Obfuscate API calls in Mach-O Binary

Anh Khoa Nguyen
ng.akhoa98@gmail.com

Abstract

The Mach-O binary format serves as the primary executable format in Apple's Operating System. Applications within the Apple ecosystem are particularly susceptible to reverse engineering attacks. To counter these threats, obfuscation techniques applied to the underlying source code and data have become increasingly prevalent. However, these methods often require access to source code and custom compiler tooling. Another form of obfuscation involves direct manipulation of binary files. Currently, this form of obfuscation for Mach-O binaries is somewhat limited, typically focusing on altering strings with relatively low impact.

In this paper, we present a novel binary obfuscation approach aims specifically for Mach-O binaries. Our obfuscation method targets dynamic symbols. The absence of these symbols renders library function invocations undefined during static analysis. Additionally, we demonstrate the applicability of our obfuscation technique to Objective-C compiled applications, which is a common choice in Apple device development. Moreover, our obfuscation strategy exhibits resilience to memory extraction by not necessitating the reconstruction of the symbols table.

1 Introduction

Software obfuscation is a widely adopted technique for enhancing software protection during production. It plays a crucial role in hindering reverse engineering attempts, effectively shielding the application from easy comprehension and reducing its vulnerability to malicious actors. Beyond that, obfuscation can also serve as a deterrent against code reuse and intellectual property theft. In the past decade, there have been strong interest in the obfuscation methods to protect applications. Nowadays, obfuscation is required for high security, high privacy, high intellectual property applications. These include banking applications, machine learning/artificial intelligence applications to name a few.

Currently, obfuscation is mostly based on direct source-code manipulation or intermediate representation during the compilation process. Both of this requires high effort in creating a toolchain capable of understanding the whole project source-code. Another less common approach is through modifications of program's binaries, usually the executable. This method spans into two different ways, direct assembly instructions modification and binary format modification.

In this paper, we present an innovative approach to obfuscating Mach-O binaries, applicable to both executables and dynamic libraries. Our method distinguishes itself by primarily concentrating on the removal of the dynamic symbols table. The absence of this table significantly enhances the complexity of binary analysis, as function invocations are no longer explicitly defined during static analysis. While targeting dynamic symbols table can disrupt the loading process of the binary in memory, we take measures to ensure that the binary loading process remains uninterrupted. Additionally, we extend our obfuscation to obscure essential information frequently utilized for the static analysis, adding an extra layer of security to the obfuscated binary. Our method can also obfuscate Objective-C compiled binaries that are commonly seen in Apple's devices.

The remainder of this paper is structured as follows: Section 2 offers an in-depth exploration of the background, encompassing obfuscation techniques, binary analysis, an overview of Apple's loader, and, more specifically, the Mach-O binary format. In Section 3, we provide a detailed account of the implementation process for obfuscating Mach-O binaries. These steps include binary modifications and the restoration of essential information during runtime. We also address the nuances of obfuscating Objective-C compiled binaries and introduce additional information that can be leveraged to enhance obfuscation, along with any associated drawbacks. Finally, in Section 4, we offer our concluding remarks.

2 Background

2.1 Obfuscation

Obfuscation techniques for software have gained significant attention as they offer crucial protection against reverse engineering and intellectual property theft. These techniques aim to make programs challenging to understand and analyze. One common approach is direct source-code modification, involving transformations applied directly to the source code. Tools like Tigress C Obfuscator [12] exemplify this method, but it is language-specific and relies on the language's expressive capabilities. Another approach involves compiler-level obfuscation, which focuses on the compiler's internal processes. With the widespread use of LLVM [31], this method has gained momentum, as LLVM provides an intermediate representation (IR) before translating code into assembly instructions. Obfuscator-LLVM [26] and works integrating LLVM passes [22, 27, 29, 50, 53, 57, 61] demonstrate how to obfuscate programs by transforming LLVM IR, making it applicable to multiple programming languages.

Obfuscation techniques often categorized by their distinctive operations, such as control-flow obfuscation, opaque predicates, substitution of expressions, and code bloat. Control-flow obfuscation, a prominent approach, seeks to obscure program control flow by employing tactics like control-flow flattening [10, 25, 30, 59, 60] and the introduction of superfluous branching, rendering the program's logic more intricate and challenging to decipher. Another technique, opaque predicates [6, 11, 15, 17, 18, 37, 39, 40, 44, 56, 58], involves constructing instruction sequences that consistently yield a constant value, significantly heightening the complexity of code analysis. Substitution of expressions replaces equivalent code segments, with complex methodologies like Mixed Boolean-Arithmetic [62, 63] expressions presenting formidable challenges for comprehension. Despite efforts to simplify such expressions [19, 20, 34, 35, 48, 49], inherent limitations persist. Furthermore, obfuscation can extend to the insertion of extraneous statements that maintain the program's functionality but purposefully create convoluted code paths, diverting the efforts of reverse engineers. Beyond these categorized techniques, other obfuscation methods like function fusion [12, 61] and self-modification [12] exist although not widely used due to its complexity in implementation.

A more intricate obfuscation method, known as the Virtual Machine obfuscation technique [12, 42, 50, 52], is also widely employed. This method involves extracting the program's functionalities and reconstructing them into a different code format that is tailored to a specific virtual machine embedded within the program. Analyzing such obfuscated code is extremely challenging, as it often requires a comprehensive understanding of the virtual machine's operation before deciphering the obfuscated program's logic. This obfuscation technique can be further enhanced by incorporating multiple virtual machines within a single program or employing various encodings for the obfuscated code. However, it's important to note that implementing this technique can be highly complex and requires designing the virtual machines to vary over time to prevent adversaries from fully comprehending one machine and then reusing that knowledge.

Source-code and compiler-based obfuscation techniques have proven effective in enhancing software security. However, they encounter practical challenges related to source-code privacy. An alternative approach, binary obfuscation, addresses these concerns by targeting compiled binary files. These binaries encapsulate assembly instructions, program data, and essential execution information for the operating system. Binary obfuscation aims to eliminate critical information that reverse engineers rely on for their analysis. Nevertheless, it is essential to note that binary obfuscation is inherently platform-specific. The predominant focus has been on Windows binaries [28, 42, 52] and Linux [23, 32, 33, 46, 55] binaries, while Apple binaries have received comparatively limited attention. Some prior work [8, 45, 55] has proposed modest adjustments to debugging information within the binaries, serving to impede the analysis capabilities of reverse engineering platforms.

Another widely recognized form of binary obfuscation is known as ”packing” [41]. This method involves compressing or encrypting the binary code and then unpacking it at runtime. However, it's important to note that because the code is unpacked during runtime, it remains susceptible to memory extraction, which could potentially allow an attacker to recover the original code.

Lastly, obfuscation should be non-deterministic and unique for each application. Deterministic obfuscation can be vulnerable to pattern-based recognition and deobfuscation techniques. When different applications employ similar obfuscation strategies, it becomes possible for attackers to reuse old analysis methods on newer versions, potentially compromising the security measures in place. To maximize the effectiveness of obfuscation, it is crucial to introduce variability and uniqueness in the obfuscation process, making it more challenging for adversaries to develop generalized attack strategies.

2.2 Binary Analysis

Programs are typically compiled from human-readable source code into binary formats that are understandable by the operating system. These binary files contain data necessary for the operating system to execute the program, as well as assembly instructions that correspond to the target computer's architecture. Currently, there are three major binary file formats used in different operating systems: Portable Executable (PE) in Windows, Executable and Linkable Format (ELF) in Linux, and Mach-O in Apple devices. These binary formats are not human-readable due to their raw binary data, and specialized software, known as binary analysis tools, is used to interpret them. Both free and commercial binary analysis tools are widely employed in the industry, including Radare2/Rizin/Cutter [2, 13, 14], Ghidra [1], Binary Ninja [23], Hopper [9], JEB [51], IDA [21], among others. These tools provide analysts with insights into the binary, displaying its assembly instructions and data.

Binary analysis tools are essential for understanding binary files, but they also rely on various binary analysis techniques to extract additional information from the binary and its assembly instructions. These techniques encompass a wide range of functionalities, e.g., uncovering call graphs, determining stack values. In the context of obfuscation, these techniques are particularly focused on de-obfuscating the binary, often utilizing approaches such as taint analysis and symbolic execution to unravel the obfuscated code's logic and behavior.

2.3 Apple's loader `dyld`

The Apple loader [3], known as `dyld`, is responsible for execution of programs, including loading the binary and its associated libraries, resolving dynamic symbols, rebasing offsets, and ultimately executing the binary.

Due to the shared-cache mechanism [24] introduced in iOS 13.5 or macOS 11.0, important libraries, including system (often referred to as `libSystem` [4]), C++, Objective-C Runtime [5], Foundation, and Swift Runtime libraries, are loaded into memory and are only available there. In older versions of Apple's operating systems, direct file system access to the `dyld` loader was possible. However, in recent versions, such access is no longer feasible as `dyld` and these libraries are now exclusively in memory since system boot.

2.4 Mach-O binary format

The Mach-O binary format is inherently complex. To gain a comprehensive understanding of our proposed obfuscation technique, it is imperative that we closely examine this binary format. It is crucial to emphasize that our obfuscation methodology does not pertain to the obfuscation of the binary code itself. Instead, our focus lies in the obfuscation of vital information stored within the binary file. Therefore, a thorough comprehension of how the Mach-O binary format stores these information is of significance in understanding our approach.

2.4.1 Basic Mach-O structure

The Mach-O binary format can be comprehensively examined from multiple perspectives. One fundamental approach is to dissect it based on its encoding of binary data. In this regard, a Mach-O binary comprises a header, a sequence of load commands, and subsequent raw binary data. The header provides essential information about the binary, encompassing its type (whether executable or library), endianness, architecture, and the number of load commands. Load commands, crucial for the loader's runtime operations, facilitate the mapping of the binary into memory and the execution of preliminary tasks. Some load commands reference the raw binary data.

Alternatively, another perspective to comprehend the Mach-O binary is through its segmentation. Typically, the binary consists of three key segments: `_TEXT` contains assembly instructions; `_DATA` and `_DATA_CONST` store static binary data; `_LINK_EDIT` segment is dedicated to loader instructions.

2.4.2 Dynamic library load chain

In most cases, programs cannot function as standalone entities but rely on dynamic libraries. These libraries are registered in the header of Mach-O binaries using commands like `LC_DYLIB` (or similar equivalents). These commands establish a load chain, organized in a specific order, and recursively link each item. In addition to these explicit commands, the loader also dynamically loads libraries that are essential for the binary's runtime operation. These may include standard libraries, Foundation libraries, as well as Objective-C and Swift libraries, among others.

The loader is responsible for locating and loading libraries into memory. These libraries fall into different categories: system installed libraries and, user-provided libraries are identified by their names within the `LC_DYLIB` load command. These names can represent either full or relative paths. Full paths are self-explanatory, whereas relative paths can be more intricate, involving file system-relative paths or the use of *rpath* variables. There are three rpath variables: `@executable_path`, `@loader_path`, and `@rpath`. They serve as references to libraries, with `@executable_path` pointing to the location of the executable, `@loader_path` indicating the loader's location, and `@rpath` being defined through a series of `LC_RPATH` commands. Libraries using the `@rpath` reference will be iteratively replaced through each item in the `LC_RPATH` chain to search for the corresponding on-disk file.

System installed libraries, providing system APIs, C++ APIs, and Objective-C APIs, can be accessed through specific paths like `/usr/lib/libSystem.B.dylib`, `/usr/lib/libc++.1.dylib`, `/usr/lib/libobjc.A.dylib`. Foundation libraries are made available via `/System/Library/Frameworks/CoreFoundation.framework/*`. Swift libraries can be found and referenced at the path `/usr/lib/swift/*`. It should be noted that these libraries are only available in memory, and direct file-system access is unavailable due to the shared-cache mechanism.

2.4.3 Dynamic Symbols

Functions from external libraries are often used as a means of code reuse. When a binary does not statically link with a library, it must specify the required library and functions statically in its binary format and will be resolved at runtime. This approach to code reuse is known as dynamic loading. In Mach-O binaries, all the information necessary for dynamic loading, usually referred to as import table, is spread across various segments, including `_LINK_EDIT`, `_DATA`, and `_DATA_CONST`. The import table in Mach-O has undergone several updates over time. The original version of the import table used a custom bytecode chain, while the updated version introduced in iOS 14 employs fix-ups chains.

During both load time and runtime, the loader of a Mach-O binary reads the import table, searching for the addresses of symbols, and rewrites them in memory for reference by the executable or library code. To facilitate this functionality, the binary allocates space for a list of stubs. These stubs serve as templates, serving as branching targets. When these stubs are resolved by the loader, the target functions become known, enabling calls to dynamic library functions as shown in Table 1.

```
foo@address:
    0x00000000

foo@stub:
    mov x8, [foo@address]
    blx x8

main:
    call foo@stub
```

(a)

```
foo@address:
    0xAABBCCDD

foo@stub:
    mov x8, [foo@address]
    blx x8

main:
    call foo@stub
```

(b)

Table 1: Assembly stubs: (a) Stub before dynamic symbol resolution and, (b) Stub after dynamic symbol resolution. `foo@address` in (a) is uninitialized while in (b) it is given a concrete address.

2.4.4 Bytecode chain

In the original design of Mach-O binaries, the import table was implemented using bytecode chains. These chains embody a basic form of a state machine. This bytecode has a special opcode `DO_BIND` to determine where a state defines a symbol. This approach optimizes storage by specifying only the changes between multiple symbol items.

In this design, the binary typically incorporate four different chains, each serving distinct purposes: Rebase, Non-Lazy, Lazy, and Weak. The Rebase chain is utilized for Position Independent Code (PIC) to recalculate the correct offset of certain values. Non-Lazy, Lazy, and Weak chains are used for dynamic symbol resolution but operate at different stages of the binary execution. Non-Lazy symbols must be resolved during the load time, while Lazy and Weak symbols can be resolved when first called.

Lazy symbols are resolved through an indirect call to the loader, which subsequently reads the bytecode chain to extract a single symbol and writes back the function address. This process is executed via a procedure in `dyld` known as `dyld_stub_binder`, a Non-Lazy symbol and resolves symbols at specific addresses. An overview of this type of resolution is given in Table 2.

```
dyld_stub_binder:
    0x11223344

foo@stub_helper:
    mov x12, foo_bytecode_offset
    mov x8, [dyld_stub_binder]
    blx x8

foo@address:
    foo@stub_helper

foo@stub:
    mov x8, [foo@address]
    blx x8

main:
    call foo@stub
```

(a)

```
dyld_stub_binder:
    0x11223344

foo@stub_helper:
    mov x12, foo_bytecode_offset
    mov x8, [dyld_stub_binder]
    blx x8

foo@address:
    0xAABBCCDD

foo@stub:
    mov x8, [foo@address]
    blx x8

main:
    call foo@stub
```

(b)

Table 2: Lazy dynamic symbols resolution: (a) Stub before lazy dynamic symbol resolution and, (b) Stub after lazy dynamic symbol resolution. `foo@address` in (a) is initialized with a `stub_helper` while in (b) it is given a concrete address. `foo_bytecode_offset` is template for the offset of `foo` symbol in the Lazy bytecode chain.

2.4.5 Fix-ups chain

In later versions of the Mach-O binary format, performance optimization led to the deprecation of bytecode chains in favor of fix-ups chains. Unlike bytecode chains, fix-ups chains do not separate between rebasing and dynamic symbol resolution, instead they are processed together. This approach significantly enhances overall performance by reducing the number of runs through the binary.

In the transition to fix-ups chains, the template values for rebasing and dynamic symbols are encoded as 8-byte values. These encoded values determine how resolutions are carried out during load time. Consequently, all resolutions are completed at load time without resorting to lazy resolution.

2.4.6 Export trie

In Mach-O binaries, dynamic symbols that are meant to be discovered during dynamic symbol resolution are often exported in a structured manner known as an export trie. This data structure resembles a prefix trie and derives its name from this resemblance. The essential characteristic of an export trie is that all item share a common root, necessitating that all symbols are prefixed with an underscore.

2.4.7 Fat binary

A fat binary is a common type of executable binary used in Apple's devices. It functions as a wrapper for a multi-architecture executable, containing a series of Mach-O binaries of a program, each designed for a different architecture. When submitting applications to Apple, a fat binary is typically required. However, when a user downloads the application to a specific device, only the Mach-O binary with the corresponding architecture for that device is actually downloaded and used. This approach ensures compatibility with various Apple devices while optimizing the download size for each specific target.

3 Implementation

In this section, we present an in-depth exploration of our obfuscation technique tailored specifically for Mach-O binaries. Initially, we delineate the comprehensive list of information that needs to be extracted, followed by a detailed explanation of the restoration process during runtime. It's important to note that Objective-C compiled binaries are subject to additional considerations and logic in order to preserve the integrity of the Objective-C runtime environment.

3.1 Design Overview

In our obfuscation approach, the primary objective is to manipulate the load-time data embedded within the binary. By modifying these critical pieces of information, we render the binary incapable of successful loading into memory. These information elements frequently serve as crucial inputs for static analysis tools, such as IDA or Ghidra. Obfuscation of this information creates an environment of partial knowledge, making it harder for analysts to reverse the binary. A similar obfuscation concept was introduced in a previous work for Windows PE binaries [28]. However, it's important to emphasize that our obfuscation method is specifically designed for Mach-O binaries in the Apple ecosystem.

The data required for load-time operations typically comprises directive instructions intended for the loader. These instructions guides the loader in initializing the binary within the system's memory. In our approach, rather than delegating the entirety of this initialization process to the loader, we introduce a sub-control flow intervention between the binary's load-in-memory and its execution. This sub-control protocol is designed to autonomously execute the tasks typically handled by the loader, utilizing the extracted instructions to ensure the binary's proper initialization within the system's memory.

Our obfuscation approach encompasses two distinct designs, both of which execute the same obfuscation procedure. One of these designs relies on an external library, while the other operates independently. Both designs share the common goal of eliminating the information utilized during the load process from the binary format, while simultaneously preserving the ability to load the binary through restoration at runtime.

The initial design of our obfuscation approach incorporates a restoration library, which is integrated into the dynamic loading sequence of the obfuscated binary. Within this library, essential information is extracted and stored. As the load chain progresses, the library is invoked to recover all the extracted information by simulating a segment of the loader's logic, thereby restoring it in its memory representation.

The use of an external library can be omitted. In this particular design, the extracted data is relocated into the `_DATA` section, and the logic for restoring this extracted data to facilitate the completion of the loading process is introduced through code injection. This addition is made as it was not originally included within the binary's inherent logic. Furthermore, the entry pointer is adjusted to point to this injected code, thus ensuring the completion of the restoration process before the binary's main procedure is initiated.

As an optional measure, the extracted data can undergo static encryption, with decryption occurring at load time when our restoration logic is executed. It's worth noting that our restoration logic may rely on functions from the loader, potentially exposing the runtime restoration process. To enhance the resilience of this restoration method, we offer a mechanism to conceal the invocation of these functions through direct address calling.

3.2 Extracting information

Because our obfuscation tampers the load processes, information for load process is considered for extracting information. These information is stored in `_LINK_EDIT` segment. Commands that uses this segments can be removed if they are not necessary. Our obfuscation chose to extract the information in

LC_DYLD_INFO_ONLY, LC_CHAINED_FIXUPS. In addition, we also extract the list of constructor functions, these are often initializers and called before the binary's main procedure.

LC_DYLD_INFO_ONLY load command provides information for the rebasing of bytecode chains, as well as the management of lazy, non-lazy, and weak imported symbols bytecode chains. In this particular scenario, we opt to entirely eliminate the lazy and weak bytecode chains. This is accomplished by configuring the bytecode chain size to a value of 0 within the load command and subsequently overwriting the bytecode chain section within the _LINK_EDIT segment with random values.

LC_CHAINED_FIXUPS load command offers a sequence of runs containing fix-ups. By traversing these runs, we can extract all the imported symbols. These symbols are typically stored as indices pointing to an indexed store of strings, where each index corresponds to the symbol's name and the hosting library. To exclude these symbols from the loading process, we undertake a two-fold process. First, we rewrite the indices for the symbols to reference relocation values. Subsequently, we completely eliminate the string store that holds the symbol names and library information.

Constructor functions refer to pointers that are invoked by the loader once all images have been loaded into memory. These functions are called in a sequential and recursive manner as part of the initialization process. To remove these functions, there are several methods available. One straightforward approach is to modify the LC_SECTION flag to exclude the section from being listed as constructor pointers. Additionally, pointers are typically checked to ensure they reside within the binary's memory region. By adjusting these pointers to point outside of the memory region, the loader will disregard them, effectively achieving the removal of these functions.

3.3 Removing redundant information

In the context of Mach-O binary files, it is imperative to acknowledge the presence of data specifically generated for debugging purposes. This data is automatically generated during the compilation process by the compiler and linker. It is essential to recognize that this information serves no inherent purpose during runtime and, as such, can be removed. Examples of such debugging-related data are defined in commands like LC_SYMTAB, LC_DYSYMTAB, LC_FUNCTION_STARTS, and LC_DATA_IN_CODE, among others. Their exclusion from the binary file does not compromise its functionality during execution but rather streamlines the binary by eliminating superfluous debugging-related content. The complete removal of this information effectively prevents basic analysis that relies on these debug symbols to make sense of the binary program.

LC_SYMTAB and **LC_DYSYMTAB** store a list of symbols in the binary, each entry specifies the name of the symbol and their relative offset in the binary.

LC_FUNCTION_STARTS stores a list of offsets to functions, this information is for debugging purposes only.

LC_DATA_IN_CODE defines a list of ranges within the _TEXT segment of a Mach-O binary. These specified ranges do not contain assembly instructions; rather, they are designated for debugging purposes exclusively.

DWARF debug information is primarily used to define code locations in the event of a crash but are not necessary for the execution of the file. Consequently, it is highly recommended to remove these debug details from the binary to reduce its size and complexity.

Depending on the nature of the load command and its functionality, it may be considered for removal. Load commands that fall under the category of informative or debugging data are typically candidates for removal, provided that their absence does not disrupt the overall load process or while executing.

3.4 Restoration

The restoration of extracted information, which is essential for the full loading of the binary, takes place at runtime. To maintain simplicity in this section, we refrain from discussing the specific storage location of the extracted information, as this topic will be addressed in subsequent sections.

To initiate the process, we register a constructor function that is scheduled to execute prior to the main executable. Utilizing the complete set of parameters passed in by the loader, we uncover a specific address, namely the base address of the main executable loaded in memory, as illustrated in Listing 1.

```
struct ProgramVars {
  void *mh; // mach_header or mach_header64
  int *NXArgcPtr;
  const char ***NXArgvPtr;
  const char ***environPtr;
  const char **__prognamePtr;
};

__attribute__((constructor)) static void
restoration(int argc, const char *const argv[], const char *const envp[],
     const char *const apple[], const struct ProgramVars *vars) {
    const void* main_binary_base = vars->mh;
    // ...
}
```

Listing 1: Using **ProgramVars** struct

Having obtained the base address of the main executable, we can proceed with the restoration process by simulating the loader's actions. For each piece of extracted information, we execute the corresponding restoration procedure in accordance with its specific logic.

`LC_DYLD_INFO_ONLY` Maintaining the `_LINK_EDIT` section as non-writable indeed poses a challenge when it comes to addressing the issue of fixing the Lazy and Weak bytecode chains. However, the bytecode chain describes a state machine, where the terminal state involves the finalization of a symbol. This symbol contains crucial information, including its name, the library that hosts the function, and the address of the symbol stub.

We can systematically traverse the entire bytecode chain. At each terminal state encountered, we possess the necessary details to determine the address of the symbol stub, including the symbol's name and the hosting library. Consequently, we can locate the symbol's address within the hosting library and subsequently overwrite the stub with this address, effectively resolving the issue while keeping the `_LINK_EDIT` section non-writable.

`LC_CHAINED_FIXUPS` Fix-ups are distinct from bytecode chains, and their mixed-up nature makes it impractical to traverse the fix-ups chain directly. However, during the extraction of the fix-ups chain, we capture comprehensive information for each symbol. This information typically includes the address that requires fixing, the symbol's name, and the library in which the symbol is hosted. Armed with this extracted data, we can effectively resolve each symbol to its corresponding function address, thereby restoring the fix-ups without the need to traverse the entire fix-ups chain.

Constructor functions can be readily restored, or even more efficiently, directly invoked. The restoration process involves reinstating the pointers/offsets to their designated locations and allowing the loader to call them during initialization. Alternatively, we can calculate the function addresses and invoke them by utilizing the parameters passed to our constructor. It's worth noting that the loader consistently passes the same arguments to all constructors during initialization, which enables us to call these functions manually without reliance on the loader.

In the process of restoring `LC_DYLD_INFO_ONLY` or `LC_CHAINED_FIXUPS`, it is essential to have a method for locating a symbol's address in another library. This task can be accomplished effectively by employing the `dlsym` function, which enables dynamic symbol resolution, making it possible to find the address of a symbol in the specified library.

It is worth noting that certain fixed locations are designated as read-only. To address this, the corresponding segment must be defined as writeable at its maximum protection value. As part of the restoration process, the page containing the fixed location is modified to be writeable, if it was previously read-only, as demonstrated in Listing 2. This adjustment is essential to facilitate the required modifications during restoration.

3.5 Objective-C compiled binary

The previously described restoration logic is highly effective when applied to binaries compiled from C or C++. However, within the Apple ecosystem, Objective-C is a predominant language for application

```
#include <mach/mach.h>
vm_protect(mach_task_self(), offset, size, 0,
           VM_PROT_READ | VM_PROT_WRITE);
```

Listing 2: Modify the virtual memory range from `offset` to `offset+size` to Read-Write.

development. Objective-C is a unique component of Apple's technology stack and is seamlessly integrated into the loading process of executables through custom passes. Consequently, addressing the challenges associated with Objective-C compiled binaries requires a distinct approach. Before delving into these nuances, it is essential to clarify the synergy between Objective-C and the `dyld` loader.

3.5.1 Relationship with `dyld`

The Objective-C runtime is consistently loaded into memory and automatically mapped to the same virtual memory space as the executable. Within this runtime, a collection of hooks is made available, and these hooks are strategically employed by the `dyld` at various stages of the binary loading and unloading processes. During the initialization of the Objective-C runtime via `libSystem`, an array of callbacks is supplied to `dyld`. The establishment of these hooks is illustrated in Listing 3.

Via the callback structure `_dyld_objc_callbacks_v1`, the Objective-C runtime registers three functions at distinct stages of the binary loading and unloading processes: when the binary is mapped into memory, when the binary is invoked to call constructors, and when the binary is subsequently unmapped from memory.

3.5.2 Objective-C data in binary

Objective-C compiled binaries include sections identified by the prefix `_objc`. These sections are integral to the functioning of the Objective-C runtime, facilitating the initialization of Objective-C classes and selectors. In a succinct overview, the Objective-C runtime carries out the initialization of class objects and selectors when the binary is mapped into memory, Objective-C `+load` methods are called during constructors invocation.

Now, we delve into the components that are influenced when extracting information, as described in Section 3.2. Objective-C employs a specific optimization technique designed to enhance performance. In this approach, class prototypes are directly stored within the binary. This differs from other Objective Oriented languages, where class prototypes are allocated alongside object creation using `vtable`. The advantage of this approach is that it enables Objective-C to efficiently determine the class of an object. This is achieved by directing an `isa` ("is a") pointer to the corresponding class prototype, statically stored within the binary file. It's important to note that within this prototype structure, there exists a parent class pointer. This parent class pointer is also represented by an `isa` pointer, which can point either to another prototype within the binary or to a prototype originating from a library. The parent pointer for classes cannot be 0 (NULL). Classes that do not have an explicitly defined parent are automatically inherited from the `NSObject` class.

Objective-C exports class prototypes as a part of the symbols list, ensuring their inclusion in the export trie. When a parent pointer references a class in another library, the resolution process for fixing these pointers to point to the corresponding class prototypes is managed through mechanisms of bytecode chain (`LC_DYLD_INFO_ONLY`) or fix-ups chain (`LC_CHAINED_FIXUPS`).

All these class prototypes are consolidated into a list within a section named `__objc_classlist`. When the binary is mapped into memory, the Objective-C runtime invokes the binary mapping hook, initiating the process of reading and adding these classes to the internal cache.

Similarly, `+load` methods are stored within the `__objc_init_func` section. The Objective-C runtime systematically traverses these functions alongside their corresponding class prototypes in the cache.

3.5.3 Fixing the restoration logic

Our current restoration logic is executed after Objective-C procedures have already run, potentially resulting in crashes due to incorrectly set pointers in certain class prototypes. Figure 1 illustrates various workflow versions to help readers understand why the current workflow is problematic. In Figure 1(a), the workflow depicts Objective-C runtime with dyld only. In Figure 1(b), our current restoration workflow

```
// dyld
struct _dyld_objc_callbacks_v1
{
    uintptr_t                       version; // == 1
    _dyld_objc_notify_mapped        mapped;
    _dyld_objc_notify_init          init;
    _dyld_objc_notify_unmapped      unmapped;
    _dyld_objc_notify_patch_class   patches;
};

void setObjCNotifiers(_dyld_objc_notify_mapped mapped,
                      _dyld_objc_notify_init init,
                      _dyld_objc_notify_unmapped unmapped,
                      _dyld_objc_notify_patch_class patchClass,
                      _dyld_objc_notify_mapped2 mapped2);

void APIs::_dyld_objc_register_callbacks(const _dyld_objc_callbacks* callbacks) {
    // ...
    if ( callbacks->version == 1 ) {
        const _dyld_objc_callbacks_v1* v1 = (const _dyld_objc_callbacks_v1*)callbacks;
        setObjCNotifiers(v1->mapped, v1->init, v1->unmapped, v1->patches, nullptr);
    }
    // ...
}

// Objc4
void _objc_init(void) {
    // ...
    _dyld_objc_callbacks_v1 callbacks = {
        1, // version
        &map_images,
        load_images,
        unmap_image,
        _objc_patch_root_of_class
    };
    _dyld_objc_register_callbacks((_dyld_objc_callbacks*)&callbacks);
    // ...
}
```

Listing 3: Objc4 and `dyld` hooks

is combined with Objective-C runtime and `dyld`. Figure 1(c) illustrates the desired workflow to prevent crashes during binary loading. To align with the correct workflow and avoid these issues, we need to make additional modifications to the binary.

1. Rename `__objc_classlist` section to something else.
2. Rename `__objc_init_func` section to something else.
3. Add a custom shellcode as described in Listing 4 before `_TEXT` segment.
4. Modify the entry point to the custom shellcode.

Since our restoration logic cannot intervene with the Objective-C runtime, we have opted to modify the section names within the binary. By doing so, we ensure that the runtime proceeds without executing any operations prematurely and also keep the pointer references to the section's data. Instead, we execute it at the appropriate moment within our logic. At the end of our previous restoration logic, we simulated how the Objective-C runtime utilizes the data within the `__objc_classlist` section. Our enhanced restoration logic introduces a new method named `restore_objc`. This method mimics the initialization procedure of the Objective-C runtime for the `__objc_init_func` section. During runtime, after all restoration tasks have been completed, the `+load` methods of all Objective-C foundation libraries are

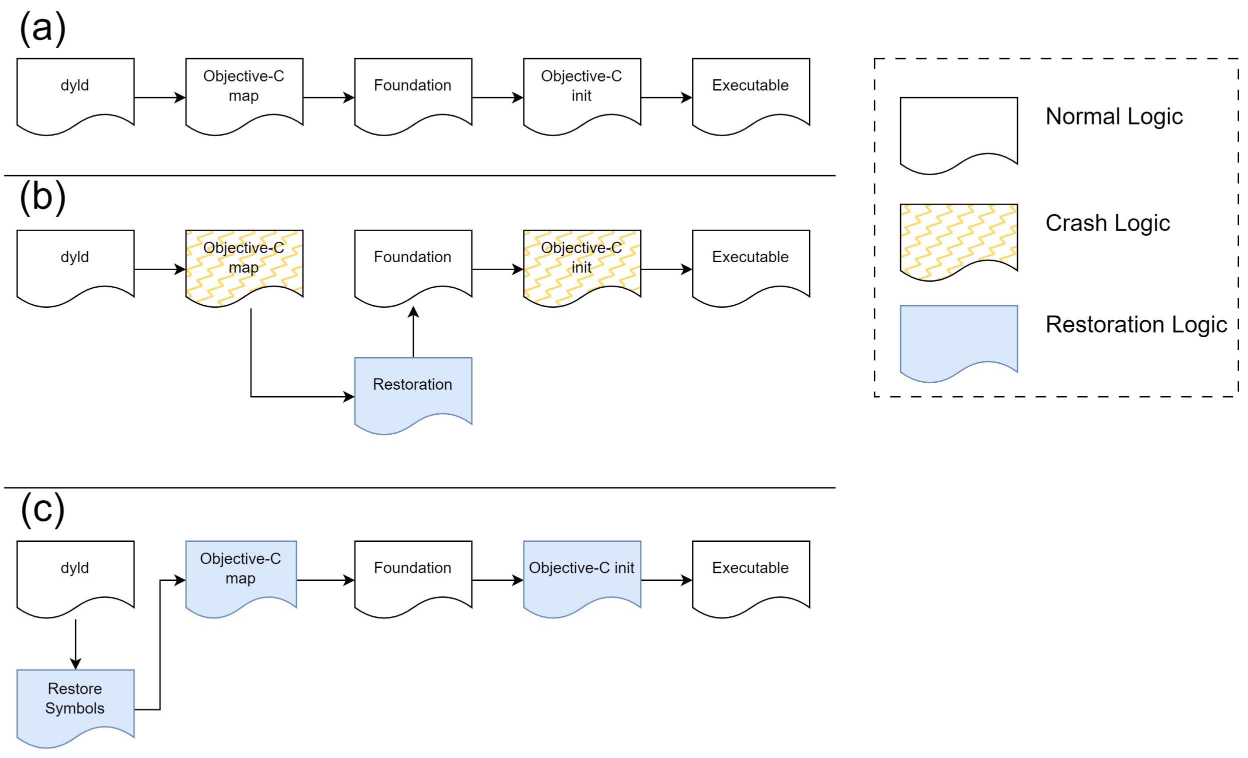

Figure 1: Different Workflows involving `dyld`, Objective-C Runtime, Foundation, and Restoration: (a) Without restoration workflow, (b) Restoration process without considering Objective-C, and (c) Restoration process compatible with Objective-C.

invoked. After that, `dyld` invoke the shellcode that has been inserted. The shellcode's role is to locate and call the `restore_objc` function to execute the `+load` methods within the binary. Subsequently, the shellcode jumps to the correct main method of the binary, initiating the execution of the binary itself. This orchestration ensures that the binary's initialization proceeds smoothly and in a manner consistent with Objective-C runtime requirements.

Objective-C logic for processing the class prototypes are done through non-public methods like `readClass`, `realizeClassWithoutSwift`, `remapClass`, to name a few. These symbols cannot be found in the export trie. However, they are available in the `LC_SYMTAB` directives. We can search for these symbols addresses and rebuild the logic as described in `_read_images` protocol of Objective-C runtime.

The process of locating the `restore_objc` procedure can be intricate. To keep the shellcode as compact as possible, we have opted to store the procedure's address in a location that can be easily calculated. Specifically, we have chosen the end of the `_DATA` segment for this purpose. At this location, the first pointer value immediately following the end of the segment represents the address of the `restore_objc` procedure, while the second pointer value indicates the address of the binary's main function. These pointer values are written after the restoration logic.

In theory, the shellcode could be more complex, but it must adhere to the fundamental requirement of executing the `+load` methods of the binary as a simulation of the Objective-C runtime's initialization procedure.

In practical scenarios, the available space for the shellcode before the `_TEXT` segment and the number of pointer values after the `_DATA` segment may be limited. Hence, it is crucial to keep the shellcode as concise as possible and minimize the number of required pointer values. It is reasonable to assume that space is generally available since these segments are page-aligned, unless the code and data sizes are exact multiples of the page size, which would leave no extra space.

```
adr x8, 0
# x9 = (offset end of __DATA) - (offset shellcode)
movz x9, #0x9999
add x8, x8, x9

# store link register and offset end __DATA
stp x30, x8, [sp], #-0x10

# store passed arguments to main
stp x3, x2, [sp], #-0x10
stp x1, x0, [sp], #-0x10

# restore objc
ldr x9, [x8]
blr x9

# fetch main arguments
ldp x1, x0, [sp, #0x10]!
ldp x3, x2, [sp, #0x10]!
# fetch link register and offset end __DATA
ldp x30, x8, [sp, #0x10]!

ldr x9, [x8, #8]
# jump to binary's main
br x9
# main returns directly to dyld because link register is set correctly
```

Listing 4: Shellcode inserted (in ARM64)

In Figure 2, we provide a visual representation of the control flow that occurs when we implement the aforementioned modifications to ensure our restoration logic functions seamlessly with the Objective-C runtime. This diagram helps illustrate the step-by-step execution of the modified process.

3.6 Concealing the restoration logic

Up to this point, our discussion has centered on the overarching implementation of the obfuscation logic. Nonetheless, it's worth noting that the restoration flow involves numerous calls to library functions, which can be inspected relatively easily. In this section, we delve into strategies for effectively concealing this logic by rendering these library function calls imperceptible.

We already possess a fundamental understanding of how the loader resolves symbols into function addresses using `LC_DYLD_INFO_ONLY` or `LC_CHAINED_FIXUPS`. However, it is worthwhile to delve into greater detail on the precise mechanism by which the loader locates these function addresses. In essence, the loader maintains a list of loaded libraries. For each loaded library, it utilizes the export trie data structure to locate public symbols by their symbol names. Consequently, if we are provided with the base address of a library and a function identifier, we should be able to navigate through the export trie to pinpoint the function's address in memory. This elucidates the fundamental concept of using the export trie for symbol resolution. However, it's important to acknowledge that at times, a function may be re-exported from another library. In such scenarios, a recursive search through libraries is necessary to ascertain the function's address. This recursive search ensures that the loader can accurately resolve symbols even when they are re-exported from different libraries.

During the process of locating symbols, it's important to take into account that symbols may store their hosting library using relative paths. These relative paths can be expressed either as directory-relative paths or through path variables such as `@rpath`, `@executable_path`, or `@loader_path`. To ensure accurate resolution of these relative paths, it is advisable to convert them into their corresponding full paths.

To access the list of loaded libraries in memory, a sequential invocation of three symbols, namely `_dyld_image_count`, `_dyld_get_image_header`, and `_dyld_get_image_name`, can be employed. These three symbols collectively offer a means to compile a comprehensive list of loaded libraries as can be

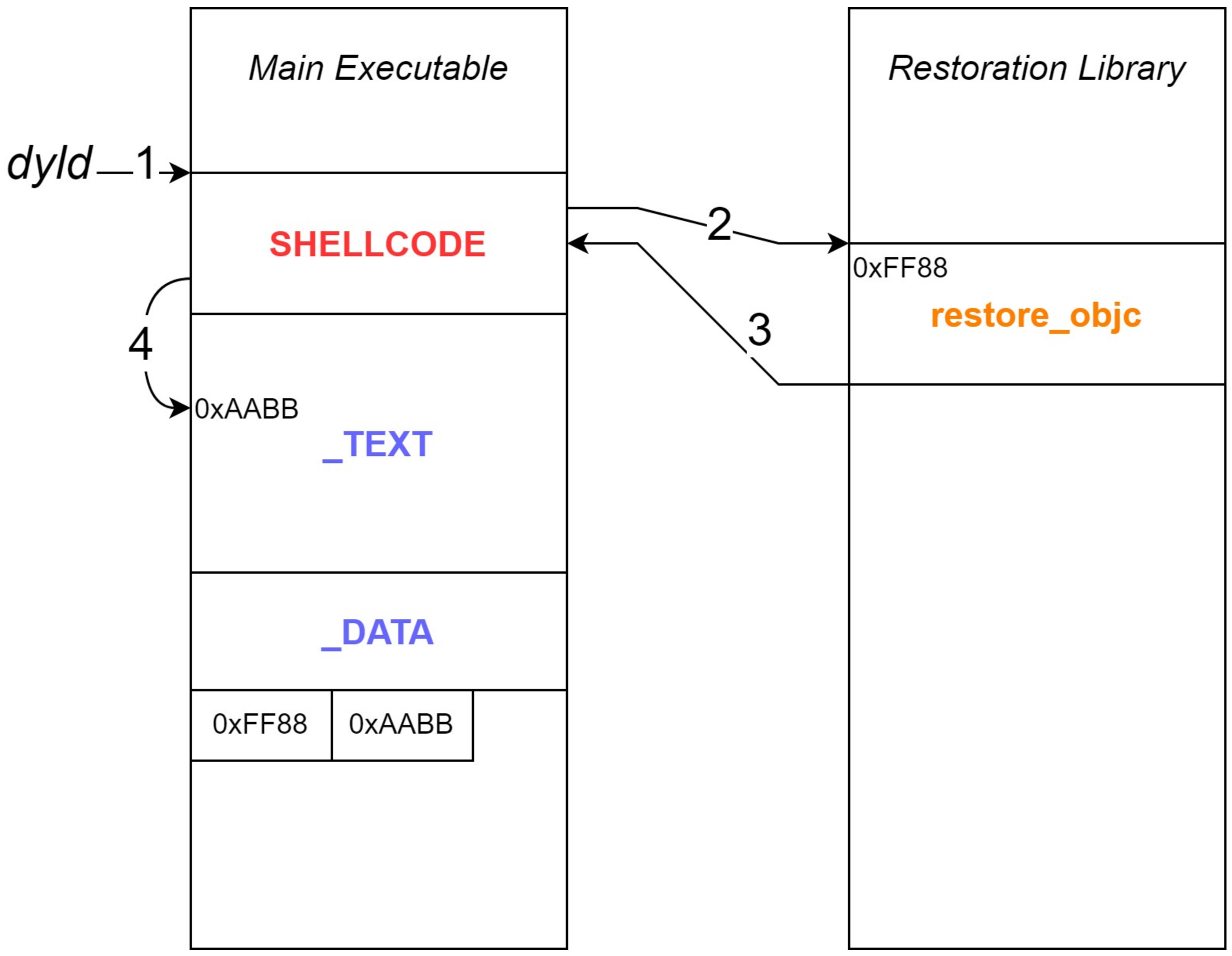

Figure 2: Shellcode control flow with dyld and main executable.

seen in Listing 5. Specifically, `_dyld_get_image_header` provides the base address of the library at a specific index, while `_dyld_get_image_name` returns the full path of the library.

```
#import <mach-o/dyld.h>
uint32_t count = _dyld_image_count();
for(uint32_t i = 0; i < count; i++) {
    const char *name = _dyld_get_image_name(i);
    const void *header = _dyld_get_image_header(i);
}
```

Listing 5: Get a list of loaded libraries

Certainly, these three functions can also be concealed from our logic. As previously explained, one merely requires the library's base address and the symbol's name to obtain the function address. Consequently, if we can locate the base address of the loader, we can access any function available in the export trie without relying on these three functions explicitly.

We outline a fundamental logic for determining the base address of any library loaded in memory. Leveraging the loader's mechanism, the entire binary is loaded into memory as a unified entity, devoid of segmentation. Given this memory layout, it is observed that the binary's header consistently resides at a lower memory address than any function within the binary. Furthermore, owing to memory optimization practices, the binary is typically mapped onto an address aligned with memory pages. Exploiting these characteristics, we formulate a straightforward search algorithm that involves iteratively traversing each memory page while scanning for the Mach-O magic value. The process, named `find_header`, is illustrated in Listing 6.

```c
const uint32_t magic64 = 0xfeedfacf;
const uint32_t magic32 = 0xfeedface;

void *find_header(void *_func) {
  const uint64_t page_size = 0x1000;
  uint64_t func = (uint64_t)_func;
  uint64_t start_searching = func + (0x1000 - (func % page_size));
  uint32_t *x = (uint32_t *)(start_searching);
  while (*x != magic64 && *x != magic32) {
    x -= 0x1000 / 4;
  }
  return (void *)x;
}
```

Listing 6: Searching for Mach-O base address

Using the `find_header` procedure, we can locate the base addresses of the libraries loaded in memory. Applying this approach, we can find the base addresses of three functions: `_dyld_image_count`, `_dyld_get_image_header`, and `_dyld_get_image_name`. It's important to note that we only need one function in `dyld` to discover its base address. To maintain a lower profile and make our logic less conspicuous, we typically choose a relatively inconspicuous function, such as `dyld_get_sdk_version`. This choice of function minimizes any insights it might offer to reverse engineers analyzing our logic.

At this stage, our logic can be obscured by manually determining the addresses of functions within libraries. This allows us to execute all function calls through intermediate values, making our logic less discernible to potential reverse engineering efforts.

Now, let's discuss where the extracted information can be stored. There are several approaches to consider:

1. **Inside the Restoration Library:** The simplest approach is to include the extracted information as part of the data within the restoration library. This keeps everything self-contained within the library, but requires a re-compilation of the restoration library for each obfuscated binary.
2. **Separated File:** You can store the information in a separate file and load it at runtime using an open file procedure. This approach provides flexibility and makes it easier to manage the data independently.
3. **__DATA Segment of the Obfuscated Binary:** Storing the information within the `_DATA` segment of the obfuscated binary itself is more complex. It may require resizing the segment, and if the segment becomes larger than originally planned, it can lead to complicated relocation processes, potentially causing issues with pointer references.
4. **Another Library:** You can create another library specifically for storing the information, containing only a `_DATA` segment. During runtime, the restoration library can access this separate library to retrieve the information needed for restoration. The Mach-O loader's permits the loading of binary files that adhere to the Mach-O format. This flexibility opens doors to the creation of custom Mach-O binaries designed for storing and loading extracted information.
5. **Internet Storage:** For more dynamic retrieval, you can consider storing the information on the internet and fetching it at runtime. This approach provides the flexibility to update the information remotely, but it also introduces dependencies on external servers and network availability, as well as high latency.

The choice of where to store the information depends on your specific requirements, complexity considerations, and performance needs. Each method has its pros and cons, and the decision should align with the goals of your obfuscation technique.

As a best practice, it is advisable to encrypt this information, decrypting it only when necessary for restoration. This additional layer of obfuscation ensures that the data remains inaccessible to human inspection, enhancing overall security and protection.

3.7 Extension

We have outlined the fundamental aspects of our method, but it's essential to note that our approach can be further enhanced to achieve more advanced levels of obfuscation. These enhancements require meticulous work and substantial effort to implement fully. We are eager to delve into these potential upgrades to provide a deeper understanding of the significant obfuscation capabilities offered by our method.

3.7.1 Obfuscate restoration library

Our obfuscation method relies on an external library, initially designed solely for restoration purposes. However, to bolster security, this library should now encompass additional runtime protections, such as detecting Jailbroken devices or identifying Frida hooking attempts. These procedures, when implemented with system APIs, are exposed to analysts. Fortunately, our library can also obfuscate itself using our obfuscation method. As long as the extracted data (of itself) is accessible during runtime, our library can first restore itself and then proceed with the restoration of the targeted obfuscated binary. It can also be obfuscated using other obfuscation techniques to obscure the underlying logic used for decrypting the extracted data and the restoration workflow. This added layers of obfuscation enhances the resilience of our restoration library.

3.7.2 Obfuscate multiple binaries

While we have primarily discussed the obfuscation of a *single* binary, it's essential to recognize that *multiple* binaries can undergo obfuscation using the same process. The key distinction is that all obfuscated binaries must be resolved at runtime. Typically, we aim to obfuscate the main executable and its required libraries.

Expanding on this, as we outlined in our method, we rely on the export trie mapped in memory for each dynamic symbol. What if we also remove the export trie for the libraries we obfuscate? By extracting the export trie data from these libraries, they no longer contain any public information about their functions. However, our restoration library can still perform the restoration process because it now possesses knowledge of all the libraries' export tries.

The approach of fully obfuscating all binaries within a package is an extreme implementation of our method. While it significantly enhances security by creating a substantial challenge for reverse engineers trying to understand the application, it comes with a trade-off in terms of startup time. Apple tends to reject applications that take too long to start up, and users can become frustrated if the app has a slow startup time.

3.7.3 A method for function hooking

At the core of our obfuscation method, which focuses on restoring dynamic symbols, lies a technique that enables us to directly modify these symbols. This capability opens the door to function replacement, commonly known as function hooking. Function hooks are frequently employed to intercept the invocation of a function, serving various purposes, including security enhancements and logging.

To override a library's functions using function hooks, we start by creating our customized version of the function ensuring that the parameters, and the return type matches the original function. Due to the deterministic nature of compilers, especially with architecture-dependent compilation, we can be confident that the registers used for passing arguments and return values are set correctly. Then, at runtime, we locate the placeholder for the dynamic symbol and write the address of our hook function to effectively replace the original function with our custom implementation.

Function hooking is a prominent topic in the field of security, and it has evolved with tools like Frida [47] and Fishhook [16]. Frida allows developers to inject hooks at arbitrary addresses, Fishhook allows developers to replace the body of a function at runtime after some setup procedures. However, the hooking method we've described here is distinct from Frida's approach, rather it's quite similar to Fishhook but on a binary level. While we don't have the flexibility to hook arbitrary addresses, our method enables us to modify function invocations, directing them to our custom functions through dynamic symbol resolution.

This approach is particularly valuable when we need to intercept and modify system API calls, such as file opening (e.g., `fopen`), or when we want to disable certain functions from being called altogether.

4 Conclusion

Throughout this paper, we have presented a comprehensive overview of obfuscation techniques for Apple device applications. We introduced our innovative approach to obfuscating these applications by directly manipulating the Mach-O binary, focusing on dynamic symbols that are resolved by the loader. The removal of these symbols poses a significant challenge for static analysis, as it eliminates visible dynamic library invocations. Moreover, these symbols are dynamically resolved at runtime without the need to rebuild the symbol table, making them resilient even in the face of memory extraction attempts to reconstruct the binary in memory. We have demonstrated the applicability of this obfuscation technique to Objective-C compiled applications, a common choice for software running on Apple's devices. While the assembly instructions remain unchanged and may be comprehensible to experienced reverse engineers, the removal of dynamic symbols significantly hinders the analysis process. Overall, our approach offers a promising strategy for enhancing the protection of Mach-O binaries against reverse engineering attacks in the Apple ecosystem.

References

[1] National Security Agency. Ghidra. URL https://ghidra-sre.org/.

[2] Sergi Alvarez. Radare2. URL https://rada.re/.

[3] Apple. dyld, . URL https://opensource.apple.com/source/dyld/.

[4] Apple. LibSystem, . URL https://opensource.apple.com/source/Libsystem/.

[5] Apple. Objective-C Runtime, . URL https://opensource.apple.com/source/objc4/.

[6] Genevieve Arboit. A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, pages 102–110. Citeseer, 2002.

[7] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 189–200, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347716. doi:10.1145/2991079.2991114. URL https://doi.org/10.1145/2991079.2991114.

[8] Kamil Borzym. MachObfuscator. URL https://github.com/kam800/MachObfuscator/.

[9] Vincent Bénony. Hopper. URL https://www.hopperapp.com/.

[10] Jan Cappaert and Bart Preneel. A general model for hiding control flow. In *Proceedings of the tenth annual ACM workshop on Digital rights management*, pages 35–42, 2010.

[11] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, 1998.

[12] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4. doi:10.1145/2420950.2420997. URL http://doi.acm.org/10.1145/2420950.2420997.

[13] Rizin Community. Cutter, . URL https://cutter.re/.

[14] Rizin Community. Rizin, . URL https://rizin.re/.

[15] Stephen Drape. Intellectual property protection using obfuscation. 2010.

[16] Facebook. fishhook. URL https://github.com/facebook/fishhook/.

[17] Roberto Fellin and Mariano Ceccato. Experimental assessment of xor-masking data obfuscation based on k-clique opaque constants. *Journal of Systems and Software*, 162:110492, 2020.

[18] Roberto Fellin and Mariano Ceccato. Experimental assessment of xor-masking data obfuscation based on k-clique opaque constants. *Journal of Systems and Software*, 162:110492, 2020.

[19] Weijie Feng, Binbin Liu, Dongpeng Xu, Qilong Zheng, and Yun Xu. Neureduce: Reducing mixed boolean-arithmetic expressions by recurrent neural network. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 635–644, 2020.

[20] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions. In *GreHack 2016*, 2016.

[21] Hex-Rays. IDA. URL https://hex-rays.com/ida-pro/.

[22] Chengyang Li Tianbo Huang, Xiarun Chen, Chenglin Xie, and Weiping Wen. iOLLVM: Enhanced version of OLLVM. In *Artificial Intelligence Trends & Technologies*. Academy and Industry Research Collaboration Center (AIRCC), feb 2022. doi:10.5121/csit.2022.120409. URL https://doi.org/10.5121%2Fcsit.2022.120409.

[23] Vector 35 Inc. Binary Ninja. URL https://binary.ninja/.

[24] iPhoneDev. dyld shared cache. URL https://iphonedev.wiki/Dyld_shared_cache/.

[25] Björn Johansson, Patrik Lantz, and Michael Liljenstam. Lightweight dispatcher constructions for control flow flattening. In *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*, pages 1–12, 2017.

[26] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm – software protection for the masses. pages 3–9, 05 2015. doi:10.1109/SPRO.2015.10.

[27] Seoyeon Kang, Sujeong Lee, Yumin Kim, Seong-Kyun Mok, and Eun-Sun Cho. Obfus: An obfuscation tool for software copyright and vulnerability protection. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, pages 309–311, 2021.

[28] Yuhei Kawakoya, Eitaro Shioji, Yuto Otsuki, Makoto Iwamura, and Takeshi Yada. Stealth loader: Trace-free program loading for api obfuscation. In *Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings*, pages 217–237. Springer, 2017.

[29] Pengwei Lan, Pei Wang, Shuai Wang, and Dinghao Wu. Lambda obfuscation. In *Security and Privacy in Communication Networks*, 2017. URL https://api.semanticscholar.org/CorpusID:19169928.

[30] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30 (1):3–19, 2009.

[31] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[32] Byoungyoung Lee, Yuna Kim, and Jong Kim. binob+ a framework for potent and stealthy binary obfuscation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 271–281, 2010.

[33] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.

[34] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. {MBA-Blast}: Unveiling and simplifying mixed {Boolean-Arithmetic} obfuscation. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1701–1718, 2021.

[35] Binbin Liu, Qilong Zheng, Jing Li, and Dongpeng Xu. An in-place simplification on mixed boolean-arithmetic expressions. *Security and Communication Networks*, 2022, 2022.

[36] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[37] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-third annual computer security applications conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[38] Dongliang Mu, Jia Guo, Wenbiao Ding, Zhilong Wang, Bing Mao, and Lei Shi. Ropob: obfuscating binary code via return oriented programming. In *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22–25, 2017, Proceedings 13*, pages 721–737. Springer, 2018.

[39] Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[40] Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. *Electronic Commerce Research*, 6(2):155–171, 2006.

[41] Markus FXJ Oberhumer. Upx the ultimate packer for executables. *http://upx. sourceforge. net/*, 2004.

[42] Oreans. Themida. URL https://www.oreans.com/Themida.php/.

[43] JJ O'Connor and EF Robertson. Lothar collatz. *St Andrews University School of Mathematics and Statistics, Scotland*, 2006.

[44] Jens Palsberg, Sowmya Krishnaswamy, Minseok Kwon, Di Ma, Qiuyun Shao, and Yi Zhang. Experience with software watermarking. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 308–316. IEEE, 2000.

[45] Polidea. ios-class-guard. URL https://github.com/Polidea/ios-class-guard/.

[46] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, pages 275–290, 2007.

[47] Ole André V. Ravnås. Frida. URL https://frida.re/.

[48] B. Reichenwallner and P. Meerwald-Stadler. Simplification of general mixed boolean-arithmetic expressions: Gamba. In *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 427–438, Los Alamitos, CA, USA, jul 2023. IEEE Computer Society. doi:10.1109/EuroSPW59978.2023.00053. URL https://doi.ieeecomputersociety.org/10.1109/EuroSPW59978.2023.00053.

[49] Benjamin Reichenwallner and Peter Meerwald-Stadler. Efficient deobfuscation of linear mixed boolean-arithmetic expressions. In *Proceedings of the 2022 ACM Workshop on Research on offensive and defensive techniques in the context of Man At The End (MATE) attacks*, pages 19–28, 2022.

[50] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3055–3073, 2022.

[51] PNF Software. JEB, . URL https://www.pnfsoftware.com/.

[52] VMProtect Software. VMProtect, . URL https://vmpsoft.com/vmprotect/overview/.

[53] Romain Thomas. O-MVLL. URL https://obfuscator.re/omvll/.

[54] Romain Thomas. Lief-library to instrument executable formats. *Retrieved February*, 22:2022, 2017.

[55] Romain Thomas. The Poor Man's Obfuscator, 2022. URL https://www.romainthomas.fr/publication/22-pst-the-poor-mans-obfuscator/.

[56] Roberto Tiella and Mariano Ceccato. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 182–192. IEEE, 2017.

[57] Yan Wang, Shuai Wang, Pei Wang, and Dinghao Wu. *Turing Obfuscation*, pages 225–244. 01 2018. ISBN 978-3-319-78812-8. doi:10.1007/978-3-319-78813-5_12.

[58] Dongpeng Xu. Opaque predicate: Attack and defense in obfuscated binary code. 2018.

[59] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015.

[60] Jiajia Yi, Lirong Chen, Haitao Zhang, Yun Li, and Huanyu Zhao. A security model and implementation of embedded software based on code obfuscation. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1606–1613. IEEE, 2020.

[61] Peihua Zhang, Chenggang Wu, Mingfan Peng, Kai Zeng, Ding Yu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. Khaos: The impact of inter-procedural code obfuscation on binary diffing techniques. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, pages 55–67, 2023.

[62] Yongxin Zhou and Alec Main. Diversity via code transformations: A solution for ngna renewable security. *NCTA-The National Show*, 2006.

[63] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *International Workshop on Information Security Applications*, pages 61–75. Springer, 2007.