Simulating Loader for Mach-O Binary Obfuscation and Hooking

Anh Khoa Nguyen
khoana@verichains.io
Verichains

Thien Nhan Nguyen
Verichains

Abstract

Mach-O binary is prevalent in Apple devices; despite its widespread use, studies on Mach-O modifications are limited and usually with minimal changes here and there. We have observed that in PE and ELF binaries, when enhanced modifications are made, they can be utilized for complex techniques such as obfuscation or hooking. This has prompted us to explore the Mach-O binary format in order to develop innovative modification techniques.

In this paper, we introduce a novel method for modifying Mach-O by simulation of the loader. Rather than allowing the loader to perform all necessary initialization, our method delegates this task to our simulated loader. This method has led us to develop two new techniques that can be applied to the obfuscation and hooking of Mach-O binaries. Earlier attempts at obfuscating Mach-O binaries merely edit trivial data which can be reconstructed, whereas our technique completely removes all dynamic symbols that make calls to libraries, including references to imported Objective-C classes, leaving them undefined. Our hooking technique has demonstrated its ability not only to hijack dynamic symbols, but also Objective-C class methods, whether they are contained within the binary or imported.

1 Introduction

Binary modification has emerged as a salient subject in the domains of Windows and Linux, with multiple obfuscation and hooking methodologies being developed for PE and ELF binaries. Despite Apple's prominence as a leading Operating System, boasting its unique Mach-O binary format, the scholarly exploration of binary modification within this format remains conspicuously sparse. Specifically, a mere two obfuscation schemes have been designed for Mach-O binaries, both of which are notably rudimentary. In contrast, the PE and ELF binaries are equipped with an array of obfuscation schemes. Intriguingly, the structure of Mach-O bears a striking resemblance to that of ELF, which raises questions as to why more research has not been directed towards the Mach-O binary. In parallel, hooking techniques have been extensively studied for PE and ELF binaries, with popular methods including IAT hooking and PLT/GOT hooking. However, the potential applicability of these techniques to Mach-O remains undocumented.

Our approach to the topic of Mach-O binaries is centered on understanding the binary format and the loading procedure performed by the loader `dyld` and the Objective-C runtime `objc4`. The loader employs a large array of metadata stored in the `_LINK_EDIT`, which we believe can be eliminated and processed using a simulated loader. Following this line of thinking, we have constructed a simulated loader that is capable of invoking constructors, initializing nonlazy Objective-C classes, and symbolizing dynamic symbols, encompassing both pure C symbols and imported Objective-C class references.

The use of a simulated loader enables the creation of an advanced use case for obfuscation and hooking. The obfuscation scheme is completed when additional data is extracted from the binary. The resulting binary has undefined external symbols as they have been relocated and stored in an encrypted manner within our simulated loader. The simulated loader can also function as a hooking tool by redirecting external symbols to a tailored hooked variant of the symbol.

In this paper, we illustrate the implementation of a simulated loader for the Mach-O binary, along with the obfuscation strategy and the hooking method developed. The methods we will discuss are adaptable to any compiled binaries, particularly those based on Objective-C, the prevalent language runtime on Apple 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. Before we outline our methodology, we introduce some previous techniques for Mach-O obfuscation in Section 3. In Section 4, we provide a detailed implementation to implement a simulation of the Mach-O loader as a sepa-

rated dynamic library. 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. After that, we detail the obfuscation scheme in Section 5 and the hooking method in Section 6.2, both based on the simulated loader strategy. Section 7 is dedicated to the evaluation of our obfuscation method, conducted on GNU Coreutils. This section also discusses limitations and how strong the obfuscation scheme based on this is. Section 8 discusses potential future works that can be extended. Finally, in Section 9, we offer our concluding remarks.

2 Background

2.1 Apple's loader `dyld`

The Apple loader [1], known as `dyld`, is responsible for the execution of programs, including the loading of the binary and its associated libraries, the resolution of dynamic symbols, the rebasing of offsets, and the final execution of the binary.
Due to the shared cache mechanism [7] introduced in iOS 13.5 or macOS 11.0, important libraries, including the system (often referred to as `libSystem` [2]), C++, Objective-C Runtime [3], 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 the system boots.

2.2 Mach-O binary format

The Mach-O binary format is inherently complex. To gain a comprehensive understanding of the techniques described in this paper, 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 on the obfuscation of vital information stored within the binary file. Therefore, a thorough understanding of how the Mach-O binary format stores this information is important in understanding our approach.

2.2.1 Basic Mach-O structure

The Mach-O binary format can be comprehensively examined from multiple perspectives. One fundamental approach is to dissect it on the basis of its encoding of binary data. In this sense, 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.

2.2.2 Dynamic library load chain

In most cases, programs cannot function as standalone entities but instead rely on dynamic libraries. These libraries are registered in the header of Mach-O binaries using commands such as `LC_DYLIB` (or similar equivalents). These commands establish a load chain, organized in a specific order. The loader dynamically loads libraries that are registered in the load chain.
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 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 file on the disk.

2.2.3 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 fixups chains.
During load time, the loader of a Mach-O binary reads the import table, searches 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 and serve as branching targets. When these stubs are resolved by the

loader, the target functions become known, allowing calls to dynamic library functions as shown in Table 1.

2.2.4 Rebase

In binary files, references (pointers) to other data are often stored as file offsets. During execution, when the binary is loaded into memory at a specific address range, these references need to be adjusted from relative (offset in the file) to absolute addresses. This process is called rebasing, where pointers are rebased from 0 to the loaded address. Readers might be familiar with Position-Independent Code, and the rebase is the design for this mechanism in Mach-O binaries. While there is no specific term for the list of pointers to be rebased at runtime, for the sake of brevity, we can refer to these as the "rebase table".

2.2.5 Bytecode chain

In the original design of Mach-O binaries, the import table and the rebase table were implemented using bytecode chains. These chains embody the basic form of a state machine instruction. This bytecode has a special opcode `BIND_OPCODE_DO_BIND` to determine where a state defines a symbol, or opcodes with prefix `REBASE_OPCODE_DO_REBASE` to define a rebase pointer. This approach optimizes storage by specifying only changes between multiple items.
In this design, there are four different chains, each serving distinct purposes: Rebase, Non-Lazy, Lazy, and Weak. The Rebase chain is the rebase table. 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 symbols can be resolved when first called. Weak symbols are used to avoid collision in the symbol name.
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`. An overview of this type of resolution is given in Table 2.

2.2.6 Fixups chains

In later versions of the Mach-O binary format, after iOS 15 or macOS 14, performance optimization led to the deprecation of bytecode chains in favor of fixups chains. Unlike bytecode chains, fixups chains do not separate between rebasing and dynamic symbol resolution; instead, they are processed together. This approach significantly improves overall performance by reducing the number of runs through the binary.
In this design, there exist sequences of contiguous 8-byte values. Each 8-byte unit incorporates a single bit to signify whether it is intended to serve as a rebase pointer or to represent a dynamic symbol. For rebase pointers, the unused bits are repurposed to specify the readdressing mechanism, while dynamic symbols utilize the remaining bits to encode both the index within the library list and the index within the string table corresponding to the symbol name. The 8-byte values are modified in place when rebased or resolved during load time.
Due to the deprecation of 32-bit hardware by Apple, each value in fixups chains is 8 bytes, which is the same size as a pointer in a 64-bit system.

2.2.7 Export trie

In Mach-O binaries, dynamic symbols that are meant to be discovered during dynamic symbol resolution are stored in 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 items share a common root, which requires that all symbols be prefixed with an underscore.

2.2.8 Fat binary

A fat binary is a common type of executable binary used in Apple devices. It functions as a wrapper for a multiarchitecture executable containing different architectures of Mach-O binaries of the same program. 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 Related Works

In this section, we survey the existing open-source solutions for Mach-O binary obfuscation, focusing specifically on methods that take a binary as input and produce an obfuscated output. We exclude approaches that involve decompilation followed by obfuscation on the decompiled code from our consideration; packers are also excluded because it is a different kind of obfuscation. This enumeration aims to provide an overview of the current landscape of Mach-O binary obfuscation techniques that adhere to the specified criteria.
We first go through a list of common obfuscation techniques on Mach-O binaries; these techniques are well known and often suggested, as well as supported, by many. These techniques often involve the removal of exported symbols. The exported symbols are not mandatory in the main executable because the binary entry point is accessible through the `LC_MAIN` load command and other symbols are not referenced by other libraries. Removing the list of exported symbols (and sometimes public symbols) can be easily performed through com-

piler / linker argument invocation or directly remove the associated load commands, LC_SYMTAB for instance.

Unused sections and data within the binary are also targeted for removal in certain obfuscation methods. This process is relatively straightforward, as these portions of the program are deemed unnecessary for execution.

Numerous obfuscation methods focus on renaming Objective-C symbols, as evident in tools like MachObfuscator [4] and ios-class-guard [14]. The underlying rationale is straightforward: Renaming Objective-C class names and methods to strings of equal length often employs random or generated names. This intentional obfuscation adds complexity for reverse engineers, which requires more effort to decipher the meaning of classes without the aid of descriptive names.

The Poor Man's Obfuscator [18] employs a more intricate obfuscation scheme. In this approach, various load commands are altered to feed incorrect information to binary analysis platforms. Obfuscation options, called transformations, include actions such as randomizing the names of exported symbols, redirecting the addresses of exported symbols to different locations, adjusting the offsets and sizes of sections in load commands, and modifying entries in the LC_FUNCTION_START table. Upon scrutiny, this obfuscation scheme introduces moderate disruptions that challenge many binary analysis platforms.

4 Loader Simulation

In this section, we present an in-depth exploration of our loader simulation technique for Mach-O binaries. This concept will be used for the obfuscation and hooking techniques described in later sections.

4.1 Design Overview

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 input for static analysis tools such as IDA or Ghidra. The removal 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 PE binaries [8].

We introduce a control-flow intervention between the binary loading process and its execution. This is necessary because the loader cannot perform a full load of the binary due to missing information that results in a crash during execution. To keep the binary working as normal, the intervened code performs a part of the loader's workflow using the extracted information. This process is later termed *restoration logic*. This intervening control flow is inserted through an external library or is injected, as described in [11].

In this paper, we use an external library to perform restoration logic. Information needed for the loading process of the binary is extracted, which is then included in the external library. The binary is also added with a load command to load our external library.

As an optional measure, the extracted data can undergo static encryption, with decryption occurring at load time when our restoration logic is executed. Note 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 by jumping to the direct address.

4.2 Extracting information

To simulate the loader, the information used by the loader is considered for extraction. This information is typically stored in _LINK_EDIT segment. Commands that use these 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 function pointers; these are often called before the binary's main procedure.

LC_DYLD_INFO_ONLY load command provides information in the form of bytecode chains. If the binary has this load command, we eliminate the non-lazy, lazy and weak bytecode chains by rewrite the data so that the loader would skip through. 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 fixups chains. By traversing these chains, 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 each chain dynamic symbol values to a rebase values. Subsequently, we completely eliminate the string table that holds the symbol names.

In both situations, the loader should still be able to process the binary without crashing. If the table is empty, for bytecode chains, the loader can skip reading the dynamic symbols. Similarly, in the case of chained fixups, they can be resolved as a rebase value. Thus, the binary can be loaded into memory successfully, although with unresolved dynamic symbols referencing the wrong address.

Constructor functions refer to functions that are invoked by the loader once all images have been loaded into memory. These functions are called sequentially as part of the initialization process. To remove these functions, several methods are 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 that they reside within the binary's memory region. When adjusting these pointers to point outside the memory region, the loader will disregard them, effectively achieving removal of these functions.

4.3 Simulate Loader

During runtime, extracted information is retrieved and used to complete the loading of the binary. We use a constructor function that is scheduled to run (by dyld) before the main executable. Using the entire set of parameters provided by the loader, we are able to determine the base address of the main executable that has been loaded into memory, as depicted in Listing 3.

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.

The load commands **`LC_DYLD_INFO_ONLY`** and **`LC_CHAINED_FIXUPS`** have different representations, but both contain a collection of dynamic symbols. Each symbol in the collection includes the symbol name, the exporting library, and the address where the function pointer is stored. By extracting data from these load commands, we generate a dynamic symbol list. During runtime, we iterate through this list to locate the symbol and update the function pointer. The symbol can be easily found using `dlsym`. Writing to the function pointer requires that the address be writable, as described in Listing 1, because the loader locks the `__DATA_CONST` segment as read-only after it finishes and our restoration logic performs after the loading process.

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

Listing 1: Modify the virtual memory range from `offset` to `offset+size` to Read-Write.

Constructor functions can be invoked directly. We can calculate the function addresses and invoke them with parameters passed to our constructor because the loader consistently passes the same arguments to all constructors during initialization, enabling us to call these functions manually without reliance on the loader.

4.4 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 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 diving into these nuances, it is essential to clarify the synergy between Objective-C and the `dyld` loader.

4.4.1 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 used 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`.

By the callback `_dyld_objc_callbacks_v1`, the Objective-C runtime registers three functions at different 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 unmapped from memory.

4.4.2 Objective-C data in binary

The binaries compiled from Objective-C 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 summary, Objective-C runtime performs the initialization of class objects and selectors when the binary is mapped to memory, and Objective-C `+load` methods are called during constructor invocation.

Objective-C binaries contain class definitions represented as data. These classes are defined using two special pointers: `isa` and `superclass`. The `isa` represents the metaclass, while the `superclass` represents the parent class. Each class has its own metaclass, and the `superclass` pointer points to the class data of the parent class. It is important to note that the `superclass` pointer can never be null because all classes in Objective-C must inherit from `NSObject`. Further details of Objective-C data stored in binary is discussed in Section 6.2.

4.4.3 Simulation for Objective-C based binaries

Dynamic symbols in Objective-C, including classes that refer to other classes in different libraries, are also considered dynamic symbols. Our simulated loader eliminates these symbols. However, the Objective-C runtime workflow requires the loading of these classes. Furthermore, the *Foundation* library must be initialized before it can be referenced. The current restoration logic is implemented prior to any Objective-C runtime and *Foundation* initialization, which would lead to crashes. To address this issue, we incorporate a shellcode snippet to redirect the execution of the main function until all Objective-C classes have been resolved.

We prevent Objective-C runtime from executing its class loading mechanism by modifying the names of two sections in the binary: `__objc_classlist` and `__objc_nlclslist`. We insert a shellcode before the start of the `_TEXT` segment and edit the `LC_MAIN` load command to points to the shellcode location.

The shellcode is created to be compact, with the primary objective of executing a function that resolves all Objective-C classes, referred to as **`restore_objc`**, and jumps to the main function after it finishes. Figure 3 illustrates components of `dyld` and `objc4` to be simulated.

Objective-C logic for processing class data is done through private 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 symbol addresses and rebuild the logic as described in `_read_images`[1] protocol of the Objective-C runtime.

In iOS environment, these symbols are not declared in `LC_SYMTAB`. However, we can try to locate these symbols indirectly through public symbols that invoke them.

The process of locating the `restore_objc` procedure can be complex. 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. We visualize the process in Figure 4.

In practical situations, the space available for the shellcode before the `_TEXT` segment and the number of pointer values after the `_DATA` segment may be limited. Therefore, it is important to keep the shellcode as concise as possible and reduce the number of required pointer values. Generally, there should be enough space available since these segments are page-aligned, unless the code and data sizes are exact multiples of the page size, which would result in no extra space. If the available space is insufficient, it is recommended to use [11] or similar methods to add additional code in the binary for the purpose of restoration logic.

5 Obfuscation

Binary executable files encapsulate assembly instructions, program data, and essential execution information for the operating system to execute. 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 (usually commercialized) [8, 13, 17] and Linux binaries [6, 9, 10, 15, 18], while Apple binaries have received comparatively limited attention. Some previous work [4, 14, 18] has proposed modest adjustments to debugging information within binaries, which helps to impede the analysis capabilities of reverse engineering platforms.

Another widely recognized form of binary obfuscation is known as "packing" [12]. This method involves compressing or encrypting the binary code and then unpacking it at runtime. However, it is 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.

We have created a method of obfuscation that relies on the simulated loader described above. Beyond the extraction of dynamic symbols, it is possible to eliminate further information as outlined in 5.1. To enhance security, we recommend the encryption of the symbol table within the simulated loader. Implementing the additional procedures referred to in Appendix A results in obfuscation of the simulated loader itself, providing an additional level of protection.

5.1 Removing redundant information

Some data are automatically generated during the compilation process by the compiler and the linker. 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 a basic analysis that relies on these debug symbols to make sense of the binary program.

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 running.

Our technique also allows for the removal of load commands related to system libraries. These libraries are always present in memory and can be accessed by any process. The inclusion of these load commands in the binary is only necessary for referencing dynamic symbols. However, since our obfuscation method extracts all dynamic symbols from the binary, the references to system libraries can also be eliminated.

[1] https://github.com/apple-oss-distributions/objc4/blob/objc4-912.3/runtime/objc-runtime-new.mm#L3927

6 Hooking

Hooking is a prominent topic in the security field and has evolved with tools such as Frida [16] and Fishhook [5]. 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 have described here is distinct from Frida's approach, rather it is quite similar to Fishhook but on a binary level. Although we do not 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.

6.1 C API hooking

As explained in Section 4, our obfuscation technique resolves dynamic symbols at runtime. This provides an opportunity to intercept the C API. The concept is straightforward: The symbol that needs to be intercepted is manually resolved, and the function pointer is then overwritten by our intercepted function.

The concept is comparable to Fishhook and `dyld interposing`, as it involves altering the API function pointer to a different function. However, unlike Fishhook, we do not need to have access to the program's source code. Additionally, unlike `dyld interposing`, we manually install the hook without relying on the dyld API.

6.2 Objective-C class method hooking

As stated in Section 4.4.2, the metadata for each class is stored statically in the binary. These metadata contain references to the metaclass, the parent class, and a struct called `class_ro`. When loading the class, the Objective-C runtime reads this struct to initialize the class prototype. The prototype includes references to the class name, the method list, the property list, the ivars, and other information. Figure 1 illustrates how the data references are stored in an Objective-C binary. Hooking can be achieved through manually editing the data of Objective-C binary.

We thoroughly examine the class structure within the binary. The class data are kept in a section named `_objc_data`. Sections `_objc_classlist` and `_objc_classref` hold pointers to this class data list. The class list is used by the Objective-C runtime to initialize classes in memory, while the class reference list is utilized in the binary (e.g., to create a class instance). The entries in the class list are of type `class_t` and contain three special pointers: *isa* (pronounced "is a") points to the metaclass, *superclass* points to the direct parent class, and *ro* points to internal class data (such as methods and attributes). `ro` points to a struct of type `class_ro_t` inside `_objc_const` section. This struct contains the pointers to a list of methods for a class. Each method list entry is a tuple of three values, the selector pointer, the selector name (method's name), and the implementation pointer (function pointer offset). This list can be encoded in 4 different types (big, small, bigSigned, bigStripped)[2]; regardless, the information they convey is the same.

From the overall structure detailed in the previous paragraph, we can deduce many hooking mechanisms. We enumerate the four straightforward approaches below and implemented two of them, first and third. All hooking mechanisms must occur prior to loading the class in memory (processed by Objective-C runtime).

1. Modifying the implementation pointer in method list.

   This method is the most straightforward, enabling the hooking of a specific function by altering its reference to point to a different function.

2. Create a method list and replace them in `class_ro_t`.

   The method list can be substituted with a different (manually crafted) list. References to the function's implementation are modified to use hooking functions. The name and type of selectors should remain unchanged.

3. Redirect pointers in `_objc_classref` section.

   For ease of use, we can define a new class containing the methods we intend to hook and then update the `_objc_classref` pointer to reference this new class. By leveraging the `superclass` property, we set the original hooked class as the superclass. This makes the newly created class a subclass of the hooked class, causing hooked methods to be invoked by default while unhooked methods are passed through to the superclass. Hooked methods can also invoke their unhooked versions by using the superclass call `[super method:]`.

   This applies to classes that are imported from other libraries. All imported classes are given as pointers within `_objc_classref`, with their symbols being resolved by the loader.

4. Create a `class_ro_t` struct and replace them in `_objc_data`.

   The class data are accessed via the `ro` pointer, allowing us to generate a personalized instance of `class_ro_t` and replace the `ro` pointer.

7 Evaluation

The Mach-O modification methods and techniques for obfuscation and hooking that we propose are highly intricate.

[2] https://github.com/apple-oss-distributions/objc4/blob/objc4-912.3/runtime/objc-runtime-new.h#L963

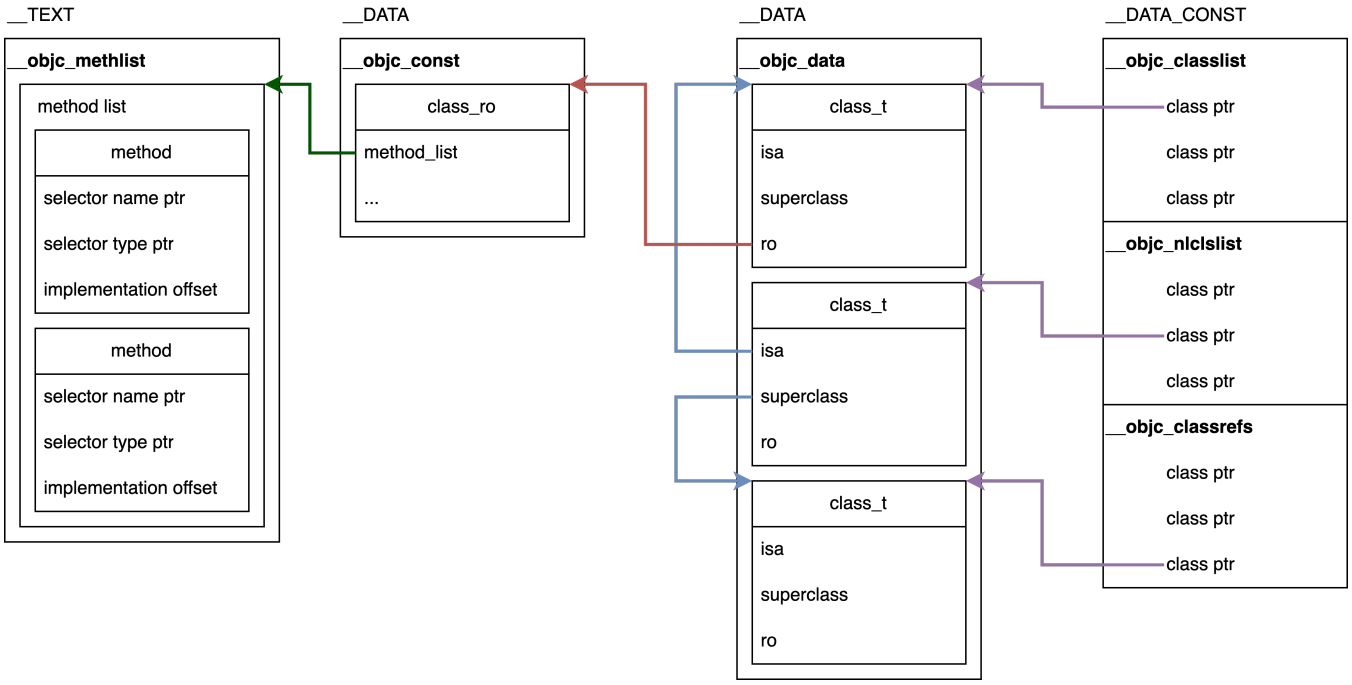

Figure 1: Objective-C class metadata stored in binary. This figure offers a comprehensive structural overview of basic Objective-C data, though it does not encompass notions like metaclass or class properties.

Nevertheless, we anticipate that the modified binary will remain unchanged before and after obfuscation; or the behavior of the hooked functions will be modified when hooking is implemented. Other elements to be assessed include performance, particularly the startup delay and the increase in size. In the scenario of obfuscation, we foresee that all symbols will be removed from the executable.

During our assessment, it became apparent that our study is a pioneer in performing an exact evaluation of Mach-O binary modifications. Consequently, no pre-existing sample set has been selected for bench-marking. We decided to select GNU Coreutils as our suite for bench-marking.

7.1 Coreutils bench-mark

We performed some benchmark tests on the GNU Coreutils suite, a collection of command-line tools that are widely used and can be found in nearly every Linux-based system, including Apple devices. During the execution of the obfuscation scheme experiments, our imlementation failed to resolve the symbols of `/usr/local/opt/gettext/lib/libintl.8.dylib`. This library is loaded using a different path and not as indicated[3]. In order to proceed with the experiment, we have opted to keep these symbols and let them be resolved by the loader.

[3]To locate libraries in memory, use `LC_ID_DYLIB` to identify the library name, not the path

We evaluated the restoration time for imported symbols. This time calculation is based on our current implementation, which can be further optimized. The results are presented in Figure 2. The variation in time measured for the same number of imports can be attributed to the search order. Our implementation does not optimize the search by caching previously found occurrences or prefixes. Moreover, each search starts anew, even though basic hashing is employed to accelerate the search for the library export trie in a (unoptimized) hash table implemented in C.

In the process of obfuscation, we conducted a comprehensive enumeration of all symbols present in the binaries following the obfuscation procedure. All obfuscated binaries were removed of their original symbols, with the sole exception being symbols originating from `/usr/local/opt/gettext/lib/libintl.8.dylib`, the rationale of which has previously been explained. Assuming the process is executed with precision, all symbols should be eliminated, resulting in a binary that is entirely stripped, including imported symbols.

We conducted tests on an Intel Mac 14 with GNU Coreutils version 9.1 (commit *ca22b9e*). Of 105 utilities, 2 failed to build, specifically *make-prime-list* and *extract-magic*. These commands are used infrequently today, so we proceeded with testing regardless. We anticipated that the remaining 103 utilities would be successfully obfuscated and functional. To our surprise, checksum-related utilities (*md5sum*, *sha1sum*,

sha224sum, *sha256sum*, *sha384sum*, *sha512sum*) triggered a `SEGFAULT` signal when invoking `set_locale` at the start of the program. Upon investigation, we found several unknown factors that contributed to this issue, including the handling of *libintl* and the discrepancy between the uppercase symbols in the search list and the lowercase names in the library. Despite this, the other **97** utilities performed correctly, giving more than 90% of the utilities in Coreutils that are obfuscatable. As a proof of concept, the overall results have demonstrated viability, allowing us to conclude the evaluation with a few considerations.

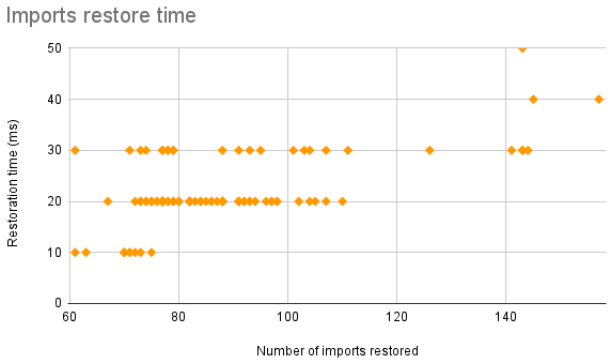

Figure 2: Restoration time for binaries in Coreutils

7.2 iOS Viability

We successfully obfuscated a basic iOS application created with Xcode using only Objective-C. The general method stayed the same. However, certain Objective-C APIs are not accessible via the `symtab`, necessitating manual symbol searches. Despite this, the iOS application was obfuscated, with imported symbols eliminated, and was successfully restored during runtime and running normally. In Figure 5, we compare side by side the before and after obfuscation of the main function.

7.3 Reversing and Rebuilding symbol tables

A crucial element of obfuscation is the capacity to deconstruct the obfuscated variant. As outlined earlier, our method involved transferring the dynamic symbol table into our emulated loader, which made the calls to external libraries indeterminate. The program logic can be ordinarily reversed, as there are no attempts to modify the control flow. If the attackers aim to comprehend the segment of code where calls are made to external libraries, they must either retrieve the symbol table or determine the symbol invocation during execution.
In an environment without protection, the symbol table can be retrieved from the emulated loader. However, this table is not identical to the bytecode chain or fixups chain. Attackers need to understand how to rebuild the table and reapply it to the binary for automatic analysis using analysis tools, or they must manually decode each symbol and its address and create a lookup table. Attackers may also connect a debugger to monitor invocations of symbols from external libraries, but they must manually track all symbols. Memory extraction strategies may be effective, but the addresses obtained are only valid for one session, unless the symbol is part of the system libraries that are always loaded in memory. Attackers must map these addresses to their library and symbol name, then reconstruct the bytecode chain or fixups chain, and reapply them to the binary. In a highly protected environment, where the symbol table is encrypted and runtime application self-protection (RASP) is in place, recovering the symbol table would be more challenging.

7.4 Limitations

Our method has several drawbacks. One major limitation is its unsuitability for statically linked binaries. Most of the elements that our simulated loader aims to load are dynamic symbols, which are missing in statically linked binaries. However, in modern times, dynamically linked binaries are almost universally used, except in some restricted environments like those in IoT devices.
Another constraint is the presumptions of the unutilized space for shellcode, placed between the header and `_TEXT` segment, and 2 pointers, positioned after `_DATA` segment. Despite having a compact shellcode, there are instances where the shellcode cannot be fully inserted. To avoid this issue, we also eliminate the load commands for libraries (`LC_DYLIB`) that belong to the system, such as `libSystem` and `libc`. These commands are only incorporated as a library index when the loader decodes the bytecode chain or the fixups chain. Therefore, they can be discarded because the library has already been loaded into memory.
Within the iOS setting, the symbol table (symtab) of the Objective-C runtime library is devoid of its original content, making it impossible to locate the symbols `readClass`, `realizeClassWithoutSwift`, `remapClass` within it. However, since these functions are invoked from other exported symbols, we have the option to conduct a manual search for these symbols by examining the assembly instructions and pinpointing the calls that are likely to be these symbols, using a heuristics source-assembly mapping.

8 Future Works

At present, we offer the simulated loader as an independent dynamic library. This strategy may have some disadvantages, as the binary is now required to be accompanied by our library. To address this, we can utilize the method described

in [11]. However, this study only details for the case of bytecode chains and not for the fixups chain, as the fixups chain is appended subsequently. Regardless, the concept remains the same, we can incorporate our dynamic library into the target binary to create a unified binary instead of having an additional binary.

9 Conclusions

This paper introduces a method for modifying Mach-O binaries, which involves the creation of a simulated loader that can be expanded into either an obfuscation scheme or a hooking technique. These two approaches can be readily applied to dynamically linked Mach-O binaries and can be utilized on binaries that contain or import Objective-C classes. The impact on startup time is negligible and does not compromise the overall performance of the targeted binary. Our method is compatible with GNU Coreutils and is anticipated to function with actual-world binaries.

References

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

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

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

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

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] Do Minh Tuan Nguyen Anh Quynh. Redback: Advanced Static Binary Injection. URL: https://groundx.io/redback/.

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

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

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

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

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

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

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

A Obfuscate the simulated loader

A.1 Manual `dlsym`

The loader's ability to resolve symbols into function addresses using either `LC_DYLD_INFO_ONLY` or `LC_CHAINED_FIXUPS` is well understood. Essentially, the loader maintains a list of loaded libraries and utilizes the export trie of each library to find public symbols based on their names. However, it is important to note that in some cases, a function may be re-exported from another library. In such situations, a recursive search through libraries is necessary to locate the address of the function. This recursive search ensures that the loader can accurately resolve symbols even when they are re-exported from different libraries. Occasionally, the re-exported symbol may also be renamed, requiring subsequent searches to use the new name.

When searching for symbols, it is crucial to consider that symbols may refer to their hosting library using relative paths. These relative paths can be expressed as either directory-relative paths or through path variables such as `@rpath`, `@executable_path`, or `@loader_path`. To ensure precise

resolution of these relative paths, it is recommended to convert them into their respective full paths.
In order to obtain the list of loaded libraries in memory, a series of three symbols can be used: `_dyld_image_count`, `_dyld_get_image_header`, and `_dyld_get_image_name`. By invoking these symbols sequentially, a comprehensive list of loaded libraries can be compiled. This functionality is demonstrated in Listing 2. More 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.

It should be noted that the file path obtained from `_dyld_get_image_name` and the library name specified in the `ID_DYLIB` load command may not match. `dyld` examines the `LC_DYLIB` load commands to determine which library to load based on the `ID_DYLIB` value.

A.2 Obfuscate the simulation library

Our obfuscation method can also be applied to the restoration library, which is also a Mach-O binary. During the obfuscation process, all symbols are removed and need to be reinstated during runtime. As the library is known to be executed first, we can utilize this opportunity to resolve our library. However, a drawback of this approach is the restricted usage of dynamic symbols.
In order to address the issue of restricted usage of dynamic symbols and ensure the restoration of the external library, a possible solution is to use one dynamic symbol as a point of reference and locate the header of the dynamic library through this reference point. Following this rationale, we can locate the header of the `dyld` library by selecting any dynamic symbol of our preference, such as `dyld_get_sdk_version`. Only `dyld` is needed because from `dyld`, we can resolve remaining symbols through manual `dlsym` as specified in Appendix A.1.
During the extraction phase, all symbols except for `dyld_get_sdk_version` are removed. This specific symbol is used to locate the `dyld` library in memory. The extracted information is then written into a new section of the library called `_EXTRACTED` for easy access. By obtaining a reference to `dyld`, we can restore the function pointers of dynamic symbols using the information stored in `_EXTRACTED`.
By executing the aforementioned procedures, both the obfuscation binary and the restoration library become completely obfuscated. The obfuscated versions of these components reveal very few symbols, and in the case of an Objective-C compiled binary, references to classes are also eliminated.

B Code snippets

```c
#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 2: Get a list of loaded libraries

```
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.

```
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.

```
adr x8, 0
movz x9, #0x9999
add x8, x8, x9
stp x30, x8, [sp], #-0x10
stp x3, x2, [sp], #-0x10
stp x1, x0, [sp], #-0x10
ldr x9, [x8]
blr x9
ldp x1, x0, [sp, #0x10]!
ldp x3, x2, [sp, #0x10]!
ldp x30, x8, [sp, #0x10]!
ldr x9, [x8, #8]
br x9
```

```
lea    r8,[rip+0x0]
mov    r9,0x4030201
add    r8,r9
push   rdi
push   rsi
push   rdx
push   rcx
push   r8
mov    r9,QWORD PTR [r8]
call   r9
pop    r8
pop    rcx
pop    rdx
pop    rsi
pop    rdi
mov    r9,QWORD PTR [r8+0x8]
jmp    r9
```

Table 3: Shellcode inserted: ARM64 on the left; Intel 64 on the right. The second instruction of both versions is the offset from the shellcode to the end of `__DATA` section. Making the pop on 2 words value of `r8` (in Intel 64) or `x8` (in ARM64) be the address of the `restore_objc` and the main function.

```
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 3: Using **ProgramVars** struct

```
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 4: Searching for Mach-O base address

C Figures

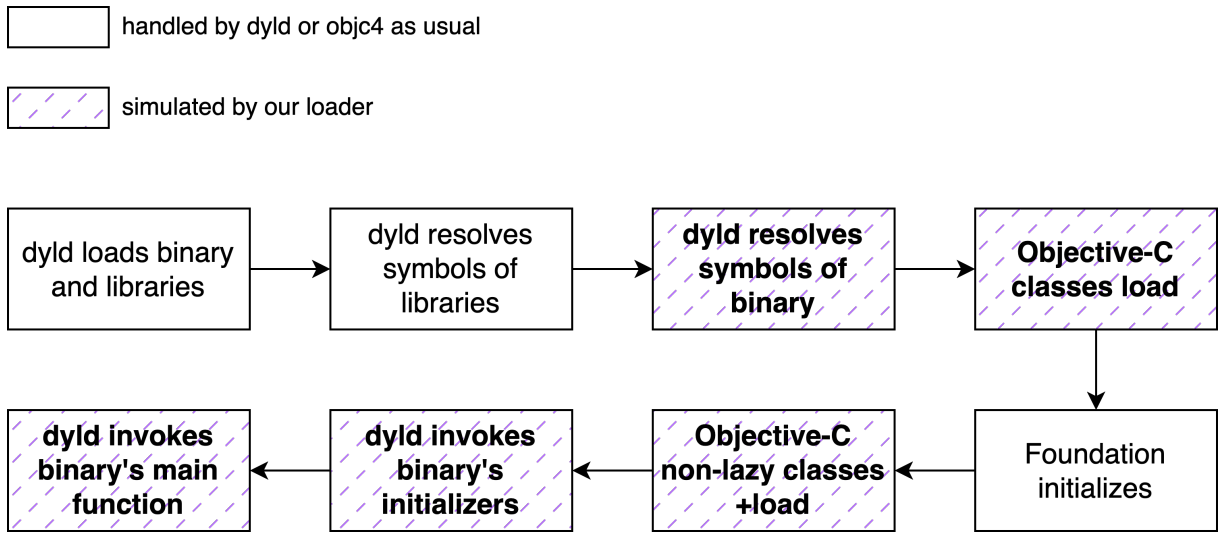

Figure 3: This figure illustrates the components of *dyld* and *objc4* that are simulated by our loader.

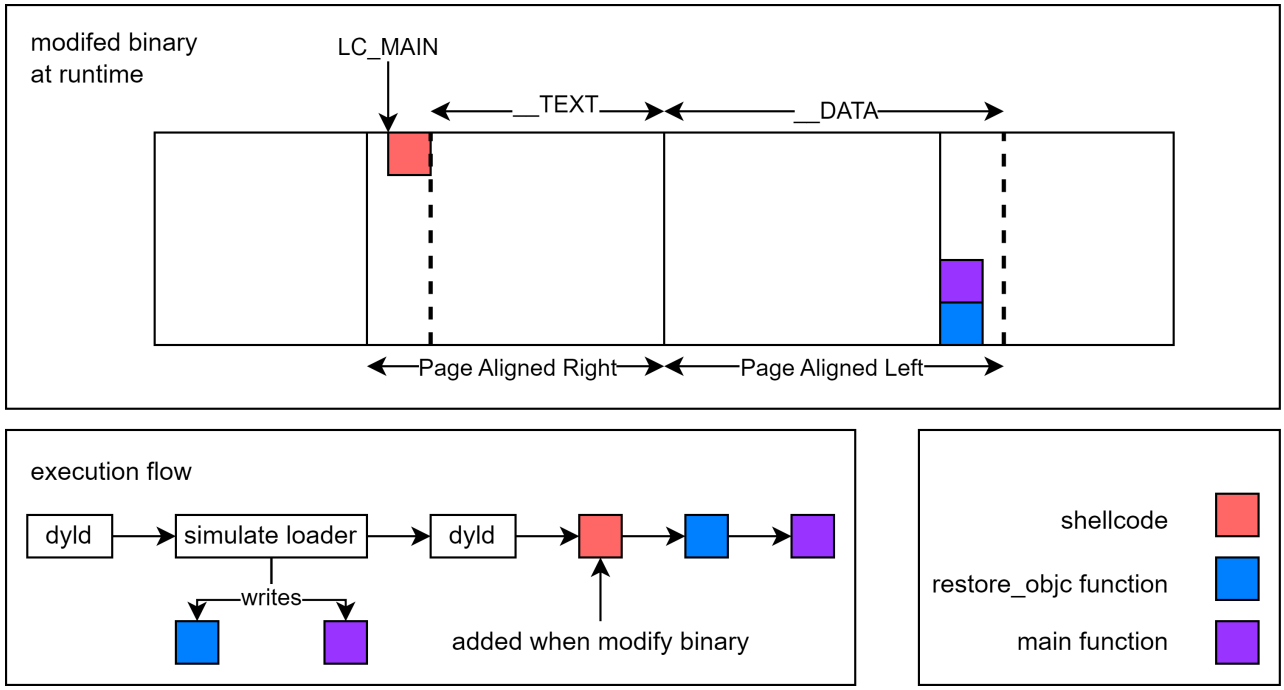

Figure 4: This figure illustrates the modified binary at runtime and the execution flow. `LC_MAIN` is modified into the inserted shellcode. During execution of the simulate loader, it writes the address of `restore_objc` and `main` function at the end of `__DATA` section. The shellcode now acts as the new main for the binary is invoked by `dyld` and calls `restore_objc` and binary's `main` functions.

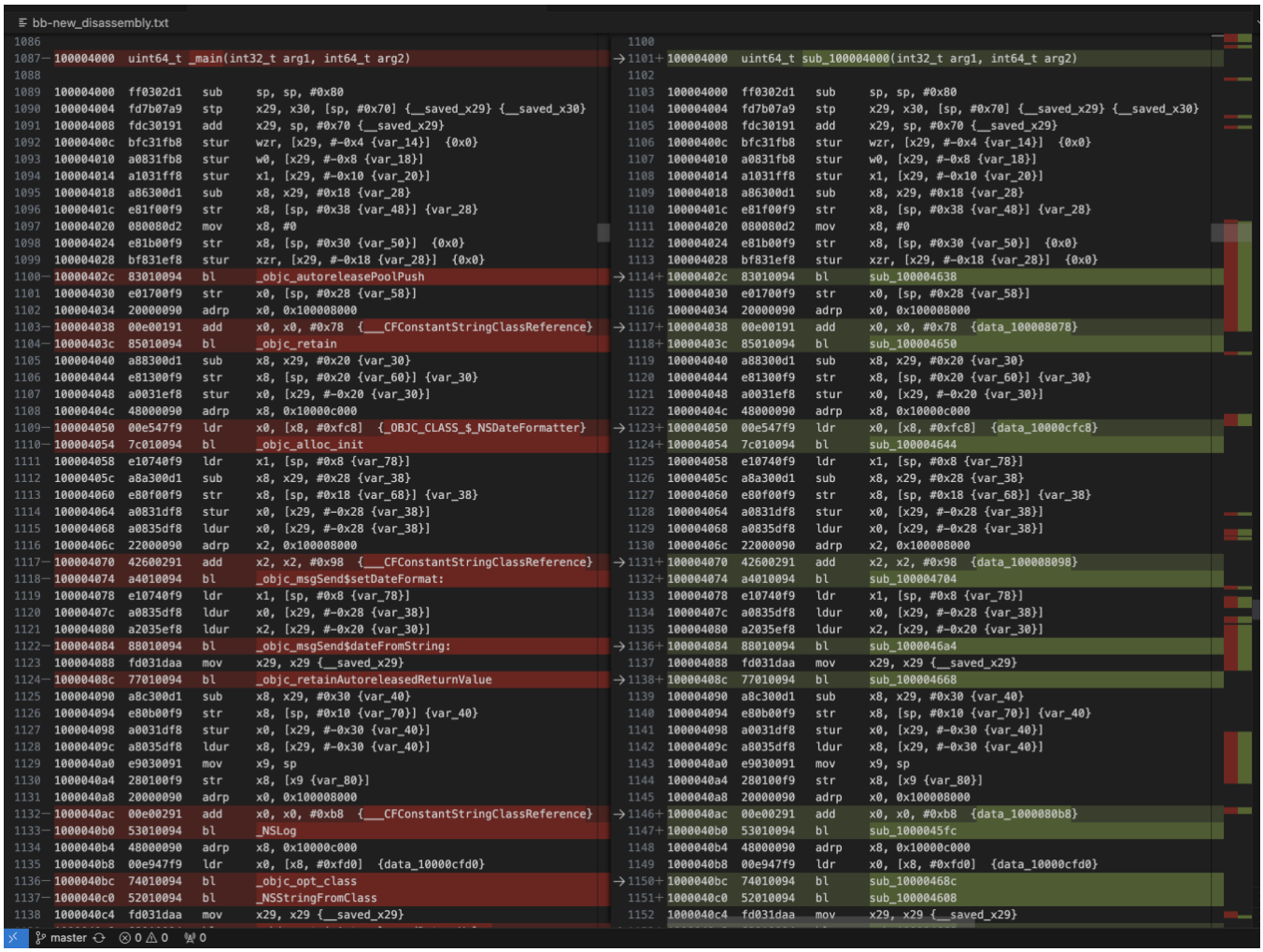

Figure 5: Disassembly (generated by Binary Ninja) comparison of the original file (left) and the obfuscated file (right). The depicted file represents the main function of a standard **iOS application** created using XCode. The image clearly shows that the obfuscated version has its symbols stripped away.

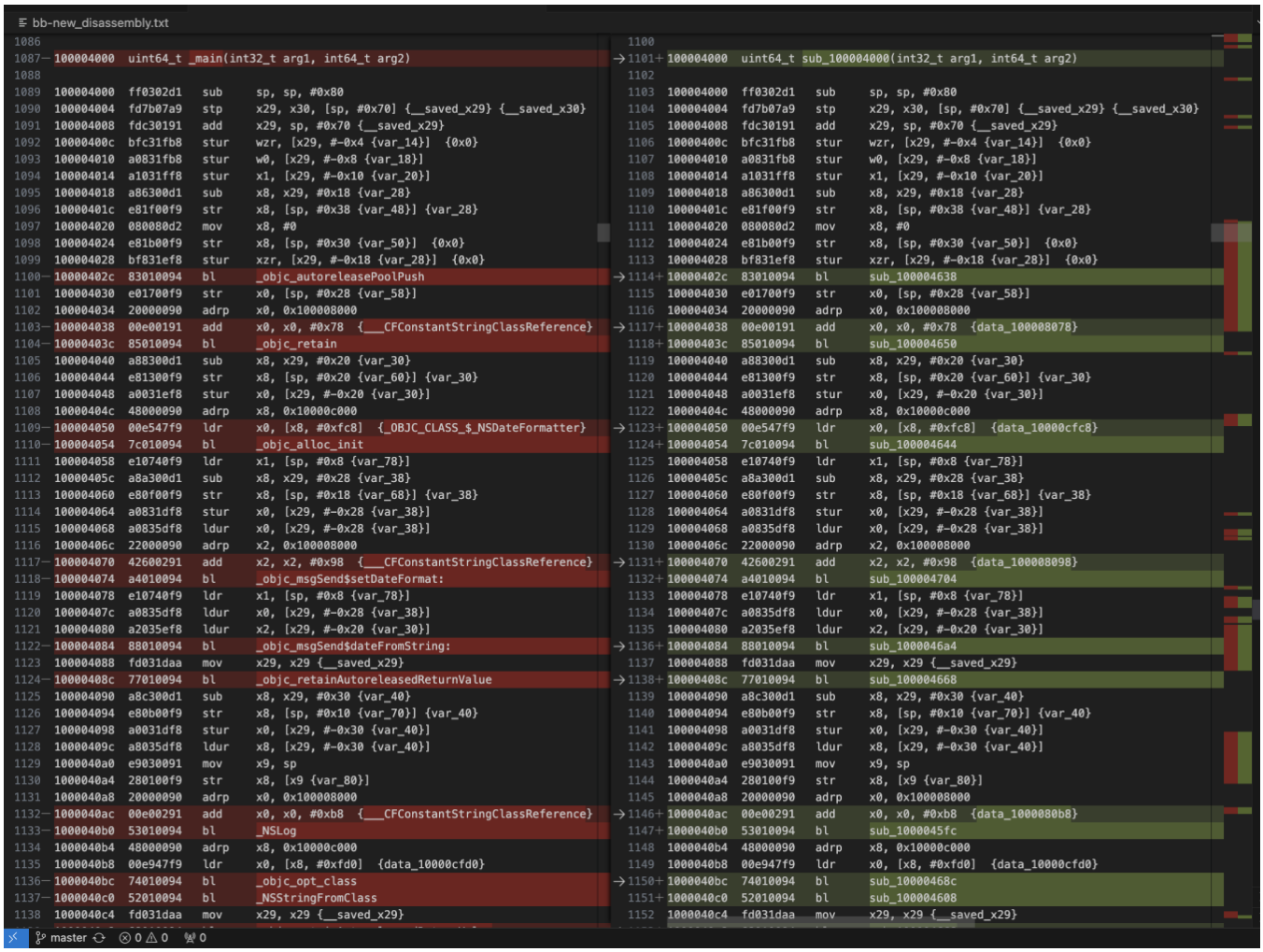

Figure 6: Disassembly (generated by Binary Ninja) comparison of the original file (left) and the obfuscated file (right). The depicted file represents the main function of a **Intel Mac macOS application**. The image clearly shows that the obfuscated version has its symbols stripped away.

```
100003fcc  int64_t _start()

100003fcc  08000010   adr     x8, 0x100003fcc
100003fd0  895292d2   mov     x9, #0x9294
100003fd4  0801098b   add     x8, x8, x9  {data_10000d260}
100003fd8  fe23bfa8   stp     x30, x8, [sp], #-0x10 {__saved_x30} {arg_8}  {data_10000d260}
100003fdc  e30bbfa8   stp     x3, x2, [sp], #-0x10 {var_10} {var_8}
100003fe0  e103bfa8   stp     x1, x0, [sp], #-0x10 {var_20} {var_18}
100003fe4  090140f9   ldr     x9, [x8]  {data_10000d260}
100003fe8  20013fd6   blr     x9
100003fec  e103c1a9   ldp     x1, x0, [sp, #0x10]! {var_20} {var_18}
100003ff0  e30bc1a9   ldp     x3, x2, [sp, #0x10]! {var_10} {var_8}
100003ff4  fe23c1a9   ldp     x30, x8, [sp, #0x10]! {__saved_x30} {arg_8}
100003ff8  090540f9   ldr     x9, [x8, #0x8]  {data_10000d268}
100003ffc  20011fd6   br      x9
```

Figure 7: Shellcode inserted for the iOS application in Figure 5.

```
100003654  int64_t _start()

100003654  4c8d0500000000     lea     r8, [rel data_10000365b]
10000365b  49c7c1294d0000     mov     r9, 0x4d29
100003662  4d01c8             add     r8, r9  {data_100008384}
100003665  57                 push    rdi {var_8}
100003666  56                 push    rsi {var_10}
100003667  52                 push    rdx {var_18}
100003668  51                 push    rcx {var_20}
100003669  4150               push    r8 {var_28}  {data_100008384}
10000366b  4d8b08             mov     r9, qword [r8]  {data_100008384}
10000366e  41ffd1             call    r9
100003671  4158               pop     r8 {var_28}
100003673  59                 pop     rcx {var_20}
100003674  5a                 pop     rdx {var_18}
100003675  5e                 pop     rsi {var_10}
100003676  5f                 pop     rdi {var_8}
100003677  4d8b4808           mov     r9, qword [r8+0x8]  {data_10000838c}
10000367b  41ffe1             jmp     r9
```

Figure 8: Shellcode inserted for the macOS application in Figure 6.