEE Symposium on. pp. 231–245. IEEE (2007)
37. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual. pp. 421–430. IEEE (2007)
38. Ng, A.Y., Jordan, M.I., Weiss, Y.: On spectral clustering: Analysis and an algorithm. In: Advances in neural information processing systems. pp. 849–856 (2002)
39. Oh, J.: Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. Black Hat. Black Hat (2009)
40. Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the Seventh European Workshop on System Security. p. 5. ACM (2014)
41. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-architecture bug search in binary executables. In: Security and Privacy (SP), 2015 IEEE Symposium on. pp. 709–724. IEEE (2015)

42. Sharma, R., Schkufza, E., Churchill, B., Aiken, A.: Data-driven equivalence checking. In: ACM SIGPLAN Notices. vol. 48, pp. 391–406. ACM (2013)
43. Wang, X., Jhi, Y.C., Zhu, S., Liu, P.: Behavior based software theft detection. In: Proceedings of the 16th ACM conference on Computer and communications security. pp. 280–290. ACM (2009)
44. Wang, X., Jhi, Y.C., Zhu, S., Liu, P.: Detecting software theft via system call based birthmarks. In: Computer Security Applications Conference, 2009. ACSAC'09. Annual. pp. 149–158. IEEE (2009)
45. Xu, D., Ming, J., Wu, D.: Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In: Security and Privacy (SP), 2017 IEEE Symposium on. pp. 921–937. IEEE (2017)
46. Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., Song, D.: Neural network-based graph embedding for cross-platform binary code similarity detection. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 363–376. ACM (2017)
47. Xue, L., Zhou, Y., Chen, T., Luo, X., Gu, G.: Malton: Towards on-device non-invasive mobile malware analysis for art. In: In 26th USENIX Security Symposium (USENIX Security 17). ACM (2017)